

Учебник по Haskell

Автор: Антон Холомьёв
Год: (CC) 2012
Лицензия: [Creative commons Attribution-NonCommercial-NoDerivs](#) 3.0
Generic (CC BY-NC-ND 3.0)
Связь [anton.kholomiov на gmail.com](mailto:anton.kholomiov@gmail.com)

Добро пожаловать в мир функционального программирования, строгой типизации, чистых функций, ленивых вычислений и классов типов. Всё это вы найдёте в замечательном языке программирования Haskell. Если вы просто где-то слышали о Haskell, и пока это описание ни о чём не говорит, ничего, об этом и о многом другом вы узнаете со страниц этой книги.

Haskell был основан на исходе восьмидесятых, как общий язык для программистов, интересующихся функциональным программированием и ленивой стратегией вычислений. Это свободный язык, он разрабатывается комитетом разработчиков, программистов, математиков, информатиков, и просто увлечённых программированием людей. Основной компилятор языка GHC разрабатывается в Microsoft Research, но несмотря на это легко доступен в интернет. Вы можете установить его через [Haskell Platform](#) (для начинающих) или напрямую с сайта [GHC](#) (для гиков и любителей квестов). Ещё нам понадобится редактор с подсветкой синтаксиса Haskell. Подойдёт простой [gedit](#) или более тяжёлые vim и Emacs. Есть и IDE для Haskell. Это [Leksah](#), но пока ещё она совсем юная.

Итак, устанавливаем компилятор GHC подбираем редактор по вкусу и [в добрый путь!](#)

Сообщить об ошибке

Если вы заметили ошибку или неточность, или явных промах в изложении, вы можете сообщить об этом [здесь](#). Прошу терпения с ответом, скорее всего, я не отвечу сразу, возможно, совсем не отвечу.

Содержание

- [Предисловие](#)
- [1. Основы](#)
- [2. Первая программа](#)
- [3. Типы](#)
- [4. Декларативный и композиционный стиль](#)
- [5. Функции высшего порядка](#)
- [6. Функторы и монады: теория](#)
- [7. Функторы и монады: примеры](#)
- [8. IO](#)
- [9. Редукция выражений](#)
- [10. Реализация Haskell в GHC](#)
- [11. Ленивые чудеса](#)
- [12. Структурная рекурсия](#)
- [13. Поиграем](#)
- [14. Лямбда-исчисление](#)
- [15. Теория категорий](#)
- [16. Категориальные типы](#)
- [17. Дополнительные возможности](#)
- [18. Средства разработки](#)
- [19. Ориентируемся по карте](#)
- [20. Императивное программирование](#)
- [21. Музыкальный пример](#)
- [Приложение](#)

Предисловие

История языка Haskell начинается в 1987 году. В 1980е годы наблюдался всплеск интереса к ленивой стратегии вычислений. Один за другим появлялись новые функциональные языки программирования. Программисты задумались и решили, объединив усилия, найти общий язык. Так появился Haskell. Он был назван в честь одного из основателей комбинаторной логики Хаскеля Кэрри (Haskell Curry).

Новый язык должен был стать свободным языком, пригодным для исследовательской деятельности и решения практических задач. Свободные языки основаны на стандарте, который формулируется комитетом разработчиков. Дальше любой желающий может заняться реализацией стандарта, написать компилятор языка. Первая версия стандарта была опубликована 1 апреля 1990 года. Haskell продолжает развиваться и сегодня, было зафиксировано два стандарта: 1998 и 2010 года. Это стабильные версии. Но кроме них в Haskell включается множество расширений, проходит обкат интересных идей. Сегодня Haskell переживает бурный рост, к сожалению, эпицентры далеки от России, это Англия, Нидерланды, Америка и Австралия. Интерес к Haskell вызван популярностью многопроцессорных технологий. Модель вычислений Haskell хорошо подходит для распараллеливания. И сейчас проводятся исследования в этой области.

Haskell очень красивый и лаконичный язык. Он придётся по душе математикам, программистам, склонным к поиску элегантных решений. В арсенале программиста: строгая типизация с выводом типов, функции высшего порядка, алгебраические типы данных, алгебраические структуры. Если пока всё это звучит как набор слов, ничего страшного, вы узнаете что это по ходу чтения книги.

Структура книги

Haskell славится высоким порогом вхождения. Он считается трудным языком для начинающих. Во многом это связано с тем, что начинающие уже имеют приличный опыт программирования на императивных языках. И при первом знакомстве оказывается, что этот опыт ничем не может им помочь. Они не могут найти в Haskell аналогов привычных синтаксических конструкций и приёмов программирования. Haskell сильно отличается от распространённых языков программирования. Но если вы совсем-совсем начинающий, скорее всего в этом плане вам будет гораздо проще. Если вы всё же не начинающий, попробуйте

подойти к материалу этой книги с открытым сердцем. Не ищите в Haskell элементы вашего любимого языка и, возможно, таким языком станет Haskell.

Ещё одна трудность связана с тем, что многие понятия тесно переплетены, Haskell не так просто разбить на маленькие части и изучать их от простого к сложному, уже в самых простейших элементах кроются черты новых и непривычных идей. Но, я надеюсь, что мы сможем преодолеть и этот барьер, мы не будем изучать Haskell по кусочкам, а окунёмся в него с головой, уже в первой главе мы пробежимся по всему языку и далее будем углубляться в отдельные моменты.

В книге много примеров. Haskell оснащён интерпретатором. Интерпретатор (также называемый REPL, от англ. read-eval-print loop) позволяет писать программы в диалоговом режиме. Мы набираем выражение языка и сразу видим ответ – вычисленное значение. Интерпретатор поможет нам разобраться во многих тонкостях языка. Мы будем обращаться к нему очень часто.

Книгу можно разбить на несколько частей:

- Основы языка (1-13). Из первых тринадцати глав вы узнаете, что такое Haskell и чем он хорош.
- Теоретическая часть (14-16). Haskell питается соками математики, многие красивые научные идеи не только находят в нём воплощение, но и являются фундаментом языка. Из этих глав вы узнаете немного теории, которая служила источником вдохновения разработчиков Haskell.
- Разработка на Haskell (10,17-20). В этих главах мы познакомимся с расширениями языка (17), мы узнаем как писать библиотеки и документацию (18), проводить тестирование и оценивать быстродействие программ (19), также мы потренируемся в написании императивного кода на Haskell (20). Из главы 10 мы узнаем как работает GHC, основной компилятор для Haskell.
- Примеры (13, 21). В этих главах мы посмотрим на несколько примеров применения Haskell. В главе 13 мы напишем программу для игры в пятнашки, а в главе 21 – midi-секвенсор и немного музыки.

Рекомендую сначала изучить основы языка, а затем обращаться к остальным частям в любом порядке.

Основные понятия

Haskell – чисто функциональный, типизированный язык программирования. Я буду очень часто говорить слова функция, типы, значения, типы, функция, функция, типы— буквально постоянно. Перед тем как мы окунёмся с головой в программный код, я бы хотел словами пояснить, что всё это значит.

Мы собираемся изучить новый язык, хоть и искусственный, но всё же язык. Языки служат описанию явлений, словами мы можем зафиксировать мысли и чувства и передать их другому. Предложение языка описывает что-то. У нас будут два разных вида описаний. Одни говорят о чём-то конкретном, их мы будем называть *значениями*, а другие говорят о самих описаниях. Например это слова “числа”, “цвета” или “люди”. Есть конкретное число: один два или три, а есть все числа. Такие описания мы будем называть *типами*. Типы описывают множество значений. *Функции* описывают одни значения через другие. Это такие шаблоны описаний. Типичный пример функции, это “вычисление площади треугольника”. Функция как бы говорит: если ты мне покажешь треугольник, то я тебе скажу его площадь (число). Функция “вычисление площади треугольника” связывает два типа между собой: тип всех треугольников и тип чисел (значение площади). Могут быть и не математические функции. Например функция “цвет глаз” говорит нам: если ты покажешь мне человека, то я скажу какого цвета у него глаза. Эта функция связывает тип “люди” и тип “цвет”. При этом связь имеет направление. Функция сначала спрашивает у нас, чего ей не хватает, а потом говорит ответ. Ответ называют значением функции (или выходом функции), а то чего ей не хватает аргументами функции (или входами). Математики говорят, что эта функция отображает значения типа “люди” в значения типа “цвет”. В Haskell функции тоже являются значениями. Функция может принимать в качестве аргумента функцию и возвращать функцию.

Функции бывают *чистыми* и с *побочными эффектами*. Чистые функции – это правдивые функции. Их основная особенность в том, что для одинаковых ответов на их вопросы, они скажут одинаковые ответы. Функции с побочными эффектами так не делают, например если мы спросим у такой функции какого цвета глаза у Коли? В один день она может сказать голубые, а в другой зелёные. В Haskell таким функциям не доверяют и огораживают их от чистых функций, но я увлёкся, обо всём об этом вы узнаете из этой книги.

Благодарности

Я бы хотел поблагодарить родителей за терпение и поддержку, сообщество Haskell, всех тех людей, у которых я мог свободно учиться языку Haskell. Когда я только начинал мне очень помогли книга Мирана Липовача (Miran Lipovaca) *Learn You A Haskell for a Great Good* и книга Хал Дама (Hal Daume III) *Yet another Haskell Tutorial*. Спасибо Дмитрию Астапову, Дугласу Мак Илрою (Douglas McIlroy) и Джону Хьюзу (John Hughes) за великодушное согласие на использование примеров из их статей. Большое спасибо Кате Столяровой за идею написания книги. Спасибо Александру Мозгунову за расширение моего кругозора в Haskell и не только. Спасибо Оксане Станевич за поддержку, а также за редактирование первой главы.

Хочется поблагодарить тех, кто присылал правки, после появления первой версии книги. Огромное спасибо Андрею Мельникову. Его поддержка и замечания значительно углубили материал книги, вывели её на новый уровень. Книга сильно изменилась после комментариев Владимира Шабанова (появились части о сборщике мусора). Многие правки внесли Евгений Бахвалов, Сергей Дмитриев, Кирилл Заборский и Михаил Печкин. Также хотелось бы отметить тех, кто вносил правки через `github` и `ru_declarative`: `lionet`, `d_a0`, `odrdo`, `ul`.

Технические благодарности: команде GHC, за компилятор Haskell, Джону Мак Фарлану (John MacFarlane) за систему вёрстки `pandoc`, команде `TeXLive`, авторам `XeLaTeX`, автору пакета `hscolor` Малькольму Уолласу (Malcolm Wallace) за подсветку синтаксиса, авторам пакетов с `Hackage`: `diagrams` (Брент Йорги (Brent Yorgey), Райан Йэйтс (Ryan Yates)) `QuickCheck` (Козн Клаессен (Koen Claessen), Бьорн Брингерт (Bjorn Bringert), Ник Смолбоун (Nick Smallbone)), `criterion` (Брайан О'Салливан (Bryan O'Sullivan)), `HCodecs` (Джордж Гиоргадзе (George Giorgidze)), `fingertree` (Росс Патерсон (Ross Paterson), Ральф Хинце (Ralf Hinze)), `Hipmunk` (Фелипе Лесса (Felipe A. Lessa)), `OpenGL` (Джэйсон Даджит (Jason Dagit), Свен Пэнн (Sven Panne) и `GLFW` (Пол Лю (Paul H. Liu), Марк Санет (Marc Sunet)).

ОСНОВЫ

Есть мнение, что Haskell очень большой язык. Это и правда так. В Haskell много разных конструкций, синтаксического сахара, которые делают код более наглядным. Также в Haskell много библиотек на разные случаи жизни. Однако, обману ли я ваши ожидания, сказав, что всё это имеет достаточно компактную основу? Это и правда так, вам осталось лишь убедиться в наглядности и простоте Haskell. В этой главе мы пробежимся по нему, охватив одним взглядом целиком весь язык. Несколько наглядных конструкций, немного моих пояснений, и вы поймёте, что к чему. Если что-то сразу не станет ясно, или где-то я опущу какие-то пояснения, будьте уверены – в следующих главах мы обязательно обратимся к этим моментам и обсудим их подробнее.

Общая картина

Программы на Haskell бывают двух видов: это *приложения* (executable) и *библиотеки* (library). Приложения представляют собой исполняемые файлы, которые решают некоторую задачу, к примеру – это может быть компилятор языка, сортировщик данных в директориях, календарь, или цитатник на каждый день, любая полезная утилита. Библиотеки тоже решают задачи, но решают их внутри самого языка. Они содержат отдельные значения, функции, которые можно подключать к другой программе Haskell, и которыми можно пользоваться.

Программа состоит из *модулей* (module). И здесь работает правило: один модуль – один файл. Имя модуля совпадает с именем файла. Имя модуля начинается с большой буквы, тогда как файлы имеют расширение .hs. Например `FirstModule.hs`. Посмотрим на типичный модуль в Haskell:

```
-----  
-- шапка
```

```
module Имя(определение1, определение2,..., определениеN) where
```

```
import Модуль1(...)
```

```
import Модуль2(...)
```

```
...
```

```
-----  
-- определения
```

```
определение1
```

```
определение2
```

```
...
```

Каждый модуль содержит набор определений. Относительно модуля определения делятся на *экспортируемые* и *внутренние*.

Экспортируемые определения могут быть использованы за пределами модуля, а внутренние – только внутри модуля, и обычно они служат для выражения экспортируемых определений.

Модуль состоит из двух частей – шапки и определений.

Шапка

В шапке после слова **module** объявляется имя модуля, за которым в скобках следует список экспортируемых определений; после скобок стоит слово **where**. Затем идут импортируемые модули. С помощью импорта модулей вы имеете возможность в данном модуле пользоваться определениями из другого модуля.

Как после имени модуля, так и в директиве **import** скобки с определениями можно не писать, так как в этом случае считается, что экспортируются/импортируются все определения.

Определения

Эта часть содержит все определения модуля, при этом порядок следования определений не имеет значения. То есть, не обязательно пользоваться в данной функции лишь теми значениями, что были определены выше.

Модули взаимодействуют друг с другом с помощью экспортируемых определений. Один модуль может сказать, что он хочет воспользоваться экспортируемыми определениями другого модуля, для этого он пишет **import Модуль(определения)**. Модуль – это айсберг, на вершине которого – те функции, ради которых он создавался (экспортируемые), а под водой – все служебные детали реализации (внутренние).

Итак, программа состоит из модулей, модули состоят из определений. Но что такое определения?

В Haskell определения могут описывать четыре вида сущностей:

- Типы.
- Значения.
- Классы типов.
- Экземпляры классов типов.

Теперь давайте рассмотрим их подробнее.

Типы

Типы представляют собой каркас программы. Они кратко описывают все возможные значения. Это очень удобно. Опытный программист на Haskell может понять смысл функции по её названию и типу. Это не очень сложно. Например, мы видим:

```
not :: Bool -> Bool
```

Выражение `v :: T` означает, что значение `v` имеет тип `T`. Стрелка `a -> b` означает функцию, то есть из `a` мы можем получить `b`. Итак, перед нами функция из `Bool` в `Bool`, под названием `not`. Мы можем предположить, что это логическая операция “не”. Или, перед нами такое определение типа:

```
reverse :: [a] -> [a]
```

Мы видим функцию с именем `reverse`, которая принимает список `[a]` и возвращает список `[a]`, и мы можем догадаться, что эта функция переворачивает список, то есть мы получаем список, у которого элементы идут в обратном порядке. Маленькая буква `a` в `[a]` является параметром типа, на место параметра может быть поставлен любой тип. Она говорит о том, что список содержит элементы типа `a`. Например, такая функция соглашается переворачивать только списки логических значений:

```
reverseBool :: [Bool] -> [Bool]
```

Программа представляет собой описание некоторого явления или процесса. Типы определяют основные слова или термины и способы их комбинирования. А значения представляют собой комбинации базовых слов. Но значения комбинируются не произвольным образом, а на основе определённых правил, которые задаются типами.

Например, такое выражение определяет тип, в котором два базовых термина `True` или `False`

```
data Bool = True | False
```

Слово `data` ключевое, с него начинается любое определение нового типа. Символ `|` означает или. Наш новый тип `Bool` является либо словом `True`, либо словом `False`. В этом типе есть только понятия, но нет способов комбинирования, посмотрим на тип, в котором есть и то, и другое:

```
data [a] = [] | a : [a]
```

Это определение списка. Как мы уже поняли, `a` – это параметр. Список `[a]` может быть либо пустым списком `[]`, либо комбинацией `a : [a]`. В этой комбинации знак `:` объединяет элемент типа `a` и ещё один список `[a]`. Это рекурсивное определение, они встречаются в Haskell очень часто. Если это пока кажется непонятным, не пугайтесь, в следующих главах будет представлено много примеров с пояснениями.

Приведём ещё несколько примеров определений; ниже типы определяют базовые понятия для мира календаря: то что стоит за `--` является комментарием и игнорируется при выполнении программы:

```
-- Дата
```

```
data Date = Date Year Month Day
```

```
-- Год
```

```
data Year = Year Int           -- Int это целые числа
```

```
-- Месяц
```

```
data Month = January | February | March | April
            | May      | June     | July  | August
            | September | October  | November | December
```

```
data Day = Day Int
```

```
-- Неделя
data Week = Monday    | Tuesday    | Wednesday
           | Thursday  | Friday    | Saturday
           | Sunday

-- Время
data Time = Time Hour Minute Second

data Hour  = Hour  Int    -- Час
data Minute = Minute Int   -- Минута
data Second = Second Int  -- Секунда
```

Одной из основных целей разработчиков Haskell была ясность. Они стремились создать язык, предложения которого будут простыми и понятными, близкий к языку спецификаций.

С символом `|` мы уже познакомились, он указывает на альтернативы, объединение пишется через пробел. Так, фраза

```
data Time = Time Hour Minute Second
```

означает, что тип `Time` – это значение с меткой `Time`, которое состоит из значений типов “час”, “время” и “секунда”, и больше ничего. Метку принято называть *конструктором*.

Фраза

```
data Year = Year Int
```

означает, что тип `Year` – это значение с конструктором `Year`, которое состоит из одного значения типа `Int`. Конструктор обычно идёт первым, а за ним через пробел следуют другие типы. Конструктор может быть и самостоятельным значением, как в случае `True` или `January`.

Типы делят выполнение программы на две стадии: *компиляцию* (compile-time) и *вычисление* (run-time). На этапе компиляции происходит проверка типов. Программа, которая не прошла проверку типов, считается бессмысленной и не вычисляется. Приложение, которое выполняет компиляцию, называют *компилятором* (compiler), а то приложение, которое проводит вычисление, называют *вычислителем* (run-time system).

Типами мы определяем основные понятия в том явлении, которое мы хотим описать, а также осмысленные способы их комбинирования. Мы говорим, как из простейших терминов получаются составные. Если мы попытаемся построить бессмысленное предложение, компилятор языка автоматически найдёт такое предложение и сообщит нам об этом. Этот процесс заключается в проверке типов, к примеру если у нас есть функция сложения чисел, и мы попытаемся передать в неё строку или список, компилятор заметит это и скажет нам об этом *перед* тем как программа начнёт выполняться. И важно то, что это произойдёт очень быстро. Если мы случайно ошиблись в выражении, которое будет вычислено через час, нам не нужно ждать пока вычислитель дойдёт до ошибки, мы узнаем об этом, не успев моргнуть, после запуска программы.

Итак, если мы попробуем составить время из месяцев и логических значений:

Time January True 23

компилятор предупредит нас об ошибке. Наверное, вы думаете, что приведенный пример надуман, ведь кому захочется составлять время из логических значений? Но когда вы пишете программу, часто процесс работы складывается так: вы думаете над одним, пишете другое, а также планируете вернуться к третьему. И знание того, что есть надежный компилятор, который не пропустит глупых ошибок, освобождает руки, вы можете не заботиться о таких пустяках, как правильное построение предложения.

Отметим, что такой подход с разделением вычисления на две стадии и проверкой типов называется *статической типизацией*. Есть и другие языки, в них типы лишь подразумеваются и программа сразу начинает вычисляться, если есть какие-то несоответствия, об ошибке программисту сообщит вычислитель, причём только тогда, когда вычисление дойдёт до ошибки. Такой подход называют *динамической типизацией*.

Типы требуют серьёзных размышлений на начальном этапе, этапе определения базовых терминов и способов их комбинирования. Не упускаем ли мы что-то важное из виду, или, может быть, типы имеют

слишком общий характер и допускают ненужные нам предложения? Приходится задумываться. Но если типы подобраны удачно, они сами начинают подсказывать, как строить программу.

Значения

Итак, мы определили типами базовые понятия и способы комбинирования. Обычно это небольшой набор слов. Например в логических выражениях всего лишь два слова. Можем ли мы на что либо рассчитывать с таким словарным запасом? Оказывается, что да. Здесь на помощь приходят синонимы. Сейчас у нас в активе лишь два слова:

```
data Bool = True | False
```

И мы можем определить два синонима:

```
true :: Bool  
true = True
```

```
false :: Bool  
false = False
```

В Haskell синонимы пишутся с маленькой буквы. Синоним определяется через знак `=`. Обратите внимание на то, что это не процесс вычисления значения. Мы всего лишь объявляем новое имя для комбинации слов.

Теперь мы имеем целых четыре слова! Тем не менее, ушли мы не далеко, и два новых слова, в сущности, не делают язык выразительнее. Такие синонимы называют *константами*. Это значит, что одним словом мы будем обозначать некоторую комбинацию других слов. В данном случае комбинации очень простые.

Но наши синонимы могут определять одни слова через другие. Синонимы могут принимать параметры. Параметры пишутся через пробел между новым именем и знаком равно:

```
not :: Bool -> Bool  
not True  = False  
not False = True
```

Мы определили новое имя `not` с типом `Bool -> Bool`. Оно определяется двумя *уравнениями* (clause). Слева от знака равно левая часть уравнения, а справа – правая. В первом уравнении мы говорим, что сочетание (`not True`) означает `False`, а сочетание (`not False`) означает `True`. Опять же, мы ничего не вычисляем, мы даём новые имена нашим константам `True` и `False`. Только в этом случае имена составные.

Если вычислителю нужно узнать, что кроется за составным именем `not False` он *последовательно* проанализирует уравнения сверху вниз, до тех пор, пока левая часть уравнения не совпадёт со значением `not False`. Сначала мы сверим с первым:

```
not True  == not False    -- нет, пошли дальше
not False == not False    -- эврика, вернём правую часть
=> True
```

Определим ещё два составных имени

```
and :: Bool -> Bool -> Bool
and False _ = False
and True  x = x
```

```
or  :: Bool -> Bool -> Bool
or  True  _ = True
or  False x = x
```

Эти синонимы определяют логические операции “и” и “или”. Здесь несколько новых конструкций, но вы не пугайтесь, они не так трудны для понимания. Начнём с `_`:

```
and False _ = False
```

Здесь символ `_` означает, что в этом уравнении, если первый параметр равен `False`, то второй нам уже не важен, мы знаем ответ. Так, если в логическом “и” один из аргументов равен `False`, то всё выражение равно `False`. Так же и в случае с `or`.

Теперь другая новая конструкция:

```
and True  x = x
```

В этом случае параметр `x` служит для того, чтобы перетащить значение из аргумента в результат. Конкретное значение нам также

не важно, но в этом случае мы полагаем, что слева и справа от `=`, `x` имеет одно и то же значение.

Итак у нас уже целых семь имён: `True`, `False`, `true`, `false`, `not`, `and`, `or`. Или не семь? На самом деле, их уже бесконечное множество. Поскольку три из них составные, мы можем создавать самые разнообразные комбинации:

```
not (and true False)
or (and true true) (or False False)
not (not true)
not (or (or True True) (or False (not True)))
...
```

Обратите внимание на использование скобок, они группируют значения. Так, если бы мы написали `not not true` вместо `not (not true)`, мы бы получили ошибку компиляции, потому что `not` ожидает один параметр, а в выражении `not not true` их два. Параметры дописываются к имени через пробел.

Посмотрим, как происходят вычисления. В сущности, процесса вычислений нет, есть процесс замены синонимов на основные понятия согласно уравнениям. Базовые понятия мы определили в типах. Так давайте “вычислим” выражение `not (and true False)`:

-- выражение	-- уравнение
<code>not (and true False)</code>	-- <code>true</code> = <code>True</code>
<code>not (and True False)</code>	-- <code>and True x = x</code> => <code>and True False = False</code>
<code>not False</code>	-- <code>not False = True</code>
<code>True</code>	

Слева в столбик написаны шаги “вычисления”, а справа уравнения, по которым синонимы заменяются на комбинации слов. Процесс замены синонима (левой части уравнения) на комбинацию слов (правую часть уравнения) называется *редукцией* (reduction).

Сначала мы заменили синоним `true` на правую часть его уравнения, то есть на конструктор `True`. Затем мы заменили выражение `(and True False)` на правую часть из уравнения для синонима `and`. Обратите внимание на то, что переменная `x` была заменена на значение `False`. Последним шагом была замена синонима `not`. В конце

концов мы пришли к базовому понятию, а именно – к одному из двух конструкторов. В данном случае `True`.

Интересно, что новые синонимы могут быть использованы в правых частях уравнений. Так мы можем определить операцию “исключающее или”:

```
xor :: Bool -> Bool -> Bool
xor a b = or (and (not a) b) (and a (not b))
```

Этим выражением мы говорим, что `xor a b` это или отрицание `a` и `b`, или `a` и отрицание `b`. Это и есть определение “исключающего или”.

Может показаться, что с типом `Bool` мы зациклены на двух конструкторах, и единственное, что нам остаётся – это давать всё новые и новые имена словам `True` и `False`. Но на самом деле это не так. С помощью типов-параметров мы можем выйти за эти рамки.

Определим функцию ветвления `ifThenElse`:

```
ifThenElse :: Bool -> a -> a -> a
ifThenElse True  t  _ = t
ifThenElse False _  e = e
```

Эта функция первым аргументом принимает значение типа `Bool`, а вторым и третьим – альтернативы некоторого типа `a`. Если первый аргумент – `True`, возвращается второй аргумент, а если – `False`, то третий.

Интересно, что в Haskell ничего не происходит, мир Haskell-значений стоит на месте. Мы просто даём имена разным комбинациям слов. Определяем новые термины. Потом на этих терминах определяем новые термины, и так далее. Кажется, если ничего не меняется, то зачем язык? И что мы собираемся программировать без вычислений?

Разгадка кроется в функциях `not`, `and` и `or`. До того как мы их определили, у нас было четыре имени, но после их определения имён стало бесконечное множество. Три синонима дополнили наш язык бесконечным набором комбинаций. В этом суть. Мы определяем базовые элементы и способы составления новых, потом мы просим “вычислить” комбинацию из них. Мы не определяли явно, чему равна комбинация `not (and true False)`, это сделал за нас вычислитель Haskell¹.

Вычислить стоит в кавычках, потому что на самом деле вычислений нет, есть замена синонимов на комбинации простейших элементов.

Ещё один пример, положим у нас есть тип:

```
data Status = Work | Rest
```

Он определяет, что делать в данный день: работать (**Work**) или отдыхать (**Rest**). У разных рабочих разный график. Например, есть функции:

```
jonny :: Week -> Status  
jonny x = ...
```

```
colin :: Week -> Status  
colin x = ...
```

Конкретное определение сейчас не важно, важно, что они определяют зависимость статуса (**Status**) от дня недели (**Week**) для работников Джонни (jonny) и Колина (colin).

Также у нас есть полезная функция:

```
calendar :: Date -> Week  
calendar x = ...
```

Она определяет по дате день недели. И теперь, зная лишь эти функции, мы можем спросить у вычислителя будет ли у Джонни выходной 8 августа 3043 года:

```
jonny (calendar (Date (Year 3043) August (Day 8)))  
=> jonny Saturday  
=> Rest
```

Интересно, у нас опять всего лишь два значения, но, дав такое большое имя одному из значений, мы смогли получить полезную нам информацию, ничего не вычисляя.

Классы типов

Если типы и значения – привычные понятия, которые можно найти в том или ином виде в любом языке программирования, то термин класс типов встречается не часто. У него нет аналогов и в обычном

языке, поэтому я сначала постараюсь объяснить его смысл на примере.

В типизированном языке у каждой функции есть тип, но бывают функции, которые могут быть определены на аргументах разных типов; по сути, они описывают схожие понятия, но определены для значений разных типов. Например, функция сравнения на равенство, говорящая о том, что два значения одного типа `a` равны, имеет тип `a -> a -> Bool`, или функция печати выражения имеет тип `a -> String`, но что такое `a` в этих типах? Тип `a` является любым типом, для которого сравнение на равенство или печать (преобразование в строку) имеют смысл. Это понятие как раз и кодируется в классах типов. **Классы** типов (`type class`) позволяют определять функции с одинаковым именем для разных типов.

У классов типов есть имена. Также как и имена классов, они начинаются с большой буквы. Например, класс сравнений на равенство называется **Eq** (от англ. *equals* – равняется), а класс печати выражений имеет имя **Show** (от англ. *show* – показывать). Посмотрим на их определения:

Класс **Eq**:

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
```

Класс **Show**:

```
class Show a where
  show :: a -> String
```

За ключевым словом **class** следует имя класса, тип-параметр и ещё одно ключевое слово **where**. Далее с отступами пишутся имена определённых в классе значений. Значения класса называются *методами*.

Мы определяем лишь типы методов, конкретная реализация будет зависеть от типа `a`. Методы определяются в экземплярах классов типов, мы скоро к ним перейдём.

Программистская аналогия класса типов это интерфейс. В интерфейсе определён набор значений (как констант, так и функций), которые могут быть применены ко всем типам, которые поддерживают данный интерфейс. К примеру, в интерфейсе “сравнение на равенство” для некоторого типа `a` определены две функции: равно (`==`) и не равно (`/=`) с одинаковым типом `a -> a -> Bool`, или в интерфейсе “печати” для любого типа `a` определена одна функция `show` типа `a -> String`.

Математическая аналогия класса типов это алгебраическая система. Алгебра изучает свойства объекта в терминах операций, определённых на нём, и взаимных ограничениях этих операций. Алгебраическая система представляет собой набор операций и свойств этих операций. Этот подход позволяет абстрагироваться от конкретного представления объектов. Например группа – это все объекты данного типа `a`, для которых определены значения: константа – единица типа `a`, бинарная операция типа `a -> a -> a` и операция взятия обратного элемента, типа `a -> a`. При этом на операции накладываются ограничения, называемые свойствами операций. Например, ассоциативность бинарной операции, или тот факт, что единица с любым другим элементом, применённые к бинарной операции, дают на выходе исходный элемент.

Давайте определим класс для группы:

```
class Group a where
  e    :: a
  (+)  :: a -> a -> a
  inv  :: a -> a
```

Класс с именем `Group` имеет для некоторого типа `a` три метода: константу `e :: a`, операцию `(+) :: a -> a -> a` и операцию взятия обратного элемента `inv :: a -> a`.

Как и в алгебре, в Haskell классы типов позволяют описывать сущности в терминах определённых на них операций или значений. В примерах мы указываем лишь наличие операций и их типы, так же и в классах типов. Класс типов содержит набор имён его значений с информацией о типах значений.

Определив класс `Group`, мы можем начать строить различные выражения, которые будут потом интерпретироваться специфическим для типа образом:

```
twice :: Group a => a -> a
twice a = a + a

isE :: (Group a, Eq a) => a -> Bool
isE x = (x == e)
```

Обратите внимание на запись `Group a =>` и `(Group a, Eq a) =>`. Это называется контекстом объявления типа. В контексте мы говорим, что данный тип должен быть из класса `Group` или из классов `Group` и `Eq`. Это значит, что для этого типа мы можем пользоваться методами из этих классов.

В первой функции `twice` мы воспользовались методом `(+)` из класса `Group`, поэтому функция имеет контекст `Group a =>`. А во второй функции `isE` мы воспользовались методом `e` из класса `Group` и методом `(==)` из класса `Eq`, поэтому функция имеет контекст `(Group a, Eq a) =>`.

Контекст классов типов. Суперклассы

Класс типов также может содержать контекст. Он указывается между словом `class` и именем класса. Например

```
class IsPerson a

class IsPerson a => HasName a where
    name :: a -> String
```

Это определение говорит о том, что мы можем сделать экземпляр класса `HasName` только для тех типов, которые содержатся в `IsPerson`. Мы говорим, что класс `HasName` содержится в `IsPerson`. В этом случае класс из контекста `IsPerson` называют *суперклассом* для данного класса `HasName`.

Это сказывается на контексте объявления типа. Теперь, если мы пишем

```
fun :: HasName a => a -> a
```

Это означает, что мы можем пользоваться для значений типа `a` как методами из класса `HasName`, так и методами из класса `IsPerson`. Поскольку если тип принадлежит классу `HasName`, то он также принадлежит и `IsPerson`.

Запись `(IsPerson a => HasName a)` немного обманывает, было бы точнее писать `IsPerson a <= HasName a`, если тип `a` в классе `HasName`, то он точно в классе `IsPerson`, но в Haskell закрепилась другая запись.

Экземпляры классов типов

В *экземплярах* (`instance`) классов типов мы даём конкретное наполнение для методов класса типов. Определение экземпляра пишется так же, как и определение класса типа, но вместо `class` мы пишем `instance`, вместо некоторого типа наш конкретный тип, а вместо типов методов – уравнения для них.

Определим экземпляры для `Bool`

Класс `Eq`:

```
instance Eq Bool where
    (==) True  True  = True
    (==) False False = True
    (==) _     _     = False

    (/=) a b      = not (a == b)
```

Класс `Show`:

```
instance Show Bool where
    show True  = "True"
    show False = "False"
```

Класс `Group`:

```
instance Group Bool where
    e      = True
    (+) a b = and a b
    inv a   = not a
```

Отметим важность наличия свойств (ограничений) у значений, определённых в классе типов. Так, например, в классе типов “сравнение на равенство” для любых двух значений данного типа

одна из операций должна вернуть “истину”, а другая “ложь”, то есть два элемента данного типа либо равны, либо не равны. Недостаточно определить равенство для конкретного типа, необходимо убедиться в том, что для всех элементов данного типа свойства понятия равенства не нарушаются.

На самом деле приведённое выше определение экземпляра для `Group` не верно, хотя по типам оно подходит. Оно не верно как раз из-за нарушения свойств. Для группы необходимо, чтобы для любого `a` выполнялось:

```
inv a + a == e
```

У нас лишь два значения, и это свойство не выполняется ни для одного из них. Проверим:

```
inv True    + True
=> (not True) + True
=> False    + True
=> and False True
=> False
```

```
inv False   + False
=> (not False) + False
=> True       + False
=> and True   False
=> False
```

Проверять свойства очень важно, потому что другие люди, читая ваш код и используя ваши функции, будут на них рассчитывать.

Ядро Haskell

Фуууухх. Мы закончили наш пробег. Теперь можно остановиться, отдышаться и подвести итоги. Давайте вспомним синтаксические конструкции, которые нам встретились.

Модули

```
module New(edef1, edef2, ..., edefN) where
```

```
import Old1(idef11, idef12, ..., idef1N)
import Old2(idef21, idef22, ..., idef2M)
...
import OldK(idefK1, idefK2, ..., idefKP)
```

```
-- определения :  
...
```

Ключевые слова: `module`, `where`, `import`. Мы определили модуль с именем `New`, который экспортирует определения `edef1`, `edef2`, ... , `edefN`. И импортирует определения из модулей `Old1`, `Old2`, и т.д., определения написаны в скобках за ключевыми словами `import` и именами модулей.

Типы

Тип определяется с помощью:

- Перечисления альтернатив через `|`

```
data Type = Alt1 | Alt2 | ... | AltN
```

Эту операцию называют *суммой* типов.

- Составления сложного типа из подтипов, пишем конструктор первым, затем через пробел подтипы:

```
data Type = Name Sub1 Sub2 ... SubN
```

Эту операцию называют *произведением* типов.

Есть одно исключение: если тип состоит из двух подтипов, мы можем дать конструктору символьное (а не буквенное) имя, но оно должно начинаться с двоеточия `:`, как в случае списка, например, можно делать такие определения типов:

```
data Type = Sub1 :+ Sub2  
data Type = Sub1 :| Sub2
```

- Комбинации суммы и произведения типов:

```
data Type = Name1 Sub11 Sub12 ... Sub1N  
          | Name2 Sub21 Sub22 ... Sub2M  
          ...  
          | NameK SubK1 SubK2 ... SubKP
```

Такие типы называют *алгебраическими типами данных*. С помощью типов мы определяем основные понятия и способы их комбинирования.

Значения

Как это ни странно, нам встретилась лишь одна операция создания значений: *определение синонима*. Она пишется так

```
name x1 x2 ... xN = Expr1
name x1 x2 ... xN = Expr2
name x1 x2 ... xN = Expr3
```

Слева от знака равно стоит составное имя, а справа от знака равно некоторое выражение, построенное согласно типам. Разные комбинации имени name с параметрами определяют разные уравнения для синонима name.

Также мы видели символ `_`, который означает “всё, что угодно” на месте аргумента. А также мы увидели, как с помощью переменных можно перетаскивать значения из аргументов в результат.

Классы типов

Нам встретилась одна конструкция определения классов типов:

```
class Name a where
  method1 :: a -> ...
  method2 :: a -> ...
  ...
  methodN :: a -> ...
```

Экземпляры классов типов

Нам встретилась одна конструкция определения экземпляров классов типов:

```
instance Name Type where
  method1 x1 ... xN = ...
  method2 x1 ... xM = ...
  ...
  methodN x1 ... xP = ...
```

Типы, значения и классы типов

Каждое значение имеет тип. Значение `v` имеет тип `T` на Haskell:

```
v :: T
```

Функциональный тип обозначается стрелкой: `a -> b`

```
fun :: a -> b
```


Тип значения может иметь контекст, он говорит о том, что параметр должен принадлежать классу типов:

```
fun1 :: C a          => a -> a
fun2 :: (C1 a, C2, ..., CN) => a -> a
```

Суперклассы

Также контекст может быть и у классов, запись

```
class A a => B a where
  ...
```

Означает, что класс **B** целиком содержится в **A**, и перед тем как объявлять экземпляр для класса **B**, необходимо определить экземпляр для класса **A**. При этом класс **A** называют суперклассом для **B**.

Двумерный синтаксис

Наверное вы обратили внимание на то, что в Haskell нет разделителей строк и дополнительных скобок, которые бы указывали границы определения классов или функций. Компилятор Haskell ориентируется по переносам строки и отступам.

Так если мы пишем в классе:

```
class Eq a where
  (==) :: a -> a -> a
  (/=) :: a -> a -> a
```

По отступам за первой строкой определения компилятор понимает, что класс содержит два метода. Если бы мы написали:

```
class Eq a where
  (==) :: a -> a -> a
  (/=) :: a -> a -> a
```

То смысл был бы совсем другим. Теперь мы определяем класс **Eq** с одним методом **==** и указываем тип некоторого значения **(/=)**. Основное правило такое: конструкции, расположенные на одном уровне, выравниваются с помощью отступов. Чем правее находится определение, тем глубже оно вложено в какую-нибудь специальную конструкцию. Пока нам встретилось лишь несколько специальных конструкций, но дальше появятся и другие. Часто отступы

набираются с помощью табуляции. Это удобно. Но лучше пользоваться пробелами или настроить ваш любимый текстовый редактор так, чтобы он автоматически заменял табуляцию на пробелы. Зачем это нужно? Дело в том, что в разных редакторах на табуляцию может быть назначено разное количество пробелов, так код набранный с двухзначной табуляцией будет очень трудно прочитать если открыть его в редакторе с четырьмя пробелами вместо табуляции. Поскольку очень часто табуляция перемежается с пробелами и выравнивание может “поехать”. Поэтому признаком хорошего стиля в Haskell считается полный отказ от табуляции.

Краткое содержание

Итак подведём итоги: у нас есть две операции для определения типов (сумма и произведение) и по одной для значений (синонимы), классов типов и экземпляров. А также бесконечное множество их комбинаций, из которых и состоит увлекательный мир Haskell. Конечно не только из них, есть нюансы, синтаксический сахар, расширения языка. Об этом и многом другом мы узнаем из этой книги.

Интересно, что в Haskell, несмотря на обилие конструкций и библиотек, ты чувствуешь, что за ними стоит нечто из мира науки, мира чистого знания. Ты не просто учишься пользоваться определёнными функциями или классами, а узнаёшь что-то новое и красивое.

Упражнения

Потренируйтесь в описаниях в рамках системы типов. Вы определяете базовые понятия и способы их комбинирования. У вас есть три операции:

- Сумма типов `data T = A1 | A2`. Перечисление альтернатив
- Произведение типов `data T = S S1 S2`. Этим мы говорим, что понятие состоит из нескольких.
- Взятие в список `[T]`. Обозначает множественное число, элементов типа `T` их может быть несколько.

Опишите что-либо: комнату, дорогу, город, человека, главу из книги, математическую теорию, всё что угодно.

Ниже приведён пример для понятий из этой главы:

```
data Program = Programm ProgramType [Module]
data ProgramType = Executable | Library

data Module = Module [Definition]

data Definition = Definition DefinitionType Element
data DefinitionType = Export | Inner

data Element = ET Type | EV Value | EC Class | EI Instance

data Type      = Type String
data Value     = Value String
data Class     = Class String
data Instance  = Instance String
```

После того как вы закончите с описанием, подумайте, какие производные связи могли бы вас заинтересовать. Какие функции вам бы хотелось определить в этом описании. Выпишите их типы без определений, например так:

```
-- Все объявления типов в модуле
getTypes :: Module -> [Type]

-- Провести редукцию значения:
reduce :: Value -> Program -> Value

-- Проверить типы:
checkTypes :: Program -> Bool

-- Заменить все определения в модуле на новые
setDefinitions :: Module -> [Definition] -> Module

-- Упорядочить определения по какому-либо принципу
orderDefinitions :: [Definition] -> [Definition]
```

Подумайте: если у вас есть все эти функции, какие производные значения могли бы вам сказать что-нибудь интересное.

Первая программа

Я вот говорю-говорю, а вдруг я вас обманываю, и ничего этого нет. В этой главе мы перейдём к программированию и запустим нашу первую программу в Haskell. Будет много примеров, на которых мы закрепим наши знания.

Интерпретатор

Для запуска кода мы будем пользоваться приложением GHC (Glorious Glasgow Haskell Compiler) наиболее развитой системой интерпретации Haskell программ. В GHC есть компилятор `ghc` и интерпретатор `ghci`. Пока мы будем пользоваться лишь интерпретатором. Если вы не знаете как установить `ghc` загляните в приложение. Также нам понадобится текстовый редактор с подсветкой синтаксиса. Подсветка синтаксиса для Haskell по умолчанию есть в редакторах Vim, Emacs, gedit, geany, yi. Есть IDE для Haskell Leksah. Мы будем писать модули в файлах и загружать их в интерпретатор. Если вы не знаете продвинутых текстовых редакторов вроде Vim или Emacs, лучше всего будет начать с gedit.

Интерпретатор позволяет загружать модуль с определениями и набирать значения в командной строке. Мы набираем значение, а интерпретатор редуцирует его и показывает нам ответ. Интерпретатор запускается командой `ghci` в терминале. Определения из модуля могут быть загружены в интерпретатор двумя способами, либо при запуске интерпретатора командой `ghci ИмяМодуля.hs` либо в самом интерпретаторе командой `:l ИмяМодуля.hs`.

Рассмотрим некоторые полезные команды интерпретатора:

`:?`

Выводит на экран список доступных команд

`:t Expression`

Возвращает тип выражения.

`:set +t`

После выполнения команды интерпретатор будет выводить на экран не только результат вычисления выражения, но и его тип.

`:set +s`

После выполнения команды интерпретатор будет выводить на экран не только результат вычисления выражения, но и статистику вычислений.

`:l ИмяМодуля`

Загружает модуль в интерпретатор.

`:cd Директория`

Перейти в данную директорию.

`:r`

Перезагружает, последний загруженный модуль. Этой командой можно пользоваться после внесения в модуль изменений.

`:q`

Выход из интерпретатора.

У-вей

Согласно даосам основной принцип жизни заключается в недеянии (у-вей). Всё происходит естественно и словно само собой. Давайте создадим модуль который ничего не делает. Создадим пустой модуль и загрузим его в интерпретатор.

```
module Empty where
```

```
import Prelude()
```

Зачем мы написали `import Prelude()`? Этой фразой мы говорим, что не хотим ничего импортировать из модуля `Prelude`. По умолчанию в любой модуль загружается модуль `Prelude`, который содержит много

полезных определений. К примеру там определяется тип `Bool`, списки и функции для них, символы, классы типов для сравнения на равенство и печати значений и много, много других определений. В первых главах я хочу сделать акцент на самом языке Haskell, а не на производных выражениях, поэтому пока мы будем в явном виде загружать из модуля `Prelude` лишь самые необходимые определения.

Сохраним модуль в файле `Empty.hs`, сделаем директорию модуля текущей и запустим интерпретатор командой `ghci Empty` (имя расширения можно не писать). Также можно просто запустить интерпретатор командой `ghci`, переключиться на директорию командой `:cd` и загрузить модуль командой `:l Empty`.

```
$ ghci
GHCi, version 7.4.1: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude> :cd ~/haskell-notes/code/ch-2/
Prelude> :l Empty.hs
[1 of 1] Compiling Empty                ( Empty.hs, interpreted )
Ok, modules loaded: Empty.
*Empty>
```

Слева от знака приглашения к вводу `>` отображаются загруженные в интерпретатор модули. По умолчанию загружается модуль `Prelude`. После выполнения команды `:l` мы видим, что `Prelude` сменилось на `Empty`.

Теперь давайте потренируемся перезагружать модули. Давайте изменим наш модуль, сделаем его не таким пустым, убрав последние две скобки от модуля `Prelude` в директиве `import`. Теперь сохраним изменения и выполним команду `:r`.

```
*Empty> :r
[1 of 1] Compiling Empty                ( Empty.hs, interpreted )
Ok, modules loaded: Empty.
*Empty>
```

Завершим сессию интерпретатора командой `:q`.

```
*Empty> :q
Leaving GHCi.
```

Внешние модули должны находиться в текущей директории. Давайте потренируемся с подключением определений из внешних модулей. Создадим модуль близнец модуля `Empty.hs`:

```
module EmptyEmpty where

import Prelude()
```

И сохраним его в той же директории, что и модуль `Empty`, теперь мы можем включить все определения из модуля `EmptyEmpty`:

```
module Empty where

import EmptyEmpty
```

Когда у нас будет много модулей мы можем разместить их по директориям. Создадим в одной директории с модулем `Empty` директорию `Sub`, а в неё поместим копию модуля `Empty`. Существует одна тонкость: поскольку модуль находится в поддиректории, для того чтобы он стал виден из текущей директории, необходимо дописать через точку имя директории в которой он находится:

```
module Sub.Empty where
```

Теперь мы можем загрузить этот модуль из исходного:

```
module Empty where

import EmptyEmpty
import Sub.Empty
```

Обратите внимание на то, что мы приписываем к модулю в поддиректории `Sub` имя поддиректории. Если бы он был заложен в ещё одной директории, то мы написали бы через точку имя и этой поддиректории:

```
module Empty where

import Sub1.Sub2.Sub3.Sub4.Empty
```

Логические значения

Пустой модуль это хорошо, но слишком скучно. Давайте перепишем объявленные в этой главе определения в модуль, загрузим его в интерпретатор и понаблюдаем значения.

Начнём с логических операций. Давайте не будем переопределять `Bool`, `Show` и `Eq`, а просто возьмём их из `Prelude`:

```
module Logic where

import Prelude (Bool(..), Show(..), Eq(..))
```

Две точки в скобках означают “все конструкторы” (в случае типа) и “все методы” (в случае класса типа). Строчку

```
import Prelude (Bool(..), Show(..), Eq(..))
```

Следует читать так: Импортируй из модуля `Prelude` тип `Bool` и все его конструкторы и классы `Show` и `Eq` со всеми их методами. Если бы мы захотели импортировать только конструктор `True`, мы бы написали `Bool(True)`, а если бы мы захотели импортировать лишь имя типа, мы бы написали просто `Bool` без скобок.

Сначала выпишем в модуль наши синонимы:

```
module Logic where

import Prelude (Bool(..), Show(..), Eq(..))
```

```
true :: Bool
true = True
```

```
false :: Bool
false = False
```

```
not :: Bool -> Bool
not True  = False
not False = True
```

```
and :: Bool -> Bool -> Bool
and False _ = False
and True  x = x
```

```
or  :: Bool -> Bool -> Bool
or True  _ = True
or False x = x
```



```
xor :: Bool -> Bool -> Bool
xor a b = or (and (not a) b) (and a (not b))
```

```
ifThenElse :: Bool -> a -> a -> a
ifThenElse True  t  _ = t
ifThenElse False _  e = e
```

Теперь сохраним модуль и загрузим его в интерпретатор. Для наглядности мы установим флаг `+t`, при этом будет возвращено не только значение, но и его тип. Понабираем разные комбинации значений:

```
*Logic> :l Logic
[1 of 1] Compiling Logic           ( Logic.hs, interpreted )
Ok, modules loaded: Logic.
*Logic> :set +t
*Logic> not (and true False)
True
it :: Bool
*Logic> or (and true true) (or False False)
True
it :: Bool
*Logic> xor (not True) (False)
False
it :: Bool
*Logic> ifThenElse (or true false) True False
True
it :: Bool
```

Разумеется в Haskell уже определены логические операции, здесь мы просто тренировались. Они называются `not`, `(&&)`, `||`. Операция `xor` это то же самое, что и `(/=)`. Для `Bool` определён экземпляр класса `Eq`. Также в Haskell есть конструкция ветвления она пишется так:

```
x = if cond then t else e
```

Слова `if`, `then` и `else` – ключевые. `cond` имеет тип `Bool`, а `t` и `e` одинаковый тип.

В коде программы обычно пишут так:

```
x = if a > 3
    then "Hello"
    else (if a < 0
          then "Hello"
          else "Bye")
```

Отступы обязательны.

Давайте загрузим в интерпретатор модуль `Prelude` и наберём те же выражения стандартными функциями:

```
*Logic> :m Prelude
Prelude> not (True && False)
True
it :: Bool
Prelude> (True && True) || (False || False)
True
it :: Bool
Prelude> not True /= False
False
it :: Bool
Prelude> if (True || False) then True else False
True
it :: Bool
```

Бинарные операции с символьными именами пишутся в инфиксной форме, то есть между аргументами как в `a && b` или `a + b`. Значение с буквенным именем также можно писать в инфиксной форме, для этого оно заключается в апострофы, например `a `and` b` или `a `plus` b`. Апострофы обычно находятся на одной кнопке с буквой “ё”. Также символьные функции можно применять в префиксной форме, заключив их в скобки, например `(&&) a b` и `(+) a b`. Попробуем в интерпретаторе:

```
Prelude> True && False
False
it :: Integer
Prelude> (&&) True False
False
it :: Bool
Prelude> let and a b = a && b
and :: Bool -> Bool -> Bool
Prelude> and True False
False
it :: Bool
Prelude> True `and` False
False
it :: Bool
```

Обратите внимание на строчку `let and a b = a && b`. В ней мы определили синоним в интерпретаторе. Сначала мы пишем ключевое слово `let` затем обычное определение синонима, как в программе. Это простое однострочное определение, но мы можем набирать в

интерпретаторе и более сложные. Мы можем написать несколько строчек в одной, разделив их точкой с запятой:

```
Prelude> let not2 True = False; not2 False = True
```

Мы можем записать это определение более наглядно, совсем как в редакторе, если воспользуемся многострочным вводом. Для этого просто наберите команду `:{`. Для выхода воспользуйтесь командой `:}`. Отметим, что точкой с запятой можно пользоваться и в обычном коде. Например в том случае если у нас много кратких определений и мы хотим записать их покомпактней, мы можем сделать это так:

```
a1 = 1;   a2 = 2;   a3 = 3
a4 = 4;   a5 = 5;   a6 = 6
```

Класс Show. Строки и символы

Мы набираем в интерпретаторе какое-нибудь сложное выражение, или составной синоним, интерпретатор проводит редукцию и выводит ответ на экран. Откуда интерпретатор знает как отображать значения типа `Bool`? Внутри интерпретатора вызывается метод класса `Show`, который переводит значение в строку. И затем мы видим на экране ответ.

Для типа `Bool` экземпляра класса `Show` уже определён, поэтому интерпретатор знает как его отображать.

Обратите внимание на эту особенность языка, вид значения определяется пользователем, в экземпляре класса `Show`. Из соображений наглядности вид значения может сильно отличаться от его внутреннего представления.

В этом разделе мы рассмотрим несколько примеров с классом `Show`, но перед этим мы поговорим о строках и символах в языке Haskell.

Строки и символы

Посмотрим в интерпретаторе на определение строк (тип `String`), для этого мы воспользуемся командой `:i` (сокращение от `:info`):

```
Prelude> :i String
type String = [Char]      -- Defined in `GHC.Base'
```

Интерпретатор показал определение типа и в комментариях указал в каком модуле тип определён. В этом определении мы видим новое ключевое слово `type`. До этого для определения типов нам встречалось лишь слово `data`. Ключевое слово `type` определяет синоним типа. При этом мы не вводим новый тип, мы лишь определяем для него псевдоним. `String` является синонимом для списка значений типа `Char`. Тип `Char` представляет символы. Итак строка – это список символов. В Haskell символы пишутся в одинарных кавычках, а строки в двойных:

```
Prelude> ['H','e','l','l','o']
"Hello"
it :: [Char]
Prelude> "Hello"
"Hello"
it :: [Char]
Prelude> '+'
'+'
it :: Char
```

Для обозначения перехода на новую строку используется специальный символ `\n`. Если строка слишком длинная и не помещается на одной строке, то её можно перенести так:

```
str = "My long long long long \
      \long long string"
```

Перенос осуществляется с помощью комбинации следующих друг за другом обратных слэшей.

Нам понадобится функция конкатенации списков (`++`), она определена в `Prelude`, с её помощью мы будем объединять строки:

```
Prelude> :t (++)
(++) :: [a] -> [a] -> [a]
Prelude> "Hello" ++ [' ' ] ++ "World"
"Hello World"
it :: [Char]
```

Пример: Отображение дат и времени

Приведём, пример в котором отображаемое значение не совпадает с видом значения в коде. Мы отобразим значения из мира календаря. Для начала давайте сохраним определения в отдельном модуле:

```

module Calendar where

import Prelude (Int, Char, String, Show(..), (++))

-- Дата
data Date = Date Year Month Day

-- Год
data Year = Year Int      -- Int это целые числа

-- Месяц
data Month = January      | February      | March        | April
            | May          | June         | July          | August
            | September    | October     | November     | December

data Day = Day Int

-- Неделя
data Week = Monday        | Tuesday       | Wednesday
            | Thursday     | Friday      | Saturday
            | Sunday

-- Время
data Time = Time Hour Minute Second

data Hour = Hour Int      -- Час
data Minute = Minute Int  -- Минута
data Second = Second Int  -- Секунда

```

Теперь сохраним наш модуль под именем `Calendar.hs` и загрузим в интерпретатор:

```

Prelude> :l Calendar
[1 of 1] Compiling Calendar          ( Calendar.hs, interpreted )
Ok, modules loaded: Calendar.
*Calendar> Monday

<interactive>:3:1:
  No instance for (Show Week)
    arising from a use of `System.IO.print'
  Possible fix: add an instance declaration for (Show Week)
  In a stmt of an interactive GHCi command: System.IO.print it

```

Смотрите мы попытались распечатать значение `Monday`, но в ответ получили ошибку. В ней интерпретатор сообщает нам о том, что для типа `Week` не определён экземпляр класса `Show`, и он не знает как его распечатывать. Давайте подскажем ему. Обычно дни недели в календарях печатают не полностью, в имя попадают лишь три первых буквы:

```
instance Show Week where
  show Monday    = "Mon"
  show Tuesday   = "Tue"
  show Wednesday = "Wed"
  show Thursday  = "Thu"
  show Friday    = "Fri"
  show Saturday  = "Sat"
  show Sunday    = "Sun"
```

Отступы перед `show` обязательны, но выравнивание по знаку равно не обязательно, мне просто нравится так писать. По отступам компилятор понимает, что все определения относятся к определению `instance`. Теперь запишем экземпляр в модуль, сохраним, и перезагрузим в интерпретатор:

```
*Calendar> :r
[1 of 1] Compiling Calendar          ( Calendar.hs, interpreted )
Ok, modules loaded: Calendar.
*Calendar> Monday
Mon
it :: Week
*Calendar> Sunday
Sun
it :: Week
```

Теперь наши дни отображаются. Я выпишу ещё один пример экземпляра для `Time`, а остальные достанутся вам в качестве упражнения.

```
instance Show Time where
  show (Time h m s) = show h ++ ":" ++ show m ++ ":" ++ show s

instance Show Hour where
  show (Hour h) = addZero (show h)

instance Show Minute where
  show (Minute m) = addZero (show m)

instance Show Second where
  show (Second s) = addZero (show s)

addZero :: String -> String
addZero (a:[]) = '0' : a : []
addZero as    = as
```

Функцией `addZero` мы добавляем ноль в начало строки, в том случае, если число однозначное, также в этом определении мы воспользовались тем, что для типа целых чисел `Int` экземпляр `Show` уже определён. Проверим в интерпретаторе:

```
*Calendar> Time (Hour 13) (Minute 25) (Second 2)
13:25:02
it :: Time
```

Автоматический вывод экземпляров классов типов

Для некоторых стандартных классов экземпляры классов типов могут быть выведены автоматически. Это делается с помощью директивы **deriving**. Она пишется сразу после объявления типа. Например так мы можем определить тип и экземпляры для классов **Show** и **Eq**:

```
data T = A | B | C
    deriving (Show, Eq)
```

Отступ за **deriving** обязателен, после ключевого слова в скобках указываются классы, которые мы хотим вывести.

Арифметика

В этом разделе мы обсудим основные арифметические операции. В Haskell много стандартных классов, которые группируют различные типы операций, есть класс для сравнения на равенство, отдельный класс для сравнения на больше/меньше, класс для умножения, класс для деления, класс для упорядоченных чисел, и много других. Зачем такое изобилие классов?

Каждый из классов отвечает независимой группе операций. Есть много объектов, которые можно только складывать, но нельзя умножать или делить. Есть объекты, для которых сравнение на равенство имеет смысл, а сравнение на больше/меньше – нет.

Для иллюстрации мы воспользуемся числами Пеано, у них компактное определение, всего два конструктора, которых тем не менее достаточно для описания множества натуральных чисел:

```
module Nat where
```

```
data Nat = Zero | Succ Nat
    deriving (Show, Eq, Ord)
```

Конструктор `Zero` указывает на число ноль, а `(Succ n)` на число следующее за данным числом `n`. В последней строке мы видим новый класс `Ord`, этот класс содержит операции сравнения на больше/меньше:

```
Prelude> :i Ord
class (Eq a) => Ord a where
  compare :: a -> a -> Ordering
  (<)    :: a -> a -> Bool
  (>=)   :: a -> a -> Bool
  (>)    :: a -> a -> Bool
  (<=)   :: a -> a -> Bool
  max    :: a -> a -> a
  min    :: a -> a -> a
```

Тип `Ordering` кодирует результаты сравнения:

```
Prelude> :i Ordering
data Ordering = LT | EQ | GT    -- Defined in GHC.Ordering
```

Он содержит конструкторы, соответствующие таким понятиям как меньше, равно и больше.

Класс `Eq`. Сравнение на равенство

Вспомним определение класса `Eq`:

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool

  a == b = not (a /= b)
  a /= b = not (a == b)
```

Появились две детали, о которых я умолчал в предыдущей главе. Это две последние строки. В них мы видим определение `==` через `/=` и наоборот. Это определения методов по умолчанию. Такие определения дают нам возможность определять не все методы класса, а лишь часть основных, а все остальные мы получим автоматически из определений по умолчанию.

Казалось бы почему не оставить в классе `Eq` один метод а другой метод определить в виде отдельной функции:


```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: Eq a => a -> a -> Bool
  a /= b = not (a == b)
```

Так не делают по соображениям эффективности. Есть типы для которых проще вычислить `/=` чем `==`. Тогда мы определим тот метод, который нам проще вычислять и второй получим автоматически.

Набор основных методов, через которые определены все остальные называют *минимальным полным определением* (minimal complete definition) класса. В случае класса `Eq` это метод `==` или метод `/=`.

Мы уже вывели экземпляр для `Eq`, поэтому мы можем пользоваться методами `==` и `/=` для значений типа `Nat`:

```
*Calendar> :l Nat
[1 of 1] Compiling Nat                ( Nat.hs, interpreted )
Ok, modules loaded: Nat.
*Nat> Zero == Succ (Succ Zero)
False
it :: Bool
*Nat> Zero /= Succ (Succ Zero)
True
it :: Bool
```

Класс Num. Сложение и умножение

Сложение и умножение определены в классе `Num`. Посмотрим на его определение:

```
*Nat> :i Num
class (Eq a, Show a) => Num a where
  (+) :: a -> a -> a
  (*) :: a -> a -> a
  (-) :: a -> a -> a
  negate :: a -> a
  abs :: a -> a
  signum :: a -> a
  fromInteger :: Integer -> a
  -- Defined in GHC.Num
```

Методы `(+)`, `(*)`, `(-)` в представлении не нуждаются, метод `negate` является унарным минусом, его можно определить через `(-)` так:

```
negate x = 0 - x
```

Метод `abs` является модулем числа, а метод `signum` возвращает знак числа, метод `fromInteger` позволяет создавать значения данного типа из стандартных целых чисел `Integer`.

Этот класс устарел, было бы лучше сделать отдельный класс для сложения и вычитания и отдельный класс для умножения. Также контекст класса, часто становится помехой. Есть объекты, которые нет смысла печатать но, есть смысл определить на них сложение и умножение. Но пока в целях совместимости с уже написанным кодом, класс `Num` остаётся прежним.

Определим экземпляр для чисел Пеано, но давайте сначала разберём функции по частям.

Сложение

Начнём со сложения:

```
instance Num Nat where
  (+) a Zero      = a
  (+) a (Succ b) = Succ (a + b)
```

Первое уравнение говорит о том, что, если второй аргумент равен нулю, то мы вернём первый аргумент в качестве результата. Во втором уравнении мы “перекидываем” конструктор `Succ` из второго аргумента за пределы суммы. Схематически вычисление суммы можно представить так:

$$3 + 2 \rightarrow 1 + (3 + 1) \rightarrow 1 + (1 + (3 + 0))$$
$$1 + (1 + (1 + (1 + (1 + 0)))) \rightarrow 5$$

Все наши числа имеют вид 0 или $1 + n$, мы принимаем на вход два числа в таком виде и хотим в результате составить число в этом же виде, для этого мы последовательно перекидываем $(1 +)$ в начало выражения из второго аргумента.

Вычитание

Операция отрицания не имеет смысла, поэтому мы воспользуемся специальной функцией `error :: String -> a`, она принимает строку с

сообщением об ошибке, при её вычислении программа остановится с ошибкой и сообщение будет выведено на экран.

```
negate _ = error "negate is undefined for Nat"
```

Умножение

Теперь посмотрим на умножение:

```
(*) a Zero      = Zero
(*) a (Succ b) = a + (a * b)
```

В первом уравнении мы вернём ноль, если второй аргумент окажется нулём, а во втором мы за каждый конструктор `Succ` во втором аргументе прибавляем к результату первый аргумент. В итоге, после вычисления `a * b` мы получим аргумент `a` сложенный `b` раз. Это и есть умножение. При этом мы воспользовались операцией сложения, которую только что определили. Посмотрим на схему вычисления:

```
$$\textbf{3*2} \rightarrow \textbf{3+} \textbf{(3*1)} \rightarrow
3+(3+\textbf{(3*0)}) \rightarrow 3+\textbf{(3+0)} \rightarrow
\textbf{3+3} \rightarrow$$
```

```
$$1+\textbf{(3+2)} \rightarrow 1+(1+\textbf{(3+1)}) \rightarrow
1+(1+(1+\textbf{(3+0)})) \rightarrow$$
```

```
$$1+(1+1+\textbf{3}) \rightarrow 1+(1+(1+(1+(1+(1+0)))))
\rightarrow 6$$
```

Операции `abs` и `signum`

Поскольку числа у нас положительные, то методы `abs` и `signum` почти ничего не делают:

```
abs    x    = x
signum Zero = Zero
signum _    = Succ Zero
```

Перегрузка чисел

Остался последний метод `fromInteger`. Он конструирует значение нашего типа из стандартного:

```
fromInteger 0 = Zero
fromInteger n = Succ (fromInteger (n-1))
```

Зачем он нужен? Попробуйте узнать тип числа `1` в интерпретаторе:

```
*Nat> :t 1
1 :: (Num t) => t
```

Интерпретатор говорит о том, тип значения `1` является некоторым типом из класса `Num`. В Haskell обозначения для чисел перегружены. Когда мы пишем `1` на самом деле мы пишем `(fromInteger (1::Integer))`. Поэтому теперь мы можем не писать цепочку `Succ`-ов, а воспользоваться методом `fromInteger`, для этого сохраним определение экземпляра для `Num` и загрузим обновлённый модуль в интерпретатор:

```
[1 of 1] Compiling Nat                ( Nat.hs, interpreted )
Ok, modules loaded: Nat.
*Nat> 7 :: Nat
Succ (Succ (Succ (Succ (Succ (Succ (Succ Zero)))))
*Nat> (2 + 2) :: Nat
Succ (Succ (Succ (Succ Zero)))
*Nat> 2 * 3 :: Nat
Succ (Succ (Succ (Succ (Succ (Succ Zero)))))
```

Вы можете убедиться насколько гибкими являются числа в Haskell:

```
*Nat> (1 + 1) :: Nat
Succ (Succ Zero)
*Nat> (1 + 1) :: Double
2.0
*Nat> 1 + 1
2
```

Мы выписали три одинаковых выражения и получили три разных результата, меняя объявление типов. В последнем выражении тип был приведён к `Integer`. Это поведение интерпретатора по умолчанию. Если мы напишем:

```
*Nat> let q = 1 + 1
*Nat> :t q
q :: Integer
```

Мы видим, что значение `q` было переведено в `Integer`, это происходит лишь в интерпретаторе, если такая переменная встретится в программе и компилятор не сможет определить её тип из контекста, произойдёт ошибка проверки типов, компилятор

скажет, что он не смог определить тип. Помочь компилятору можно, добавив объявление типа с помощью конструкции `(v :: T)`.

Посмотрим ещё раз на определение экземпляра `Num` для `Nat` целиком:

```
instance Num Nat where
  (+) a Zero      = a
  (+) a (Succ b) = Succ (a + b)

  (*) a Zero      = Zero
  (*) a (Succ b) = a + (a * b)

  fromInteger 0 = Zero
  fromInteger n = Succ (fromInteger (n-1))

  abs    x      = x
  signum Zero = Zero
  signum _     = Succ Zero

  negate _ = error "negate is undefined for Nat"
```

Класс Fractional. Деление

Деление определено в классе `Fractional`:

```
*Nat>:m Prelude
Prelude> :i Fractional
class Num a => Fractional a where
  (/) :: a -> a -> a
  recip :: a -> a
  fromRational :: Rational -> a
  -- Defined in `GHC.Real'
instance Fractional Float -- Defined in `GHC.Float'
instance Fractional Double -- Defined in `GHC.Float'
```

Функция `recip`, это аналог `negate` для `Num`. Она делит единицу на данное число. Функция `fromRational` строит число данного типа из дробного числа. Если мы пишем `2`, то к нему подспудно будет применена функция `fromInteger`, а если `2.0`, то будет применена функция `fromRational`.

Стандартные числа

В этом подразделе мы рассмотрим несколько стандартных типов для чисел в Haskell. Все эти числа являются экземплярами основных численных классов. Тех, которые мы рассмотрели, и многих-многих других.

Целые числа

В Haskell предусмотрено два типа для целых чисел. Это `Integer` и `Int`. Чем они отличаются? Значения типа `Integer` не ограничены, мы можем проводить вычисления с очень-очень-очень большими числами, если памяти на нашем компьютере хватит. Числа из типа `Int` ограничены. Каждое число занимает определённый размер в памяти компьютера. Диапазон значений для `Int` составляет от -2^{29} до $2^{29}-1$. Вычисления с `Int` более эффективны.

Действительные числа

Действительные числа бывают дробными (тип `Rational`), с одинарной точностью `Float` и с двойной точностью `Double`. Числа из типа `Float` занимают меньше места, но они не такие точные как `Double`. Если вы сомневаетесь, чем пользоваться, выбирайте `Double`, обычно `Float` используется только там, где необходимо хранить огромные массивы чисел. В этом случае мы экономим много памяти.

Преобразование численных типов

Во многих языках программирования при сложении или умножении чисел разных типов проводится автоматическое приведение типов. Обычно целые числа становятся действительными, `Float` превращается в `Double` и так далее. Это противоречит строгой типизации, поэтому в Haskell этого нет:

```
Prelude> (1::Int) + (1::Double)
```

```
<interactive>:2:13:
```

```
Couldn't match expected type `Int' with actual type `Double'
In the second argument of `(+)', namely `(1 :: Double)'
In the expression: (1 :: Int) + (1 :: Double)
In an equation for `it': it = (1 :: Int) + (1 :: Double)
```

Любое преобразование типов контролируется пользователем. Мы должны вызвать специальную функцию.

От целых к действительным: Часто возникает необходимость приведения целых чисел к действительным при делении. Для этого можно воспользоваться функцией: `fromIntegral`

```
Prelude> :i fromIntegral
fromIntegral :: (Integral a, Num b) => a -> b
```

```
-- Defined in `GHC.Real'
```

Определим функцию поиска среднего между двумя целыми числами:

```
meanInt :: Int -> Int -> Double
meanInt a b = fromIntegral (a + b) / 2
```

В этой функции двойка имеет тип `Double`. Обратите внимание на скобки: составной синоним всегда притягивает аргументы сильнее чем бинарная операция.

От действительных к целым: В этом нам поможет класс `RealFrac`. Методы говорят сами за себя:

```
Prelude GHC.Float> :i RealFrac
class (Real a, Fractional a) => RealFrac a where
  properFraction :: Integral b => a -> (b, a)
  truncate :: Integral b => a -> b
  round :: Integral b => a -> b
  ceiling :: Integral b => a -> b
  floor :: Integral b => a -> b
  -- Defined in `GHC.Real'
instance RealFrac Float -- Defined in `GHC.Float'
instance RealFrac Double -- Defined in `GHC.Float'
```

Метод `properFraction` отделяет целую часть числа от дробной:

```
properFraction :: Integral b => a -> (b, a)
```

Для того, чтобы вернуть сразу два значения используется кортеж (кортежи пишутся в обычных скобках, значения следуют через запятую):

```
Prelude> properFraction 2.5
(2,0.5)
```

Для пар (кортеж, состоящий из двух элементов) определены две удобные функции извлечения элементов, их смысл можно понять по одним лишь типам:

```
fst :: (a, b) -> a
snd :: (a, b) -> b
```

Проверим:

```
Prelude> let x = properFraction 2.5
Prelude> (fst x, snd x)
(2, 0.5)
```

Мы бы и сами могли определить такие функции:

```
fst :: (a, b) -> a
fst (a, _) = a
```

```
snd :: (a, b) -> b
snd (_, b) = b
```

Между действительными числами: Кто-то написал очень хорошую функцию, но она определена на `Double`, а вам приходится использовать `Float`. Как быть? Нам поможет функция `realToFrac`:

```
Prelude> :i realToFrac
realToFrac :: (Real a, Fractional b) => a -> b
-- Defined in `GHC.Real'
```

Она принимает значение из класса `Real` и приводит его к значению, которое можно делить. Что это за класс `Real`? Математики наверное смекнут, что это противоположность комплексным числам (где-то должен быть определён тип или класс `Complex`, и он правда есть, но об этом в следующем разделе). При переходе к комплексным числам мы теряем способность сравнения на больше/меньше, но сохраняем возможность вычисления арифметических операций, поэтому класс `Real` это пересечение классов `Num` и `Ord`:

```
Prelude> :i Real
class (Num a, Ord a) => Real a where
  toRational :: a -> Rational
```

Здесь “пересечение” означает “и тот и другой”. Пересечение классов кодируется с помощью контекста. Вернёмся к нашему первому примеру:

```
Prelude> realToFrac (1::Float) + (1::Double)
2.0
```

Отметим, что этой функцией можно пользоваться не только для типов `Float` и `Double`, в Haskell возможны самые экзотические числа.

Если преобразования между `Float` и `Double` происходят очень-очень часто, возможно имеет смысл воспользоваться специальными для `GHC` функциями: Они определены в модуле `GHC.Float`:


```
Prelude> :m +GHC.Float
Prelude GHC.Float> :t float2Double
float2Double :: Float -> Double
Prelude GHC.Float> :t double2float
double2Float :: Double -> Float
```

Документация

К этой главе мы уже рассмотрели основные конструкции языка и базовые типы. Если у вас есть какая-то задача, вы уже можете начать её решать. Для этого сначала нужно будет описать в типах проблему, затем выразить с помощью функций её решение.

Но не стоит писать все функции самостоятельно, если функция достаточно общая её наверняка кто-нибудь уже написал. Самые полезные функции и классы определены в модуле `Prelude` и основных стандартных библиотечных модулях. Было бы излишним описывать каждую функцию, книга превратилась бы в справочник. Вместо этого давайте научимся искать функции в документации. Нам понадобится умение составлять типы функций и небольшое знание английского языка.

Для начала о том, где находится документация к стандартным модулям. Если вы установили `ghc` вместе с `Haskell Platform` под Windows скорее всего во вкладке `Пуск`, там где иконка `ghc` там же находится и документация. В Linux необходимо найти директорию с документацией, скорее всего она в директории `/usr/local/share/doc/ghc/libraries`. Также документацию можно найти в интернете, наберите в поисковике `Haskell Hierarchical Libraries`. На главной странице документации вы найдёте огромное количество модулей. Нас пока интересуют разделы `Data` и `Prelude`. Разделы расположены по алфавиту. То что вы видите это стандартный вид документации в `Haskell`. Документация делается с помощью специального приложения `Haddock`, мы тоже научимся такие делать, но позже, пока мы попробуем разобраться с тем как искать в документации функции.

Предположим нам нужно вычислить длину списка. Нам нужна функция, которая принимает список и возвращает целое число, скорее всего её тип `[a] -> Int`, обычно во всех библиотечных функциях для целых

чисел используется тип `Int`, также на месте параметра используются буквы `a`, `b`, `c`. Мы можем открыть документацию к `Prelude` набрав в строке поиска тип `[a] -> Int`. Или поискать такую функцию в разделе функций для списков `List Operations`. Тогда мы увидим единственную функцию с таким типом, под говорящим именем `length`. Так мы нашли то, что искали.

Или мы ищем функцию, которая переворачивает список, нам нужна функция с типом `[a] -> [a]`. Таких функций в `Prelude` несколько, но имя `reverse` одной из них может намекнуть на её смысл.

Но одной `Prelude` мир стандартных функций Haskell не ограничивается, если вы не нашли необходимую вам функцию в `Prelude` её стоит поискать в других библиотечных модулях. Обычно функции разделяются по тому на каких типах они определены. Так например функция `sort :: Ord a => [a] -> [a]` определена не в `Prelude`, а в отдельном библиотечном модуле для списков он называется `Data.List`. Так же есть много других модулей для разных типов, таких как `Data.Bool`, `Data.Char`, `Data.Function`, `Data.Maybe` и многие другие. Не пугайтесь изобилия модулей постепенно они станут вашей опорой.

Для поиска в стандартных библиотеках есть замечательный интернет-сервис `Hoogle` (<http://www.haskell.org/hoogle/>). `Hoogle` может искать значения не только по имени, но и по типам. Например мы хотим узнать целочисленный код символа. Поиск по типу `Char -> Int` выдаёт искомую функцию `digitToInt`.

Краткое содержание

В этой главе мы познакомились с интерпретатором `ghci` и основными типами. Рассмотрели много примеров.

Типы

<code>Bool</code>	Основные операции: <code>&&</code> , <code> </code> , <code>not</code> , <code>if c then t else e</code>
<code>Char</code>	Значения пишутся в одинарных кавычках, как в <code>'H'</code> , <code>'+'</code>
<code>String</code>	Значения пишутся в двойных кавычках, как в <code>"Hello World"</code>
<code>Int</code>	Эффективные целые числа, но ограниченные
<code>Integer</code>	Не ограниченные целые числа, но не эффективные
<code>Double</code>	Числа с двойной точностью
<code>Float</code>	Числа с одинарной точностью
<code>Rational</code>	Дробные числа

Нам впервые встретились кортежи (на функции `properFraction`).

Кортежи используются для возвращения из функции нескольких значений. Элементы кортежа могут иметь разные типы. Для извлечения элементов из кортежей-пар используются функции `fst` и `snd`. Кортежи пишутся в скобках, и элементы разделены запятыми:

```
(a, b)
(a, b, c)
(a, b, c, d)
...
```

Классы

<code>Show</code>	Печать
<code>Eq</code>	Сравнение на равенство
<code>Num</code>	Сложение и умножение
<code>Fractional</code>	Деление

Особенности синтаксиса

Запись применения функции:

Префиксная Инфиксная

`add a b` `a `add` b`

`(+) a b` `a + b`

Также мы научились приводить одни численные типы к другим и пользоваться документацией.

Упражнения

- Напишите функцию `beside :: Nat -> Nat -> Bool`, которая будет возвращать `True` только в том случае, если два аргумента находятся рядом, то есть один из них можно получить через другой операцией `Succ`.
- Напишите функцию `beside2 :: Nat -> Nat -> Bool`, которая будет возвращать `True` только если аргументы являются соседями через некоторое другое число.
- Мы написали очень неэффективную функцию сложения натуральных чисел. Проблема в том, что число рекурсивных вызовов функции зависит от величины второго аргумента. Если мы захотим прибавить единицу к сотне, то порядок следования аргументов существенно повлияет на скорость вычисления. Напишите функцию, которая лишена этого недостатка.
- Напишите функцию возведения в степень `pow :: Nat -> Nat -> Nat`.
- Напишите тип, описывающий бинарные деревья `BinTree a`. Бинарное дерево может быть либо листом со значением типа `a`, либо хранить два поддерева.
- Напишите функцию `reverse :: BinTree a -> BinTree a`, которая переворачивает дерево. Она меняет местами два элемента в узле дерева.

- Напишите функцию `depth :: BinTree a -> Nat`, которая вычисляет глубину дерева, то есть самый длинный путь от корня дерева к листу.
- Напишите функцию `leaves :: BinTree a -> [a]`, которая переводит бинарное дерево в список, возвращая все элементы в листьях дерева.
- Обратите внимание на раздел `List Operations` в `Prelude`. Посмотрите на функции и их типы. Попробуйте догадаться по типу функции и названию что она делает.
- Попробуйте разобраться по документации с классами `Ord` (сравнение на больше/меньше), `Enum` (перечисления) и `Integral` (целые числа). Также стоит отметить класс `Floating`. Если у вас не получится, не беда, они обязательно встретятся нам вновь. Там и разберёмся.
- Найдите функцию, которая переставляет элементы пары местами (элементы могут быть разных типов). Потренируйтесь с кортежами. Определите аналоги функций `fst` и `snd` для не пар. Обратите внимание на то, что сочетание символов `(,)` это функция-конструктор пары:

```
Prelude> (,) "Hi" 101
("Hi",101)
Prelude> :t (,)
(,) :: a -> b -> (a, b)
```

Также определены `(,,)`, `(,,,)` и другие.

Типы

С помощью типов мы определяем все возможные значения в нашей программе. Мы определяем основные примитивы и способы их комбинирования. Например в типе `Nat`:

```
data Nat = Zero | Succ Nat
```

Один конструктор-примитив `Zero`, и один конструктор `Succ`, с помощью которого мы можем делать составные значения. Определив тип `Nat` таким образом, мы говорим, что значения типа `Nat` могут быть только такими:

```
Zero, Succ Zero, Succ (Succ Zero), Succ (Succ (Succ Zero)), ...
```

Все значения являются цепочками `Succ` с `Zero` на конце. Если где-нибудь мы попытаемся построить значение, которое не соответствует нашему типу, мы получим ошибку компиляции, то есть программа не пройдёт проверку типов. Так типы описывают множество допустимых значений.

Значения, которые проходят проверку типов мы будем называть *допустимыми*, а те, которые не проходят соответственно *недопустимыми*. Так например следующие значения недопустимы для `Nat`

```
Succ Zero Zero, Succ Succ, True, Zero (Zero Succ), ...
```

Недопустимых значений конечно гораздо больше. Такое проявляется и в естественном языке, бессмысленных комбинаций слов гораздо больше, чем осмысленных предложений. Обратите внимание на то, что мы говорим о значениях (не)допустимых для некоторого типа, например значение `True` допустимо для `Bool`, но недопустимо для `Nat`.

Сами типы строятся не произвольным образом. Мы узнали, что при их построении используются две основные операции, это сумма и произведение типов. Это говорит о том, что в типах должны быть какие-то закономерности, которые распространяются на все значения. В этой главе мы посмотрим на эти закономерности.

Структура алгебраических типов данных

Итак у нас лишь две операции: сумма и произведение. Давайте для начала рассмотрим два крайних случая.

- Только произведение типов

```
data T = Name T1 T2 ... TN
```

Мы говорим, что значение нашего нового типа **T** состоит из значений типов **T1**, **T2**, ... , **TN** и у нас есть лишь один способ составить значение этого типа. Единственное, что мы можем сделать это применить к значениям типов **Ti** конструктор **Name**.

Пример:

```
data Time = Time Hour Second Minute
```

- Только сумма типов

```
data T = Name1 | Name2 | ... | NameN
```

Мы говорим, что у нашего нового типа **T** может быть лишь несколько значений, и перечисляем их в альтернативах через знак **|**.

Пример:

```
data Bool = True | False
```

Сделаем первое наблюдение: каждое произведение типов определяет новый конструктор. Число конструкторов в типе равно числу альтернатив. Так в первом случае у нас была одна альтернатива и следовательно у нас был лишь один конструктор **Name**.

Имена конструкторов должны быть уникальными в пределах модуля. У нас нет таких двух типов, у которых совпадают конструкторы. Это говорит о том, что по имени конструктора компилятор знает значение какого типа он может построить.

Произведение типов состоит из конструктора, за которым через пробел идут подтипы. Такая структура не случайна, она копирует структуру функции. В качестве имени функции выступает

конструктор, а в качестве аргументов – значения заданных в произведении подтипов. Функция-конструктор после применения “оборачивает” значения аргументов и создаёт новое значение. За счёт этого мы могли бы определить типы по-другому. Мы могли бы определить их в стиле классов типов:

```
data Bool where
  True  :: Bool
  False :: Bool
```

Мы видим “класс” `Bool`, у которого два метода. Или определим в таком стиле `Nat`:

```
data Nat where
  Zero  :: Nat
  Succ  :: Nat -> Nat
```

Мы переписываем подтипы по порядку в аргументы метода. Или определим в таком стиле списки:

```
data [a] where
  []    :: [a]
  (:)   :: a -> [a] -> [a]
```

Конструктор пустого списка `[]` является константой, а конструктор объединения элемента со списком `(:)`, является функцией. Когда я говорил, что типы определяют примитивы и методы составления из примитивов, я имел ввиду, что некоторые конструкторы по сути являются константами, а другие функциями. Эти “методы” определяют базовые значения типа, все другие значения будут комбинациями базовых.

Структура констант

Мы уже знаем, что значения могут быть функциями и константами. Объявляя константу, мы даём имя-синоним некоторой комбинации базовых конструкторов. В функции мы говорим как по одним значениям получить другие. В этом и следующем разделе мы посмотрим на то, как типы определяют структуру констант и функций.

Давайте присмотримся к константам:


```
Succ (Succ Zero)
Neg (Add One (Mul Six Ten))
Not (Follows A (And A B))
Cons 1 (Cons 2 (Cons 3 (Cons 4 Nil)))
```

Заменяем все функциональные конструкторы на букву *f* (от слова *function*), а все примитивные конструкторы на букву *c* (от слова *constant*).

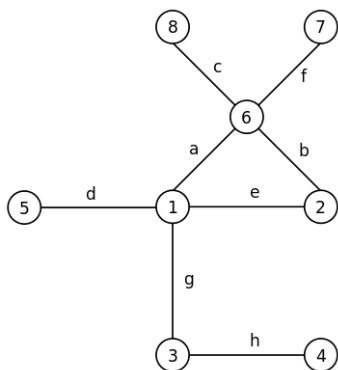
```
f (f c)
f (f c (f c c))
f (f c (f c c))
f c (f c (f c (f c c)))
```

Те кто знаком с теорией графов, возможно уже узнали в этой записи строчную запись дерева. Все значения в Haskell являются деревьями. Узел дерева содержит составной конструктор, а лист дерева содержит примитивный конструктор. Далее будет небольшой подраздел посвящённый терминологии теории графов, которая нам понадобится, будет много картинок, если вам это известно, то вы можете спокойно его пропустить.

Несколько слов о теории графов

Если вы не знакомы с теорией графов, то сейчас как раз самое время с ней познакомиться, хотя бы на уровне основных терминов. Теория графов изучает дискретные объекты в терминах зависимостей между объектами или связей. При этом объекты и связи можно изобразить графически.

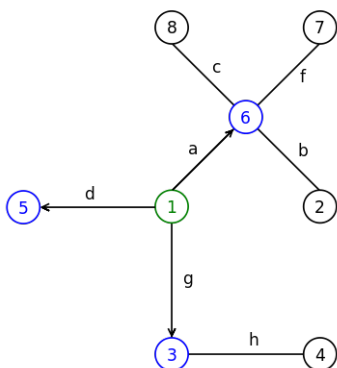
Граф состоит из *узлов* и *рёбер*, которые соединяют узлы. Приведём пример графа:



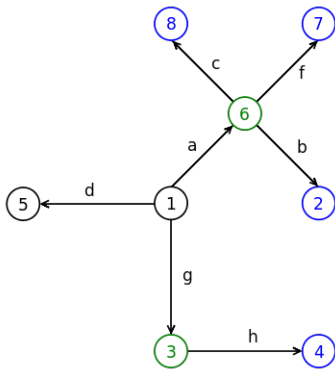
В этом графе восемь узлов, они пронумерованы, и восемь рёбер, они обозначены буквами. Теорию графов придумал Леонард Эйлер, когда решал задачу о кёнингсбергских мостах. Он решал задачу о том, можно ли обойти все семь кёнингсбергских мостов так, чтобы пройти по каждому лишь один раз. Эйлер представил мосты в виде рёбер а участки суши в виде узлов графа и показал, что это сделать нельзя. Но мы отвлеклись.

А что такое дерево? *Дерево* это такой связанный граф, у которого нет циклов. Несвязанный граф образует несколько островков, или множеств узлов, которые не соединены рёбрами. Циклы – это замкнутые последовательности рёбер. Например граф на рисунке выше не является деревом, но если мы сотрём ребро *e*, то у нас получится дерево.

Ориентированный граф – это такой граф, у которого все рёбра являются стрелками, они ориентированы, отсюда и название. При этом теперь каждое ребро не просто связывает узлы, но имеет начало и конец. В ориентированных деревьях обычно выделяют один узел, который называют *корнем*. Его особенность заключается в том, что все стрелки в ориентированном дереве как бы “разбегаются” от корня или сбегаются к корню. Корень определяет все стрелки в дереве. Ориентированное дерево похоже на иерархию. У нас есть корневой элемент и набор его дочерних поддеревьев, каждое из поддеревьев в свою очередь является ориентированным деревом и так далее. Проиллюстрируем на картинке, давайте сотрём ребро *e* и назовём первый узел корнем. Все наши стрелки будут идти от корня. Сначала мы проведём стрелки к узлам связанным с корнем:

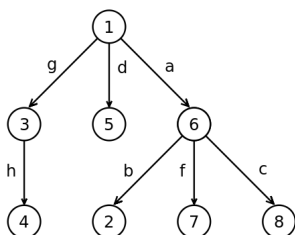


Затем представим, что каждый из этих узлов сам является корнем в своём дереве и повторим эту процедуру. На этом шаге мы дорисовываем стрелки в поддеревьях, которые находятся в узлах 3 и 6. Узел 5 является вырожденным деревом, в нём всего лишь одна вершина. Мы будем называть такие поддеревья *листьями*. А невырожденные поддеревья мы будем называть узлами. Корневой узел в данном поддереве называют родительским. А его соседние узлы, в которые направлены исходящие из него стрелки называют дочерними узлами. На предыдущем шаге у нас появился один родительский узел 1, у которого три дочерних узла: 3, 6, и 5. А на этом шаге у нас появились ещё два родительских узла 3 и 6. У узла 3 один дочерний узел (4), а у узла 6 – три дочерних узла (2, 8, 7).



Отметим, что положение узлов и рёбер на картинке не важно, главное это то, какие рёбра какие узлы соединяют. Мы можем перерисовать это дерево в более привычном виде.

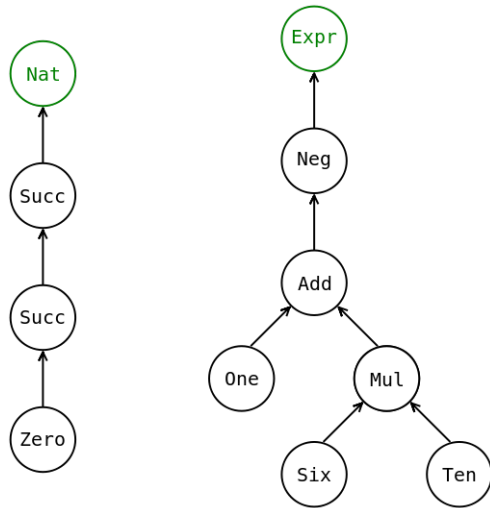
Теперь если вы посмотрите на константы в Haskell вы заметите, что очень похожи на деревья. Листья содержат примитивные конструкторы, а узлы – составные. Это происходит из-за того, что каждый конструктор содержит метку и набор подтипов. В этой аналогии метки становятся узлами, а подтипы-аргументы становятся поддеревьями.



Но есть одна тонкость, в которой заключается отличие констант Haskell от деревьев из теории графов. В теории графов порядок поддеревьев не важен, мы могли бы нарисовать поддеревья в любом порядке, главное сохранить связи. А в Haskell порядок следования аргументов в конструкторе важен.

На следующем рисунке изображены две константы:

`Succ (Succ Zero) :: Nat` и `Neg (Add One (Mul Six Ten)) :: Expr`. Но они изображены немного по-другому. Я перевернул стрелки и добавил корнем ещё один узел, это тип константы.



Стрелки перевёрнуты так, чтобы стрелки на картинке соответствовали стрелкам в типе конструктора. Например по виду узла `Succ :: Nat -> Nat`, можно понять, что это функция от одного аргумента, в неё впадает одна стрелка-аргумент и вытекает одна стрелка-значение. В конструктор `Mul` впадает две стрелки, значит это конструктор-функция от двух аргументов.

Константы похожи на деревья за счёт структуры операции произведения типов. В произведении типов мы пишем:

```
data Tnew = New T1 T2 ... Tn
```

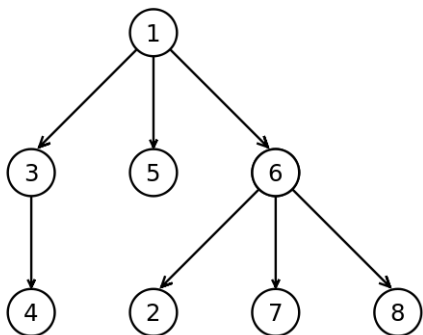
Так и получается, что у нашего узла `New` одна вытекающая стрелка, которая символизирует значение типа `Tnew` и несколько впадающих стрелок `T1`, `T2`, ..., `Tn`, они символизируют аргументы конструктора.

Потренируйтесь изображать константы в виде деревьев, вспомните константы из предыдущей главы, или придумайте какие-нибудь новые.

Строчная запись деревьев

Итак все константы в Haskell за счёт особой структуры построения типов являются деревьями, но мы программируем в текстовом редакторе, а не в редакторе векторной графики, поэтому нам нужен удобный способ строчной записи дерева. Мы им уже активно пользуемся, но сейчас давайте опишем его по-подробнее.

Мы сидим на корне дерева и спускаемся по его вершинам. Нам могут встретиться вершины двух типов узлы и листья. Сначала мы пишем имя в текущем узле, затем через пробел имена в дочерних узлах, если нам встречается невырожденный узел мы заключаем его в скобки. Давайте последовательно запишем в строчной записи дерево из первого примера:



Ориентированное дерево

Начнём с корня и будем последовательно дописывать поддеревья, точками обозначаются дочерние узлы, которые нам ещё предстоит дописать:

```
(1      .      .      .      )  
(1  (3 .)  5  (6 . . .))  
(1  (3 4)  5  (6 2 7 8))
```

Мы можем ставить любое число пробелов между дочерними узлами, здесь для наглядности точки выровнены. Так мы можем закодировать исходное дерево строкой. Часто самые внешние скобки опускаются. В итоге получилась такая запись:

```
tree = 1 (3 4) 5 (6 2 7 8)
```

По этой записи мы можем понять, что у нас есть два конструктора трёх аргументов **1** и **6**, один конструктор одного аргумента **3** и пять примитивных конструкторов. Точно так же мы строим и все другие константы в Haskell:

```
Succ (Succ (Succ Zero))  
Time (Hour 13) (Minute 10) (Second 0)  
Mul (Add One Ten) (Neg (Mul Six Zero))
```

За одним исключением, если конструктор бинарный, символьный (начинается с двоеточия), мы помещаем его между аргументов:

```
(One :+: Ten) :* (Neg (Six :* Zero))
```

Структура функций

Теперь мы разобрались с тем, что константы это деревья. Значит функции строят одни деревья из других. Как они это делают? Для этого в Haskell есть две операции: это композиция и декомпозиция деревьев. С помощью *композиции* мы строим из простых деревьев сложные, а с помощью *декомпозиции* разбиваем составные деревья на простейшие.

Композиция и декомпозиция объединены в одной операции, с которой мы уже встречались, это операция определения синонима. Давайте вспомним какое-нибудь объявление функции:

```
(+) a Zero      = a  
(+) a (Succ b) = Succ (a + b)
```

Смотрите в этой функции слева от знака равно мы проводим декомпозицию второго аргумента, а в правой части мы составляем новое дерево из тех значений, что были нами получены слева от знака равно. Или посмотрим на другой пример:

```
show (Time h m s) = show h ++ ":" ++ show m ++ ":" ++ show s
```

Слева от знака равно мы также выделили из составного дерева (**Time** h m s) три его дочерних для корня узла и связали их с переменными

`h`, `m` и `s`. А справа от знака равно мы составили из этих переменных новое выражение.

Итак операцию объявления синонима можно представить в таком виде:

`name декомпозиция = композиция`

В каждом уравнении у нас три части: новое имя, декомпозиция, поступающих на вход аргументов, и композиция нового значения. Остановимся поподробнее на каждой из этих операций.

Композиция и частичное применение

Композиция строится по очень простому правилу, если у нас есть значение `f` типа `a -> b` и значение `x` типа `a`, мы можем получить новое значение `(f x)` типа `b`. Это основное правило построения новых значений, поэтому давайте запишем его отдельно:

$$\frac{f :: a \rightarrow b, \quad x :: a}{(f \ x) :: b}$$

Сверху от черты, то что у нас есть, а снизу от черты то, что мы можем получить. Это операция называется *применением* или аппликацией.

Выражения, полученные таким образом, напоминают строчную запись дерева, но есть одна тонкость, которую мы обошли стороной. В случае деревьев мы строили только константы, и конструктор получал столько аргументов, сколько у него было дочерних узлов (или подтипов). Так мы строили константы. Но в Haskell мы можем с помощью применения строить функции на лету, передавая меньшее число аргументов, этот процесс называется *частичным применением* или каррированием (*currying*). Поясним на примере, предположим у нас есть функция двух аргументов:

```
add :: Nat -> Nat -> Nat
add a b = ...
```

На самом деле компилятор воспринимает эту запись так:

```
add :: Nat -> (Nat -> Nat)
add a b = ...
```

Функция `add` является функцией одного аргумента, которая в свою очередь возвращает функцию одного аргумента (`Nat -> Nat`). Когда мы пишем в где-нибудь в правой части функции:

```
... = ... (add Zero (Succ Zero)) ...
```

Компилятор воспринимает эту запись так:

```
... = ... ((add Zero) (Succ Zero)) ...
```

Присмотримся к этому выражению, что изменилось? У нас появились новые скобки, вокруг выражения `(add Zero)`. Давайте посмотрим как происходит применение:

$$\frac{\text{add} :: \text{Nat} \rightarrow (\text{Nat} \rightarrow \text{Nat}), \quad \text{Zero} :: \text{Nat}}{(\text{add Zero}) :: \text{Nat} \rightarrow \text{Nat}}$$

Итак применение функции `add` к `Zero` возвращает новую функцию `(add Zero)`, которая зависит от одного аргумента. Теперь применим к этой функции второе значение:

$$\frac{(\text{add Zero}) :: \text{Nat} \rightarrow \text{Nat}, \quad (\text{Succ Zero}) :: \text{Nat}}{((\text{add Zero}) (\text{Succ Zero})) :: \text{Nat}}$$

И только теперь мы получили константу. Обратите внимание на то, что получившаяся константа не может принять ещё один аргумент. Поскольку в правиле для применения функция *f* должна *содержать стрелку*, а у нас есть лишь `Nat`, это значение может участвовать в других выражениях лишь на месте аргумента.

Тоже самое работает и для функций от большего числа аргументов, если мы пишем

```
fun :: a1 -> a2 -> a3 -> a4 -> res
```

```
... = fun a b c d
```

На самом деле мы пишем

```
fun :: a1 -> (a2 -> (a3 -> (a4 -> res)))
```

```
... = (((fun a) b) c) d
```


Это очень удобно. Так, определив лишь одну функцию `fun`, мы получили в подарок ещё три функции (`fun a`), (`fun a b`) и (`fun a b c`). С ростом числа аргументов растёт и число подарков. Если смотреть на функцию `fun`, как на функцию одного аргумента, то она представляется таким генератором функций типа `a2 -> a3 -> a4 -> res`, который зависит от параметра. Применение функций через пробел значительно упрощает процесс комбинирования функций.

Поэтому в Haskell аргументы функций, которые играют роль параметров или специфических флагов, то есть аргументы, которые меняются редко обычно пишутся в начале функции. Например

```
process :: Param1 -> Param2 -> Arg1 -> Arg2 -> Result
```

Два первых аргумента функции `process` выступают в роли параметров для генерации функций с типом `Arg1 -> Arg2 -> Result`.

Давайте потренируемся с частичным применением в интерпретаторе. Для этого загрузим модуль `Nat` из предыдущей главы:

```
Prelude> :l Nat
[1 of 1] Compiling Nat                ( Nat.hs, interpreted )
Ok, modules loaded: Nat.
*Nat> let add = (+) :: Nat -> Nat -> Nat
*Nat> let addTwo = add (Succ (Succ Zero))
*Nat> :t addTwo
addTwo :: Nat -> Nat
*Nat> addTwo (Succ Zero)
Succ (Succ (Succ Zero))
*Nat> addTwo (addTwo Zero)
Succ (Succ (Succ (Succ Zero)))
```

Сначала мы ввели локальную переменную `add`, и присвоили ей метод `(+)` из класса `Num` для `Nat`. Нам пришлось выписать тип функции, поскольку `ghci` не знает для какого экземпляра мы хотим определить этот синоним. В данном случае мы подсказали ему, что это `Nat`. Затем с помощью частичного применения мы объявили новый синоним `addTwo`, как мы видим из следующей строки это функция одного аргумента. Она принимает любое значение типа `Nat` и прибавляет к нему двойку. Мы видим, что этой функцией можно пользоваться также как и обычной функцией.

Попробуем выполнить тоже самое для функции с символьной записью имени:

```
*Nat> let add2 = (+) (Succ (Succ Zero))
*Nat> add2 Zero
Succ (Succ Zero)
```

Мы рассмотрели частичное применение для функций в префиксной форме записи. В префиксной форме записи функция пишется первой, затем следуют аргументы. Для функций в инфиксной форме записи существует два правила применения.

Это применение слева:

$$\frac{(*) :: a \rightarrow (b \rightarrow c), \quad x :: a}{(x \ *) :: b \rightarrow c}$$

И применение справа:

$$\frac{(*) :: a \rightarrow (b \rightarrow c), \quad x :: b}{(\ * \ x) :: a \rightarrow c}$$

Обратите внимание на типы аргумента и возвращаемого значения. Скобки в выражениях $(x*)$ и $(*x)$ обязательны. Применением слева мы фиксируем в бинарной операции первый аргумент, а применением справа – второй.

Поясним на примере, для этого давайте возьмём функцию минус $(-)$. Если мы напишем $(2-)$ 1 то мы получим 1, а если мы напишем (-2) 1, то мы получим -1. Проверим в интерпретаторе:

```
*Nat> (2-) 1
1
*Nat> (-2) 1
```

```
<interactive>:4:2:
  No instance for (Num (a0 -> t0))
    arising from a use of syntactic negation
  Possible fix: add an instance declaration for (Num (a0 -> t0))
  In the expression: - 2
  In the expression: (- 2) 1
  In an equation for `it`: it = (- 2) 1
```

Ох уж этот минус. Незадача. Ошибка произошла из-за того, что минус является хамелеоном. Если мы пишем -2, компилятор

воспринимает минус как унарную операцию, и думает, что мы написали константу минус два. Это сделано для удобства, но иногда это мешает. Это единственное такое исключение в Haskell. Давайте введём новый синоним для операции минус:

```
*Nat> let (#) = (-)
*Nat> (2#) 1
1
*Nat> (#2) 1
-1
```

Эти правила левого и правого применения работают и для буквенных имён в инфиксной форме записи:

```
*Nat> let minus = (-)
*Nat> (2 `minus` ) 1
1
*Nat> ( `minus` 2) 1
-1
```

Так если мы хотим на лету получить новую функцию, связав в функции второй аргумент мы можем написать:

```
... = ... ( `fun` x) ...
```

Частичное применение для функций в инфиксной форме записи называют *сечением* (section), они бывают соответственно левыми и правыми.

Связь с логикой

Отметим связь основного правила применения с Modus Ponens, известным правилом вывода в логике:

$$\frac{a \rightarrow b, \quad a}{b}$$

Оно говорит о том, что если у нас есть выражение из a следует b и мы знаем, что a истинно, мы смело можем утверждать, что b тоже истинно. Если перевести это правило на Haskell, то мы получим: Если у нас определена функция типа $a \rightarrow b$ и у нас есть значение типа a , то мы можем получить значение типа b .

Декомпозиция и сопоставление с образцом

Декомпозиция применяется слева от знака равно, при этом наша задача состоит в том, чтобы опознать дерево определённого вида и выделить из него некоторые поддеревья. Мы уже пользовались декомпозицией много раз в предыдущих главах, давайте выпишем примеры декомпозиции:

```
not :: Bool -> Bool
not True  = ...
not False = ...

xor :: Bool -> Bool -> Bool
xor a b = ...

show :: Show a => a -> String
show (Time h m s) = ...

addZero :: String -> String
addZero (a:[]) = ...
addZero as      = ...

(*) a Zero      = ...
(*) a (Succ b)  = ...
```

Декомпозицию можно проводить в аргументах функции. Там мы видим строчную запись дерева, в узлах стоят конструкторы (начинаются с большой буквы), переменные (с маленькой буквы) или символ безразличной переменной (подчёркивание).

С помощью конструкторов, мы указываем те части, которые обязательно должны быть в дереве для данного уравнения. Так уравнение

```
not True  = ...
```

сработает, только если на вход функции поступит значение `True`. Мы можем углубляться в дерево значения настолько, насколько нам позволят типы, так мы можем определить функцию:

```
is7 :: Nat -> Bool
is7 (Succ (Succ (Succ (Succ (Succ (Succ Zero))))) ) = True
is7 _                                             = False
```

С помощью переменных мы даём синонимы поддеревьям. Этими синонимами мы можем пользоваться в правой части функции. Так в уравнении

```
addZero (a:[])
```

мы извлекаем первый элемент из списка, и одновременно говорим о том, что список может содержать только один элемент. Отметим, что если мы хотим дать синоним всему дереву а не какой-то части, мы просто пишем на месте аргумента переменную, как в случае функции xor:

```
xor a b = ...
```

С помощью безразличной переменной говорим, что нам не важно, что находится у дерева в этом узле. Уравнения в определении синонима обходятся сверху вниз, поэтому часто безразличной переменной пользуются в смысле “а во всех остальных случаях”, как в:

```
instance Eq Nat where
  (==) Zero      Zero      = True
  (==) (Succ a) (Succ b) = a == b
  (==) _         _         = False
```

Переменные и безразличные переменные также могут уходить вглубь дерева сколь угодно далеко (или ввысь дерева, поскольку первый уровень в строчной записи это корень):

```
lessThan7 :: Nat -> Bool
lessThan7 (Succ (Succ (Succ (Succ (Succ (Succ (Succ _))))))) = False
lessThan7 _ = True
```

Декомпозицию можно применять только к значениям-константам. Проявляется интересная закономерность: если для композиции необходимым элементом было значение со стрелочным типом (функция), то в случае декомпозиции нам нужно значение с типом без стрелок (константа). Это говорит о том, что все функции будут полностью применены, то есть константы будут записаны в виде строчной записи дерева. Если мы ожидаем на входе функцию, то мы можем только дать ей синоним с помощью с помощью переменной или проигнорировать её безразличной переменной.

Как в

```
name (Succ (Succ Zero)) = ...  
name (Zero : Succ Zero : []) = ...
```

Но не

```
name Succ = ...  
name (Zero :) = ...
```

Отметим, что для композиции это допустимые значения, в первом случае это функция `Nat -> Nat`, а во втором типа `[Nat] -> [Nat]`.

Ещё одна особенность декомпозиции заключается в том, что при декомпозиции мы можем пользоваться только “настоящими” значениями, то есть конструкторами, объявленными в типах. В случае композиции мы могли пользоваться как конструкторами, так и синонимами.

Например мы не можем написать в декомпозиции:

```
name (add Zero Zero) = ...  
name (or (xor a b) True) = ...
```

В Haskell декомпозицию принято называть *сопоставлением с образцом* (pattern matching). Термин намекает на то, что в аргументе мы выписываем шаблон (или заготовку) для целого набора значений. Наборы значений могут получиться, если мы пользуемся переменными. Конструкторы дают нам возможность зафиксировать вид ожидаемого на вход дерева.

Проверка типов

В этом разделе мы поговорим об ошибках проверки типов. Почти все ошибки, которые происходят в Haskell, связаны с проверкой типов. Проверка типов происходит согласно правилам применения, которые встретились нам в разделе о композиции значений. Мы остановимся лишь на случае для префиксной формы записи, правила для сечений работают аналогично. Давайте вспомним основное правило:

$$\begin{array}{c} f :: a \rightarrow b, \quad x :: a \\ \hline (f \ x) :: b \end{array}$$

Что может привести к ошибке? В этом правиле есть два источника ошибки.

- Тип f не содержит стрелок, или f не является функцией.
- Типы x и аргумента для f не совпадают.

Вот и все ошибки. Универсальное представление всех функций в виде функций одного аргумента, значительно сокращает число различных видов ошибок. Итак мы можем ошибиться, применяя значение к константе и передав в функцию не то, что она ожидает.

Потренируемся в интерпретаторе, сначала попытаемся создать ошибку первого типа:

```
*Nat> Zero Zero
```

```
<interactive>:1:1:
```

```
The function `Zero' is applied to one argument,  
but its type `Nat' has none  
In the expression: Zero Zero  
In an equation for `it': it = Zero Zero
```

Если перевести на русский интерпретатор говорит:

```
*Nat> Zero Zero
```

```
<interactive>:1:1:
```

```
Функция 'Zero' применяется к одному аргументу,  
но её тип 'Nat' не имеет аргументов  
В выражении: Zero Zero  
В уравнении для `it': it = Zero Zero
```

Компилятор увидел применение функции f x , далее он посмотрел, что $x = \text{Zero}$, из этого на основе правила применения он сделал вывод о том, что f имеет тип $\text{Nat} \rightarrow t$, тогда он заглянул в f и нашёл там $\text{Zero} :: \text{Nat}$, что и привело к несовпадению типов.

Перейдём к ошибкам второго типа. Попробуем вызывать функции с неправильными аргументами:

```
*Nat> :m +Prelude
```

```
*Nat Prelude> not (Succ Zero)
```

```
<interactive>:9:6:
```

```
Couldn't match expected type `Bool' with actual type `Nat'  
In the return type of a call of `Succ'
```

In the first argument of ``not'`, namely ``(Succ Zero)'`

In the expression: `not (Succ Zero)`

Опишем действия компилятора в терминах правила применения. В этом выражении у нас есть три значения: `not`, `Succ` и `Zero`. Нам нужно узнать тип выражения и проверить правильно ли оно построено.

`not (Succ Zero)` - ?

`not :: Bool -> Bool, Succ :: Nat -> Nat, Zero :: Nat`

`f x, f = not` и `x = (Succ Zero)`

`f :: Bool -> Bool` следовательно `x :: Bool`

`(Succ Zero) :: Bool`

Воспользовавшись правилом применения мы узнали, что тип выражения `Succ Zero` должен быть равен `Bool`. Проверим, так ли это?

`(Succ Zero)` - ?

`Succ :: Nat -> Nat, Zero :: Nat`

`f x, f = Succ, x = Zero` следовательно `(f x) :: Nat`

`(Succ Zero) :: Nat`

Из этой цепочки следует, что `(Succ Zero)` имеет тип `Nat`. Мы пришли к противоречию и сообщаем об этом пользователю.

<interactive>:1:5:

He могу сопоставить ожидаемый тип `'Bool'` с выведенным `'Nat'`

В типе результата вызова `'Succ'`

В первом аргументе ``not'`, а именно ``(Succ Zero)'`

В выражении: `not (Succ Zero)`

Потренируйтесь в составлении неправильных выражений и посмотрите почему они не правильные. Мысленно сверьтесь с правилом применения в каждом из слагаемых.

Специализация типов при подстановке

Мы говорили о том, что тип аргумента функции и тип подставляемого значения должны совпадать, но на самом деле есть и другая возможность. Тип аргумента или тип значения могут быть полиморфными. В этом случае происходит специализация общего типа. Например, при выполнении выражения:


```
*Nat> Succ Zero + Zero
Succ (Succ Zero)
```

Происходит специализация общей функции $(+) :: \text{Num } a \Rightarrow a \rightarrow a \rightarrow a$ до функции $(+) :: \text{Nat} \rightarrow \text{Nat} \rightarrow \text{Nat}$, которая определена в экземпляре `Num` для `Nat`.

Проверка типов с контекстом

Предположим, что у функции f есть контекст, который говорит о том, что первый аргумент принадлежит некоторому классу $f :: C \ a \Rightarrow a \rightarrow b$, тогда значение, которое мы подставляем в функцию, должно быть экземпляром класса `C`.

Для иллюстрации давайте попробуем сложить логические значения:

```
*Nat Prelude> True + False
```

```
<interactive>:11:6:
  No instance for (Num Bool)
    arising from a use of `+'
  Possible fix: add an instance declaration for (Num Bool)
  In the expression: True + False
  In an equation for `it': it = True + False
```

Компилятор говорит о том, что для типа `Bool` не определён экземпляр для класса `Num`.

```
No instance for (Num Bool)
```

Запишем это в виде правила:

$$\frac{f :: C \ a \Rightarrow a \rightarrow b, \quad x :: T, \quad \text{instance } C \ T}{(f \ x) :: b}$$

Важно отметить, что x имеет конкретный тип `T`. Если x – значение, у которого тип с параметром, компилятор не сможет определить для какого типа конкретно мы хотим выполнить применение. Мы будем называть такую ситуацию неопределённостью:

```
x :: T \ a \Rightarrow a
f :: C \ a \Rightarrow a \rightarrow b
```

```
f x :: ??  -- неопределённость
```

Мы видим, что тип `x`, это какой-то тип, одновременно принадлежащий и классу `T` и классу `C`. Но мы не можем сказать какой это тип. У этого поведения есть исключение: по умолчанию числа приводятся к `Integer`, если они не содержат знаков после точки, и к `Double` – если содержат.

```
*Nat Prelude> let f = (1.5 + )
*Nat Prelude> :t f
f :: Double -> Double
*Nat Prelude> let x = 5 + 0
*Nat Prelude> :t x
x :: Integer
*Nat Prelude> let x = 5 + Zero
*Nat Prelude> :t x
x :: Nat
```

Умолчания определены только для класса `Num`. Для этого есть специальное ключевое слово `default`. В рамках модуля мы можем указать какие типы считаются числами по умолчанию. Например, так (такое умолчание действует в каждом модуле, но мы можем переопределить его):

```
default (Integer, Double)
```

Работает правило: если произошла неопределённость и один из участвующих классов является `Num`, а все остальные классы – это стандартные классы, определённые в `Prelude`, то компилятор начинает последовательно пробовать все типы, перечисленные за ключевым словом `default`, пока один из них не подойдёт. Если такого типа не окажется, компилятор скажет об ошибке.

Ограничение мономорфизма

С выводом типов в классах связана одна тонкость. Мы говорили, что не обязательно выписывать типы выражений, компилятор может вывести их самостоятельно. Например, мы постоянно пользуемся этим в интерпретаторе. Также когда мы говорили о частичном применении, мы сказали об очень полезном умолчании в типах функций. О том, что за счёт частичного применения, все функции являются функциями одного аргумента. Эта особенность позволяет записывать выражения очень кратко. Но иногда они получаются чересчур краткими, и вводят компилятор в заблуждение. Зайдём в интерпретатор:

```
Prelude> let add = (+)
Prelude> :t add
add :: Integer -> Integer -> Integer
```

Мы хотели определить синоним для метода плюс из класса `Num`, но вместо ожидаемого общего типа получили более частный. Сработало умолчание для численного типа. Но зачем оно сработало? Если мы попробуем дать синоним методу из класса `Eq`, ситуация станет ещё более странной:

```
Prelude> let eq = (==)
Prelude> :t eq
eq :: () -> () -> Bool
```

Мы получили какую-то ерунду. Если мы попытаемся загрузить модуль с этими определениями:

```
module MR where
```

```
add = (+)
eq  = (==)
```

то получим:

```
*MR> :l MR
[1 of 1] Compiling MR                ( MR.hs, interpreted )
```

```
MR.hs:4:7:
  Ambiguous type variable `a0' in the constraint:
    (Eq a0) arising from a use of `=='
  Possible cause: the monomorphism restriction applied to the following:
    eq :: a0 -> a0 -> Bool (bound at MR.hs:4:1)
  Probable fix: give these definition(s) an explicit type signature
                  or use -XNoMonomorphismRestriction
  In the expression: (==)
  In an equation for `eq': eq = (==)
Failed, modules loaded: none.
```

Компилятор жалуется о том, что в определении для `eq` ему встретилась неопределённость и он не смог вывести тип. Если же мы допишем недостающие типы:

```
module MR where
```

```
add :: Num a => a -> a -> a
add = (+)
eq :: Eq a => a -> a -> Bool
eq  = (==)
```

то всё пройдет гладко:

```
Prelude> :l MR
[1 of 1] Compiling MR                ( MR.hs, interpreted )
Ok, modules loaded: MR.
*MR> eq 2 3
False
```

Но оказывается, что если мы допишем аргументы у функций и сотрём объявления, компилятор сможет вывести тип, и тип окажется общим. Это можно проверить в интерпретаторе. Для этого начнём новую сессию:

```
Prelude> let eq a b = (==) a b
Prelude> :t eq
eq :: Eq a => a -> a -> Bool
Prelude> let add a = (+) a
Prelude> :t add
add :: Num a => a -> a -> a
```

Запишите эти выражения в модуле без типов и попробуйте загрузить. Почему так происходит? По смыслу определения

```
add a b = (+) a b
add     = (+)
```

ничем не отличаются друг от друга, но второе сбивает компилятор толку. Компилятор путается из-за того, что второй вариант похож на определение константы. Мы с вами знаем, что выражение справа от знака равно является функцией, но компилятор, посчитав аргументы слева от знака равно, думает, что это возможно константа, потому что она выглядит как константа. У таких возможно-констант есть специальное имя, они называются константными аппликативными формами (constant applicative form или сокращённо CAF). Константы можно вычислять один раз, на то они и константы. Но если тип константы перегружен, и мы не знаем что это за тип (если пользователь не подсказал нам об этом в объявлении типа), то нам приходится вычислять его каждый раз заново. Посмотрим на пример:

```
res = s + s

s = someLongLongComputation 10

someLongLongComputation :: Num a => a -> a
```

Здесь значение `s` содержит результат вычисления какой-то большой-пребольшой функции. Перед компилятором стоит задача вывода типов. По тексту можно определить, что у `s` и `res` некоторый числовой тип. Проблема в том, что поскольку компилятор не знает какой тип у `s` конкретно в выражении `s + s`, он вынужден вычислить `s` дважды. Это привело разработчиков Haskell к мысли о том, что все выражения, которые выглядят как константы должны вычисляться как константы, то есть лишь один раз. Это ограничение называют ограничением *мономорфизма*. По умолчанию все константы должны иметь конкретный тип, если только пользователь не укажет обратное в типе или не подскажет компилятору косвенно, подставив неопределённое значение в другое значение, тип которого определён. Например, такой модуль загрузится без ошибок:

```
eqToOne = eq one
```

```
eq = (==)
```

```
one :: Int
one = 1
```

Только в этом случае мы не получим общего типа для `eq`: компилятор постарается вывести значение, которое не содержит контекста. Поэтому получится, что функция `eq` определена на `Int`. Эта очень спорная особенность языка, поскольку на практике получается так, что ситуации, в которых она мешает, возникают гораздо чаще. Немного забегаая вперёд, отметим, что это поведение компилятора по умолчанию, и его можно изменить. Компилятор даже подсказал нам как это сделать в сообщении об ошибке:

```
Probable fix: give these definition(s) an explicit type signature
or use -XNoMonomorphismRestriction
```

Мы можем активировать расширение языка, которое отменяет это ограничение. Сделать это можно несколькими способами. Мы можем запустить интерпретатор с флагом `-XNoMonomorphismRestriction`:

```
Prelude> :q
Leaving GHCi.
$ ghci -XNoMonomorphismRestriction
Prelude> let eq = (==)
Prelude> :t eq
eq :: Eq a => a -> a -> Bool
```

или в самом начале модуля написать:

```
{-# Language NoMonomorphismRestriction #-}
```

Расширение будет действовать только в рамках данного модуля.

Рекурсивные типы

Обсудим ещё одну особенность системы типов Haskell. Типы могут быть рекурсивными, то есть одним из подтипов в определении типа может быть сам определяемый тип. Мы уже пользовались этим в определении для `Nat`

```
data Nat = Zero | Succ Nat
```

Видите, во второй альтернативе участвует сам тип `Nat`. Это приводит к бесконечному числу значений. Таким простым и коротким определением мы описываем все положительные числа. Рекурсивные определения типов приводят к рекурсивным функциям. Помните, мы определяли сложение и умножение:

```
(+) a Zero      = a
(+) a (Succ b)  = Succ (a + b)

(*) a Zero      = Zero
(*) a (Succ b)  = a + (a * b)
```

И та и другая функция получились рекурсивными. Они следуют по одному сценарию: сначала определяем базу рекурсии~– тот случай, в котором мы заканчиваем вычисление функции, и затем определяем путь к базе~– цепочку рекурсивных вызовов.

Рассмотрим тип по-сложнее. Списки:

```
data [a] = [] | a : [a]
```

Деревья значений для `Nat` напоминают цепочку конструкторов `Succ`, которая венчается конструктором `Zero`. Дерево значений для списка отличается лишь тем, что теперь у каждого конструктора `Succ` есть отросток, который содержит значение некоторого типа `a`. Значение заканчивается пустым списком `[]`.

Мы можем предположить, что функции для списков также будут рекурсивными. Это и правда так. Посмотрим на три основные функции для списков. Все они определены в **Prelude**. Начнём с функции преобразования всех элементов списка:

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

Посмотрим как она работает:

```
Prelude> map (+100) [1,2,3]
[101,102,103]
Prelude> map not [True, True, False, False, False]
[False,False,True,True,True]
Prelude> :m +Data.Char
Prelude Data.Char> map toUpper "Hello World"
"HELLO WORLD"
```

Теперь опишем эту функцию. Базой рекурсии будет случай для пустого списка. В нём мы говорим, что если элементы закончились, нам нечего больше преобразовывать, и возвращаем пустой список. Во втором уравнении нам встретится узел дерева, который содержит конструктор **:**, а в дочерних узлах сидят элемент списка **a** и оставшаяся часть списка **as**. В этом случае мы составляем новый список, элемент которого содержит преобразованный элемент (**f a**) исходного списка и оставшуюся часть списка, которую мы также преобразуем с помощью функции **map**:

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

Какое длинное объяснение для такой короткой функции! Надеюсь, что мне не удалось сбить вас с толку. Обратите внимание на то, что поскольку конструктор символьный (начинается с двоеточия) мы пишем его между дочерними поддеревьями, а не сначала. Немного отвлекитесь и поэкспериментируйте с этой функцией в интерпретаторе, она очень важная. Составляйте самые разные списки. Чтобы не перенабивать каждый раз списки водите синонимы с помощью **let**.

Перейдём к следующей функции. Это функция фильтрации:

```
filter :: (a -> Bool) -> [a] -> [a]
```

Она принимает предикат и список. Угадайте, что она делает:

```
Prelude Data.Char> filter isUpper "Hello World"
"HW"
Prelude Data.Char> filter even [1,2,3,4,5]
[2,4]
Prelude Data.Char> filter (>10) [1,2,3,4,5]
[]
```

Да, она оставляет лишь те элементы, на которых предикат вернёт истину. Потренируйтесь и с этой функцией.

Теперь определение:

```
filter :: (a -> Bool) -> [a] -> [a]
filter p [] = []
filter p (x:xs) = if p x then x : filter p xs else filter p xs
```

Попробуйте разобраться с ним самостоятельно, по аналогии с map. Оно может показаться немного громоздким, но это ничего, совсем скоро мы узнаем как записать его гораздо проще.

Рассмотрим ещё одну функцию для списков, она называется функцией свёртки:

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

Визуально её действие можно представить как замену всех конструкторов в дереве значения на подходящие по типу функции. В этой маленькой функции кроется невероятная сила. Посмотрим на несколько примеров:

```
Prelude Data.Char> :m -Data.Char
Prelude> let xs = [1,2,3,4,5]
Prelude> foldr (:) [] xs
[1,2,3,4,5]
```

Мы заменили конструкторы на самих себя и получили исходный список, теперь давайте сделаем что-нибудь более конструктивное. Например вычислим сумму всех элементов или произведение:

```
Prelude> foldr (+) 0 xs
15
Prelude> foldr (*) 1 xs
120
```



```
Prelude> foldr max (head xs) xs
```

```
5
```

Краткое содержание

В этой главе мы присмотрелись к типам и узнали как ограничения, общие для всех типов, сказываются на структуре значений. Мы узнали, что константы в Haskell очень похожи на деревья, а запись констант – на строчную запись дерева. Также мы присмотрелись к функциям и узнали, что операция определения синонима состоит из композиции и декомпозиции значений.

name декомпозиция = композиция

Существует несколько правил для построения композиций:

- Одно для функций в префиксной форме записи:

$$\frac{f :: a \rightarrow b, \quad x :: a}{(f \ x) :: b}$$

- И два для функций в инфиксной форме записи:

Это левое сечение:

$$\frac{(*) :: a \rightarrow (b \rightarrow c), \quad x :: a}{(x \ *) :: b \rightarrow c}$$

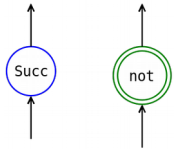
И правое сечение:

$$\frac{(*) :: a \rightarrow (b \rightarrow c), \quad x :: b}{(* \ x) :: a \rightarrow c}$$

Декомпозиция происходит в аргументах функции. С её помощью мы можем извлечь из составной константы-дерева какую-нибудь часть или указать на какие константы мы реагируем в данном уравнении.

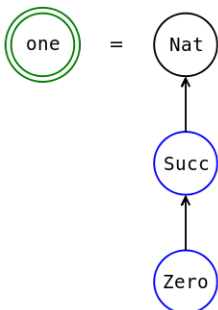
Ещё мы узнали о *частичном применении*. О том, что все функции в Haskell являются функциями одного аргумента, которые возвращают константы или другие функции одного аргумента.

Мы потренировались в составлении неправильных выражений и посмотрели как компилятор на основе правил применения узнаёт что они неправильные. Мы узнали, что такое ограничение мономорфизма и как оно появляется. Также мы присмотрелись к рекурсивным функциям.



Упражнения

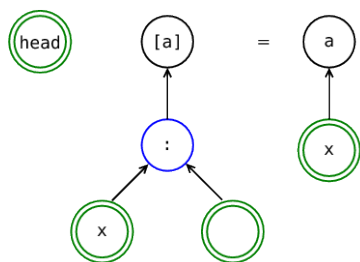
- Составьте в интерпретаторе как можно больше неправильных выражений и посмотрите на сообщения об ошибках. Разберитесь почему выражение оказалось неправильным. Для этого проверьте типы с помощью правил применения. Составьте несколько выражений, ведущих к ошибке из-за ограничения мономорфизма.
- Потренируйтесь в интерпретаторе с функциями `map`, `filter` и `foldr`. Попробуйте их с самыми разными функциями. Воспользуйтесь и теми функциями, что были определены в прошлой главе в тексте или в упражнениях.
- В этой главе было много картинок и графических аналогий, попробуйте попрограммировать в картинках. Нарисуйте определённые нами функции или какие-нибудь новые в виде деревьев. Например, это можно сделать так. Мы будем отличать конструкторы от синонимов. Конструкторы будем рисовать в одинарном кружке, а синонимы в двойном.



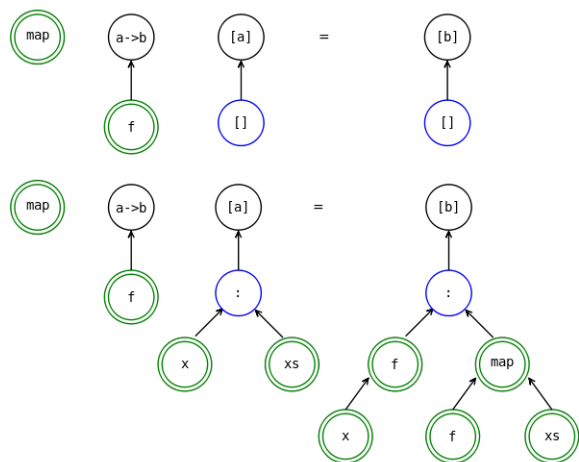
Мы будем все функции писать также как и прежде, но вместо аргументов слева от знака равно и выражений справа от знака равно, будем рисовать деревья.

Например, объявим простой синоним-константу. Мы будем дорисовывать сверху типы значений вместо объявления типа функции.

Несколько функций для списков. Извлечение первого элемента и функция преобразования всех элементов списка. Попробуйте в таком же духе определить несколько функций.



Функция извлечения первого элемента списка



Функция преобразования элементов списка

Декларативный и композиционный стиль

В Haskell существует несколько встроенных выражений, которые облегчают построение функций и делают код более наглядным. Их можно разделить на два вида: выражения, которые поддерживают *декларативный стиль* (declarative style) определения функций, и выражения которые поддерживают *композиционный стиль* (expression style).

Что это за стили? В декларативном стиле определения функций больше похожи на математическую нотацию, словно это предложения языка. В композиционном стиле мы строим из маленьких выражений более сложные, применяем к этим выражениям другие выражения и строим ещё большие.

В Haskell есть полноценная поддержка и того и другого стиля, поэтому конструкции которые мы рассмотрим в этой главе будут по смыслу дублировать друг друга. Выбор стиля скорее дело вкуса, существуют приверженцы и того и другого стиля, поэтому разработчики Haskell не хотели никого ограничивать.

Локальные переменные

Вспомним формулу вычисления площади треугольника по трём сторонам:

$$S = \sqrt{p \cdot (p - a) \cdot (p - b) \cdot (p - c)}$$

Где a , b и c – длины сторон треугольника, а p это полупериметр.

Как бы мы определили эту функцию теми средствами, что у нас есть? Наверное, мы бы написали так:

$$\text{square } a \ b \ c = \text{sqrt } (p \ a \ b \ c * (p \ a \ b \ c - a) * (p \ a \ b \ c - b) * (p \ a \ b \ c - c))$$

$$p \ a \ b \ c = (a + b + c) / 2$$

Согласитесь это не многим лучше чем решение в лоб:

```
square a b c = sqrt ((a+b+c)/2 * ((a+b+c)/2 - a) * ((a+b+c)/2 - b) *
((a+b+c)/2 - c))
```

И в том и в другом случае нам приходится дублировать выражения, нам бы хотелось чтобы определение выглядело так же, как и обычное математическое определение:

```
square a b c = sqrt (p * (p - a) * (p - b) * (p - c))
```

```
p = (a + b + c) / 2
```

Нам нужно, чтобы `p` знало, что `a`, `b` и `c` берутся из аргументов функции `square`. В этом нам помогут локальные переменные.

where-выражения

В декларативном стиле для этого предусмотрены **where**-выражения. Они пишутся так:

```
square a b c = sqrt (p * (p - a) * (p - b) * (p - c))
  where p = (a + b + c) / 2
```

Или так:

```
square a b c = sqrt (p * (p - a) * (p - b) * (p - c)) where
  p = (a + b + c) / 2
```

За определением функции следует специальное слово **where**, которое вводит локальные имена-синонимы. При этом аргументы функции включены в область видимости имён. Синонимов может быть несколько:

```
square a b c = sqrt (p * pa * pb * pc)
  where p  = (a + b + c) / 2
        pa = p - a
        pb = p - b
        pc = p - c
```

Отметим, что отступы обязательны. `Haskell` по отступам понимает, что эти выражения относятся к **where**.

Как и в случае объявления функций порядок следования локальных переменных в **where**-выражении не важен. Главное чтобы в выражениях справа от знака равно мы пользовались именами из списка аргументов исходной функции или другими определёнными именами.

Локальные переменные видны только в пределах той функции, в которой они вводятся.

Что интересно, слева от знака равно в **where**-выражениях можно проводить декомпозицию значений, также как и в аргументах функции:

```
pred :: Nat -> Nat
pred x = y
  where (Succ y) = x
```

Эта функция делает тоже самое что и функция

```
pred :: Nat -> Nat
pred (Succ y) = y
```

В **where**-выражениях можно определять новые функции а также выписывать их типы:

```
add2 x = succ (succ x)
  where succ :: Int -> Int
        succ x = x + 1
```

А можно и не выписывать, компилятор догадается:

```
add2 x = succ (succ x)
  where succ x = x + 1
```

Но иногда это бывает полезно, при использовании классов типов, для избежания неопределённости применения.

Приведём ещё один пример. Посмотрим на функцию фильтрации списков, она определена в **Prelude**:

```
filter :: (a -> Bool) -> [a] -> [a]
filter p []      = []
filter p (x:xs) = if p x then x : rest else rest
  where rest = filter p xs
```

Мы определили локальную переменную *rest*, которая указывает на рекурсивный вызов функции на оставшейся части списка.

where-выражения определяются для каждого уравнения в определении функции:

```

even :: Nat -> Bool
even Zero      = res
  where res = True
even (Succ Zero) = res
  where res = False
even x = even res
  where (Succ (Succ res)) = x

```

Конечно в этом примере **where** не нужны, но здесь они приведены для иллюстрации привязки **where**-выражения к данному уравнению. Мы определили три локальных переменных с одним и тем же именем.

where-выражения могут быть и у значений, которые определяются внутри **where**-выражений. Но лучше избегать сильно вложенных выражений.

let-выражения

В композиционном стиле функция вычисления площади треугольника будет выглядеть так:

```

square a b c = let p = (a + b + c) / 2
               in sqrt (p * (p - a) * (p - b) * (p - c))

```

Слова **let** и **in** – ключевые. Выгодным отличием **let**-выражений является то, что они являются обычными выражениями и не привязаны к определённому месту как **where**-выражения. Они могут участвовать в любой части обычного выражения:

```

square a b c = let p = (a + b + c) / 2
               in sqrt ((let pa = p - a in p * pa) *
                        (let pb = p - b
                          pc = p - c
                        in pb * pc))

```

В этом проявляется их принадлежность композиционному стилю. **let**-выражения могут участвовать в любом подвыражении, они также группируются скобками. А **where**-выражения привязаны к уравнениям в определении функции.

Также как и в **where**-выражениях, в **let**-выражениях слева от знака равно можно проводить декомпозицию значений.

```

pred :: Nat -> Nat
pred x = let (Succ y) = x
         in y

```

Определим функцию фильтрации списков через `let`:

```
filter :: (a -> Bool) -> [a] -> [a]
filter p []      = []
filter p (x:xs) =
  let rest = filter p xs
  in if p x then x : rest else rest
```

Декомпозиция

Декомпозиция или сопоставление с образцом позволяет выделять из составных значений, простейшие значения с помощью которых они были построены

```
pred (Succ x) = x
```

и организовывать условные вычисления которые зависят от вида поступающих на вход функции значений

```
not True  = False
not False = True
```

Сопоставление с образцом

Декомпозицию в декларативном стиле мы уже изучили, это обычный случай разбора значений в аргументах функции. Рассмотрим одну полезную возможность при декомпозиции. Иногда нам хочется провести декомпозицию и дать псевдоним всему значению. Это можно сделать с помощью специального символа `@`.

Например определим функцию, которая возвращает соседние числа для данного числа Пеано:

```
beside :: Nat -> (Nat, Nat)
beside Zero      = error "undefined"
beside x@(Succ y) = (y, Succ x)
```

В выражении `x@(Succ y)` мы одновременно проводим разбор и даём имя всему значению.

case-выражения

Оказывается декомпозицию можно проводить в любом выражении, для этого существуют `case`-выражения:


```

data AnotherNat = None | One | Two | Many
    deriving (Show, Eq)

toAnother :: Nat -> AnotherNat
toAnother x =
    case x of
        Zero           -> None
        Succ Zero      -> One
        Succ (Succ Zero) -> Two
        _              -> Many

fromAnother :: AnotherNat -> Nat
fromAnother None      = Zero
fromAnother One       = Succ Zero
fromAnother Two       = Succ (Succ Zero)
fromAnother Many     = error "undefined"

```

Слова **case** и **of** – ключевые. Выгодным отличием **case**-выражений является то, что нам не приходится каждый раз выписывать имя функции. Обратите внимание на то, что в **case**-выражениях также можно пользоваться обычными переменными и безымянными переменными.

Для проведения декомпозиции по нескольким переменным можно воспользоваться кортежами. Например определим знакомую функцию равенства для **Nat**:

```

instance Eq Nat where
    (==) a b =
        case (a, b) of
            (Zero, Zero)      -> True
            (Succ a', Succ b') -> a' == b'
            _                  -> False

```

Мы проводим сопоставление с образцом по кортежу (a, b), соответственно слева от знака **->** мы проверяем значения в кортежах, для этого мы также заключаем значения в скобки и пишем их через запятую.

Давайте определим функцию **filter** в ещё более композиционном стиле. Для этого мы заменим в исходном определении **where** на **let** и декомпозицию в аргументах на **case**-выражение:

```
filter :: (a -> Bool) -> [a] -> [a]
filter p a =
  case a of
    []      -> []
    x:xs    -> let rest = filter p xs
                in  if (p x)
                    then (x:rest)
                    else rest
```

Условные выражения

С условными выражениями мы уже сталкивались в сопоставлении с образцом. Например в определении функции `not`:

```
not True  = False
not False = True
```

В зависимости от поступающего значения мы выбираем одну из двух альтернатив. Условные выражения в сопоставлении с образцом позволяют реагировать лишь на частичное (с учётом переменных) совпадение дерева значения в аргументах функции.

Часто нам хочется определить более сложные условия для альтернатив. Например, если значение на входе функции больше 2, но меньше 10, верни `A`, а если больше 10, верни `B`, а во всех остальных случаях верни `C`. Или если на вход поступила строка состоящая только из букв латинского алфавита, верни `A`, а в противном случае верни `B`. Нам бы хотелось реагировать лишь в том случае, если значение некоторого типа `a` удовлетворяет некоторому предикату. Предикатами обычно называют функции типа `a -> Bool`. Мы говорим, что значение удовлетворяет предикату, если предикат для этого значения возвращает `True`.

Охранные выражения

В декларативном стиле условные выражения представлены *охранными выражениями* (`guards`). Предположим у нас есть тип:

```
data HowMany = Little | Enough | Many
```

И мы хотим написать функцию, которая принимает число людей, которые хотят посетить выставку, а возвращает значение типа

HowMany. Эта функция оценивает вместительность выставочного зала. С помощью охранных выражений мы можем написать её так:

```
hallCapacity :: Int -> HowMany
hallCapacity n
  | n < 10    = Little
  | n < 30    = Enough
  | True     = Many
```

Специальный символ `|` уже встречался нам в определении типов. Там он играл роль разделителя альтернатив в сумме типов. Здесь же он разделяет альтернативы в условных выражениях. Сначала мы пишем `|` затем выражение-предикат, которое возвращает значение типа **Bool**, затем равно и после равно – возвращаемое значение. Альтернативы так же как и в случае декомпозиции аргументов функции обходятся сверху вниз, до тех пор пока в одной из альтернатив предикат не вернёт значение **True**. Обратите внимание на то, что нам не нужно писать во второй альтернативе:

```
| 10 <= n && n < 30 = Enough
```

Если вычислитель дошёл до этой альтернативы, значит значение точно больше либо равно **10**. Поскольку в предыдущей альтернативе предикат вернул **False**.

Предикат в последней альтернативе является константой **True**, он пройдёт сопоставление с любым значением `n`. В данном случае, если учесть предыдущие альтернативы мы знаем, что если вычислитель дошёл до последней альтернативы, значение `n` больше либо равно **30**. Для повышения наглядности кода в **Prelude** определена специальная константа-синоним значению **True** под именем `otherwise`.

Определим функцию `filter` для списков в более декларативном стиле, для этого заменим `if`-выражение в исходной версии на охранные выражения:

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

Или мы можем разместить охранные выражения по-другому:

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

Отметим то, что локальная переменная `rest` видна и в той и в другой альтернативе. Вы спокойно можете пользоваться локальными переменными в любой части уравнения, в котором они определены.

Определим с помощью охранных выражений функцию `all`, она принимает предикат и список, и проверяет удовлетворяют ли все элементы списка данному предикату.

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

С помощью охранных выражений можно очень наглядно описывать условные выражения. Но иногда можно обойтись и простыми логическими операциями. Например функцию `all` можно было бы определить так:

```
all :: (a -> Bool) -> [a] -> Bool
all p [] = True
all p (x:xs) = p x && all p xs
```

Или так:

```
all :: (a -> Bool) -> [a] -> Bool
all p xs = null (filter notP xs)
    where notP x = not (p x)
```

Или даже так:

```
import Prelude(all)
```

Функция `null` определена в `Prelude` она возвращает `True` только если список пуст.

if-выражения

В композиционном стиле в качестве условных выражений используются уже знакомые нам `if`-выражения. Вспомним как они выглядят:

```
a = if bool
    then x1
    else x2
```

Слова **if**, **then** и **else** – ключевые. Тип `a`, `x1` и `x2` совпадают.

Любое охранный выражение, в котором больше одной альтернативы, можно представить в виде **if**-выражения и наоборот. Перепишем все функции их предыдущего подраздела с помощью **if**-выражений:

```
hallCapacity :: Int -> HowMany
hallCapacity n =
    if (n < 10)
    then Little
    else (if n < 30
        then Enough
        else Many)
```

```
all :: (a -> Bool) -> [a] -> Bool
all p [] = True
all p (x:xs) = if (p x) then all p xs else False
```

Определение функций

Под функцией мы понимаем составной синоним, который принимает аргументы, возможно разбирает их на части и составляет из этих частей новые выражения. Теперь посмотрим как такие синонимы определяются в каждом из стилей.

Уравнения

В декларативном стиле функции определяются с помощью уравнений. Пока мы видели лишь этот способ определения функций, примерами могут служить все предыдущие примеры. Вкратце напомним, что функция определяется набором уравнений вида:

```
name декомпозиция1 = композиция1
name декомпозиция2 = композиция2
...
name декомпозицияN = композицияN
```

Где `name` – имя функции. В декомпозиции происходит разбор поступающих на вход значений, а в композиции происходит составление значения результата. Уравнения обходятся вычислителем сверху вниз до тех пор пока он не найдёт такое уравнение, для

которого переданные в функции значения не подойдут в указанный в декомпозиции шаблон значений (если сопоставление с образцом аргументов пройдёт успешно). Как только такое уравнение найдено, составляется выражение справа от знака равно (композиция). Это значение будет результатом функции. Если такое уравнение не будет найдено программа остановится с ошибкой.

К примеру попробуйте вычислить в интерпретаторе выражение `notT False`, для такой функции:

```
notT :: Bool -> Bool
notT True = False
```

Что мы увидим?

```
Prelude> notT False
*** Exception: <interactive>:1:4-20: Non-exhaustive patterns in function notT
```

Интерпретатор сообщил нам о том, что он не нашёл уравнения для переданного в функцию значения.

Безымянные функции

В композиционном стиле функции определяются по-другому. Это необычный метод, он пришёл в Haskell из лямбда-исчисления. Функции строятся с помощью специальных конструкций, которые называются лямбда-функциями. По сути лямбда-функции являются безымянными функциями. Давайте посмотрим на лямбда функцию, которая прибавляет к аргументу единицу:

```
\x -> x + 1
```

Для того, чтобы превратить лямбда-функцию в обычную функцию мысленно замените знак `\` на имя `noName`, а стрелку на знак равно:

```
noName x = x + 1
```

Мы получили обычную функцию Haskell, с такими мы уже много раз встречались. Зачем специальный синтаксис для определения безымянных функций? Ведь можно определить её в виде уравнений. К тому же кому могут понадобиться безымянные функции? Ведь смысл функции в том, чтобы выделить определённый шаблон поведения и затем сослаться на него по имени функции.

Смысл безымянной функции в том, что ею, также как и любым другим элементом композиционного стиля, можно пользоваться в любой части обычных выражений. С её помощью мы можем создавать функции “на лету”. Предположим, что мы хотим профильтровать список чисел, мы хотим выбрать из них лишь те, что меньше 10, но больше 2, и к тому же они должны быть чётными. Мы можем написать:

```
f :: [Int] -> [Int]
f = filter p
  where p x = x > 2 && x < 10 && even x
```

При этом нам приходится давать какое-нибудь имя предикату, например `p`. С помощью безымянной функции мы могли бы написать так:

```
f :: [Int] -> [Int]
f = filter (\x -> x > 2 && x < 10 && even x)
```

Смотрите мы составили предикат сразу в аргументе функции `filter`. Выражение `(\x -> x > 2 && x < 10 && even x)` является обычным значением.

Возможно у вас появился вопрос, где аргумент функции? Где тот список по которому мы проводим фильтрацию. Ответ на этот вопрос кроется в частичном применении. Давайте вычислим по правилу применения тип функции `filter`:

```
f :: (a -> Bool) -> [a] -> [a],    x :: (Int -> Bool)
-----
(f x) :: [Int] -> [Int]
```

После применения параметр `a` связывается с типом `Int`, поскольку при применении происходит сопоставление более общего предиката `a -> Bool` из функции `filter` с тем, который мы передали первым аргументом `Int -> Bool`. После этого мы получаем тип `(f x) :: [Int] -> [Int]` это как раз тип функции, которая принимает список целых чисел и возвращает список целых чисел. Частичное применение позволяет нам не писать в таких выражениях:

```
f xs = filter p xs
  where p x = ...
```

последний аргумент `xs`.

К примеру вместо

```
add a b = (+) a b
```

мы можем просто написать:

```
add = (+)
```

Такой стиль определения функций называют *бесточечным* (point-free).

Давайте выразим функцию `filter` с помощью лямбда-функций:

```
filter :: (a -> Bool) -> ([a] -> [a])
filter = \p -> \xs -> case xs of
  []      -> []
  (x:xs) -> let rest = filter p xs
             in if   p x
                then x : rest
                else rest
```

Мы определили функцию `filter` пользуясь только элементами композиционного стиля. Обратите внимание на скобки в объявлении типа функции. Я хотел напомнить вам о том, что все функции в Haskell являются функциями одного аргумента. Это определение функции `filter` как нельзя лучше подчёркивает этот факт. Мы говорим, что функция `filter` является функцией одного аргумента `p` в выражении `\p ->`, которая возвращает также функцию одного аргумента. Мы выписываем это в явном виде в выражении `\xs ->`. Далее идёт выражение, которое содержит определение функции.

Отметим, что лямбда функции могут принимать несколько аргументов, в предыдущем определении мы могли бы написать:

```
filter :: (a -> Bool) -> ([a] -> [a])
filter = \p xs -> case xs of
  ...
```

но это лишь синтаксический сахар, который разворачивается в предыдущую запись.

Для тренировки определим несколько стандартных функций для работы с кортежами с помощью лямбда-функций (все они определены в `Prelude`):


```
fst :: (a, b) -> a
fst = \(a, _) -> a
```

```
snd :: (a, b) -> b
snd = \(_, b) -> b
```

```
swap :: (a, b) -> (b, a)
swap = \(a, b) -> (b, a)
```

Обратите внимание на то, что все функции словно являются константами. Они не содержат аргументов. Аргументы мы “пристраиваем” с помощью безымянных функций.

Определим функции преобразования первого и второго элемента кортежа (эти функции определены в модуле `Control.Arrow`)

```
first :: (a -> a') -> (a, b) -> (a', b)
first = \f (a, b) -> (f a, b)
```

```
second :: (b -> b') -> (a, b) -> (a, b')
second = \f (a, b) -> (a, f b)
```

Также в `Prelude` есть полезные функции, которые превращают функции с частичным применением в обычные функции и наоборот:

```
curry :: ((a, b) -> c) -> a -> b -> c
curry = \f -> \a -> \b -> f (a, b)
```

```
uncurry :: (a -> b -> c) -> ((a, b) -> c)
uncurry = \f -> \a, b -> f a b
```

Функция `curry` принимает функцию двух аргументов для которой частичное применение невозможно. Это имитируется с помощью кортежей. Функция принимает кортеж из двух элементов. Функция `curry` (от слова каррирование, частичное применение) превращает такую функцию в обычную функцию Haskell. А функция `uncurry` выполняет обратное преобразование.

С помощью лямбда-функций можно имитировать локальные переменные. Так например можно переписать формулу для вычисления площади треугольника:

```
square a b c =
  (\p -> sqrt (p * (p - a) * (p - b) * (p - c)))
  ((a + b + c) / 2)
```

Смотрите мы определили функцию, которая принимает параметром полупериметр p и передала в неё значение $((a + b + c) / 2)$. Если в нашей функции несколько локальных переменных, то мы можем составить лямбда-функцию от нескольких переменных и подставить в неё нужные значения.

Какой стиль лучше?

Основной критерий выбора заключается в том, сделает ли этот элемент код более *ясным*. Наглядность кода станет залогом успешной поддержки. Его будет легче понять и улучшить при необходимости.

Далее мы рассмотрим несколько примеров определений из `Prelude` и подумаем, почему был выбран тот или иной стиль. Начнём с класса `Ord` и посмотрим на определения по умолчанию:

-- Тип упорядочивания

```
data Ordering = LT | EQ | GT
              deriving (Eq, Ord, Enum, Read, Show, Bounded)
```

```
class (Eq a) => Ord a where
  compare      :: a -> a -> Ordering
  (<), (<=), (>=), (>) :: a -> a -> Bool
  max, min     :: a -> a -> a

  -- Минимальное полное определение:
  --   (<=) или compare
  -- Использование compare может оказаться более
  -- эффективным для сложных типов.
```

```
compare x y
  | x == y   = EQ
  | x <= y   = LT
  | otherwise = GT
```

```
x <= y      = compare x y /= GT
x < y       = compare x y == LT
x >= y      = compare x y /= LT
x > y       = compare x y == GT
```

```
max x y
  | x <= y   = y
  | otherwise = x
min x y
  | x <= y   = x
  | otherwise = y
```

Все функции определены в декларативном стиле. Тип **Ordering** кодирует результат операции сравнения. Два числа могут быть либо равны (значение **EQ**), либо первое меньше второго (значение **LT**), либо первое больше второго (значение **GT**).

Обратите внимание на функцию `compare`. Мы не пишем дословное определение значений типа **Ordering**:

```
compare x y
  | x == y  = EQ
  | x < y   = LT
  | x > y   = GT
```

В этом случае функция `compare` была бы определена через две других функции класса **Ord**, а именно больше `>` и меньше `<`. Мы же хотим минимизировать число функций в этом определении. Поэтому вместо этого определения мы полагаемся на очерёдность обхода альтернатив в охранном выражении.

Если первый случай не прошёл, то во втором случае нет разницы между функциями `<` и `<=`. А если не прошёл и этот случай, то остаётся только вернуть значение **GT**. Так мы определили функцию `compare` через одну функцию класса **Ord**.

Теперь посмотрим на несколько полезных функций для списков. Посмотрим на три основные функции для списков, одна из них возможно вам уже порядком поднадоела:

-- Преобразование списка

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

-- Фильтрация списка

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

-- Свёртка списка

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

Приведём несколько примеров для функции `foldr`:

```
and, or :: [Bool] -> Bool
```

```
and = foldr (&&) True
```

```
or  = foldr (||) False
```

```
(++) :: [a] -> [a] -> [a]
```

```
[] ++ ys = ys
```

```
(x:xs) ++ ys = x : (xs ++ ys)
```

```
concat :: [[a]] -> [a]
```

```
concat = foldr (++) []
```

Функции `and` и `or` выполняют логические операции на списках. Так каждый конструктор `(:)` заменяется на соответствующую логическую операцию, а пустой список заменяется на значение, которое не влияет на результат выполнения данной логической операции. Имеется ввиду, что функции `(&& True)` и `(|| False)` дают тот же результат, что и функция `id x = x`. Функция `(++)` объединяет два списка, а функция `concat` выполняет ту же операцию, но на списке списков.

Функция `zip` принимает два списка и смешивает их в список пар. Как только один из списков оборвётся оборвётся и список-результат. Эта функция является частным случаем более общей функции `zipWith`, которая принимает функцию двух аргументов и два списка и составляет новый список попарных применений.

```
-- zip-ы
```

```
zip :: [a] -> [b] -> [(a, b)]
```

```
zip = zipWith (,)
```

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
```

```
zipWith z (a:as) (b:bs) = z a b : zipWith z as bs
```

```
zipWith _ _ _ = []
```

Посмотрим как работают эти функции в интерпретаторе:

```
Prelude> zip [1,2,3] "hello"
```

```
[(1,'h'),(2,'e'),(3,'l')]
```

```
Prelude> zipWith (+) [1,2,3] [3,2,1]
```

```
[4,4,4]
```

```
Prelude> zipWith (*) [1,2,3] [5,4,3,2,1]
```

```
[5,8,9]
```

Отметим, что в `Prelude` также определена обратная функция `unzip`:

```
unzip :: [(a,b)] -> ([a], [b])
```

Она берёт список пар и разбивает его на два списка.

Пока по этим определениям кажется, что композиционный стиль совсем нигде не применяется. Он встретился нам лишь в функции `break`. Но давайте посмотрим и на функции с композиционным стилем:

```
lines      :: String -> [String]
lines ""   = []
lines s    = let (l, s') = break (== '\n') s
              in  l : case s' of
                        []      -> []
                        (_:s'') -> lines s''
```

Функция `lines` разбивает строку на список строк. Эти строки были разделены в исходной строке символом переноса `'\n'`.

Функция `break` принимает предикат и список и возвращает два списка. В первом все элементы от начала списка, которые не удовлетворяют предикату, а во втором все остальные. Наш предикат `(== '\n')` выделяет все символы кроме переноса каретки. В строке

```
let (l, s') = break (== '\n') s
```

Мы сохраняем все символы до `'\n'` от начала строки в переменной `l`. Затем мы рекурсивно вызываем функцию `lines` на оставшейся части списка:

```
in  l : case s' of
      []      -> []
      (_:s'') -> lines s''
```

При этом мы пропускаем в `s'` первый элемент, поскольку он содержит символ переноса каретки.

Посмотрим на ещё одну функцию для работы со строками.

```
words      :: String -> [String]
words s    = case dropWhile Char.isSpace s of
  "" -> []
  s' -> w : words s'
      where (w, s'') = break Char.isSpace s'
```

Функция `words` делает тоже самое, что и `lines`, только теперь в качестве разделителя выступает пробел. Функция `dropWhile` отбрасывает от начала списка все элементы, которые удовлетворяют предикату. В строке

```
case dropWhile Char.isSpace s of
```

Мы одновременно отбрасываем все первые пробелы и готовим значение для декомпозиции. Далее мы рассматриваем два возможных случая для строк.

```
"" -> []  
s' -> w : words s' '  
      where (w, s'') = break Char.isSpace s'
```

Если строка пуста, то делать больше нечего. Если – нет, мы также как и в предыдущей функции применяем функцию `break` для того, чтобы выделить все элементы кроме пробела, а затем рекурсивно вызываем функцию `words` на оставшейся части списка.

Краткое содержание

В этой главе мы узнали очень много новых синтаксических конструкций для определения функций. Они появлялись парами. Сведём их в таблицу:

Элемент	Декларативный стиль	Композиционный
Локальные переменные	<code>where</code> -выражения	<code>let</code> -выражения
Декомпозиция	Сопоставление с образцом	<code>case</code> -выражения
Условные выражения	Охранные выражения	<code>if</code> -выражения
Определение функций	Уравнения	лямбда-функции

Особенности синтаксиса

Нам встретилась новая конструкция в сопоставлении с образцом:

```
beside :: Nat -> (Nat, Nat)  
beside Zero      = error "undefined"  
beside x@(Succ y) = (y, Succ x)
```

Она позволяет проводить декомпозицию и давать имя всему значению одновременно. Такие выражения `x@(...)` в англоязычной литературе принято называть `as-patterns`.

Упражнения

- В этой главе нам встретилось много полезных стандартных функций, потренируйтесь с ними в интерпретаторе. Вызывайте их с различными значениями, экспериментируйте.
- Попробуйте определить функции из предыдущих глав в чисто композиционном стиле.
- Посмотрите на те функции, которые мы прошли и попробуйте переписать их определения шиворот на выворот. Если вы видите, что элемент написан композиционным стилем перепишите его в декларативном и наоборот. Получившиеся функции могут показаться монстрами, но это упражнение может помочь вам в закреплении новых конструкций и почувствовать сильные и слабые стороны того или иного стиля.
- Определите модуль, который будет вычислять площади простых фигур, треугольника, окружности, прямоугольника, трапеции. Помните, что фигуры могут задаваться различными способами.
- Поток это бесконечный список, или список, у которого нет конструктора пустого списка:

```
data Stream a = a :& Stream a
```

Так например мы можем составить поток из всех чисел Пеано:

```
nats :: Nat -> Stream Nat
nats a = a :& nats (Succ a)
```

Или поток, который содержит один и тот же элемент:

```
constStream :: a -> Stream a
constStream a = a :& constStream a
```

Напишите модуль для потоков. В первую очередь нам понадобятся функции выделения частей потока, поскольку мы не сможем распечатать поток целиком (ведь он бесконечный):

```
-- Первый элемент потока
head :: Stream a -> a

-- Хвост потока, всё кроме первого элемента
tail :: Stream a -> Stream a

-- n-тый элемент потока
(!!) :: Stream a -> Int -> a

-- Берёт из потока несколько первых элементов:
take :: Int -> Stream a -> [a]
```

Имена этих функций будут совпадать с именами функций для списков чтобы избежать коллизий имён мы воспользуемся квалифицированным импортом функций. Делается это так:

```
import qualified Prelude as P( определения )
```

Слова `qualified` и `as` – ключевые. Теперь для использования функций из модуля `Prelude` мы будем писать `P.имяФункции`. Такие имена называются квалифицированными. Для того чтобы пользоваться квалифицированными именами только для тех функций, для которых возможна коллизия имён можно поступить так:

```
import qualified Prelude as P
import Prelude
```

Компилятор разберётся, какую функцию мы имеем в виду.

Для удобства тестирования можно определить такую функцию печати потоков:

```
instance Show a => Show (Stream a) where
    show xs = showInfinity (show (take 5 xs))
        where showInfinity x = P.init x P.++ "..."
```

Функция `P.init` выделяет все элементы списка кроме последнего. В данном случае она откусит от строки закрывающую скобку. После этого мы добавляем троеточие, как символ бесконечности списка.

Функции преобразования потоков:

-- Преобразование потока

```
map :: (a -> b) -> Stream a -> Stream b
```

-- Фильтрация потока

```
filter :: (a -> Bool) -> Stream a -> Stream a
```

-- zip-ы для потоков:

```
zip :: Stream a -> Stream b -> Stream (a, b)
```

```
zipWith :: (a -> b -> c) -> Stream a -> Stream b -> Stream c
```

Функция генерации потока:

```
iterate :: (a -> a) -> a -> Stream a
```

Эта функция принимает два аргумента: функцию следующего элемента потока и значение первого элемента потока и возвращает поток:

```
iterate f a = a :& f a :& f (f a) :& f (f (f a)) :& ...
```

Так с помощью этой функции можно создать поток всех чисел Пеано от нуля или постоянный поток:

```
nats          = iterate Succ Zero
constStream a = iterate (\x -> x) a
```

Возможно вас удивляет тот факт, что в этом упражнении мы оперируем бесконечными значениями, но пока мы не будем вдаваться в детали того как это работает, просто попробуйте определить этот модуль и посмотрите в интерпретаторе, что получится.

Функции высшего порядка

Функцией высшего порядка называют функцию, которая может принимать на вход функции или возвращать функции в качестве результата. За счёт частичного применения в Haskell все функции, которые принимают более одного аргумента, являются функциями высшего порядка.

В этой главе мы подробно обсудим способы составления функций, недаром Haskell – функциональный язык. В Haskell функции являются очень гибким объектом, они позволяют выделять сложные способы комбинирования значений. Часто за счёт развитых средств составления новых функций в Haskell пользователь определяет лишь базовые функции, получая остальные “на лету” применением двух-трёх операций, это выглядит примерно как $(2+3)*5$, где вместо чисел стоят базовые функции, а операции $+$ и $*$ составляют новые функции из простейших.

Обобщённые функции

В этом разделе мы познакомимся с несколькими функциями, которые принимают одни функции и составляют по ним другие. Эти функции используются в Haskell очень часто. Все они живут в модуле `Data.Function`. Модуль `Prelude` экспортирует их из этого модуля.

Функция тождества

Начнём с самой простой функции. Это функция `id`. Она ничего не делает с аргументом, просто возвращает его:

```
id :: a -> a
id x = x
```

Зачем нам может понадобиться такая функция? Сама по себе она бесполезна. Она приобретает ценность при совместном использовании с другими функциями, поэтому пока мы не будем приводить примеров.

Константная функция

Следующая функция `const` принимает значение и возвращает постоянную функцию. Эта функция будет возвращать константу для любого переданного в неё значения:

```
const :: a -> b -> a
const a _ = a
```

Функция `const` является конструктором постоянных функций, так например мы получаем пятёрки на любой аргумент:

```
Prelude> let onlyFive = const 5
Prelude> :t onlyFive
onlyFive :: b -> Integer
Prelude> onlyFive "Hi"
5
Prelude> onlyFive (1,2,3)
5
Prelude> map onlyFive "abracadabra"
[5,5,5,5,5,5,5,5,5,5]
```

С её помощью мы можем легко построить и постоянную функцию двух аргументов:

```
const2 a = const (const a)
```

Вспомним определение для `&&`:

```
(&&) :: Bool -> Bool -> Bool
(&&) True  x  = x
(&&) False _  = False
```

С помощью функций `id` и `const` мы можем сократить число аргументов и уравнений:

```
(&&) :: Bool -> Bool -> Bool
(&&) a = if a then id else (const False)
```

Также мы можем определить и логическое “или”:

```
(||) :: Bool -> Bool -> Bool
(||) a = if a then (const True) else id
```

Функция композиции

Функция композиции принимает две функции и составляет из них последовательное применение функций:

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

Это очень полезная функция. Она позволяет нанизывать функции друг на друга. Мы перехватываем выход второй функции, сразу подставляем его в первую и возвращаем её выход в качестве результата. Например перевернём список символов и затем сделаем все буквы заглавными:

```
Prelude> :m +Data.Char
Prelude Data.Char> (map toUpper . reverse) "abracadabra"
"ARBADACARBA"
```

Приведём пример посложнее:

```
add :: Nat -> Nat -> Nat
add a Zero      = a
add a (Succ b) = Succ (add a b)
```

Если мы определим функцию свёртки для `Nat`, которая будет заменять в значении типа `Nat` конструкторы на соответствующие по типу функции:

```
foldNat :: a -> (a -> a) -> Nat -> a
foldNat zero succ Zero      = zero
foldNat zero succ (Succ b) = succ (foldNat zero succ b)
```

То мы можем переписать с помощью функции композиции эту функцию так:

```
add :: Nat -> Nat -> Nat
add = foldNat id (Succ . )
```

Куда делись аргументы? Они выражаются через функции `id` и `(.)`. Поведение этой функции лучше проиллюстрировать на примере. Пусть у нас есть два числа типа `Nat`:

```
two      = Succ (Succ Zero)
three    = Succ (Succ (Succ Zero))
```

Вычислим

```
add two three
```

Вспомним о частичном применении:

```
add two three
=> (add two) three
=> (foldNat id (Succ .) (Succ (Succ Zero))) three
```

Теперь функция свёртки заменит все конструкторы `Succ` на `(Succ .)`, а конструкторы `Zero` на `id`:

```
=> ((Succ .) ((Succ .) id)) three
```

Что это за монстр?

```
((Succ .) ((Succ .) id))
```

Функция `(Succ .)` это левое сечение операции `(.)`. Эта функция, которая принимает функции и возвращает функции. Она принимает функцию и навешивает на её выход конструктор `Succ`. Давайте упростим это большое выражение с помощью определений функций `(.)` и `id`:

```
((Succ .) ((Succ .) id))
=> (Succ .) (\x -> Succ (id x))
=> (Succ .) (\x -> Succ x)
=> \x -> Succ (Succ x)
```

Теперь нам осталось применить к этой функции наше второе значение:

```
(\x -> Succ (Succ x)) three
=> Succ (Succ three)
=> Succ (Succ (Succ (Succ (Succ x))))
```

Так мы получили, что и ожидалось от сложения. За каждый конструктор `Succ` в первом аргументе мы добавляем применение `Succ` к результату, а вместо `Zero` протаскиваем через `id` второй аргумент.

Аналогия с числами

С помощью функции композиции мы можем нанизывать друг на друга списки функций. Попробуем в интерпретаторе:

```
Prelude> let f = foldr (.) id [sin, cos, sin, cos, exp, (+1), tan]
Prelude> f 2
0.6330525927559899
```

```
Prelude> f 15  
0.7978497904127007
```

Функция `foldr` заменит в списке каждый конструктор `(:)` на функцию композиции, а пустой список на функцию `id`. В результате получается композиция из всех функций в списке.

Это очень похоже на сложение или умножение чисел в списке. При этом в качестве нуля (для сложения) или единицы (для умножения) мы используем функцию `id`. Мы пользуемся тем, что по определению для любой функции `f` выполнены тождества:

```
f . id == f  
id . f == f
```

Поэтому мы можем использовать `id` в качестве накопителя результата композиции, как в случае:

```
Prelude> foldr (*) 1 [1,2,3,4]  
24
```

Если сравнить `(.)` с умножением, то `id` похоже на единицу, а `(const a)` на ноль. В самом деле для любой функции `f` и любого значения `a` выполнено тождество:

```
const a . f == const a
```

Мы словно умножаем функцию на ноль, делая её вычисление бессмысленным.

Функция перестановки

Функция перестановки `flip` принимает функцию двух аргументов и меняет аргументы местами:

```
flip :: (a -> b -> c) -> b -> a -> c  
flip f x y = f y x
```

К примеру:

```
Prelude> foldr (-) 0 [1,2,3,4]  
-2  
Prelude> foldr (flip (-)) 0 [1,2,3,4]  
-10
```

Иногда это бывает полезно.

Функция on

Функция `on` (от англ. на) перед применением бинарной функции пропускает аргументы через унарную функцию:

```
on :: (b -> b -> c) -> (a -> b) -> a -> a -> c
(*.) `on` f = \x y -> f x (*.) f y
```

Она часто используется в сочетании с функцией `sortBy` из модуля `Data.List`. Эта функция имеет тип:

```
sortBy :: (a -> a -> Ordering) -> [a] -> [a]
```

Она сортирует элементы списка согласно некоторой функции упорядочивания `f :: (a -> a -> Ordering)`. С помощью функции `on` мы можем легко составить такую функцию на лету:

```
let xs = [(3, "John"), (2, "Jack"), (34, "Jim"), (100, "Jenny"), (-3, "Josh")]
Prelude> :m +Data.List Data.Function
Prelude Data.List Data.Function>
Prelude Data.List Data.Function> sortBy (compare `on` fst) xs
[(-3,"Josh"),(2,"Jack"),(3,"John"),(34,"Jim"),(100,"Jenny")]
Prelude Data.List Data.Function> map fst (sortBy (compare `on` fst) xs)
[-3,2,3,34,100]
Prelude Data.List Data.Function> map snd (sortBy (compare `on` fst) xs)
["Josh","Jack","John","Jim","Jenny"]
```

Мы импортировали в интерпретатор модуль `Data.List` для функции `sortBy` а также модуль `Data.Function` для функции `on`. Они не импортируются модулем `Prelude`.

Выражением `(compare `on` fst)` мы составили функцию

```
\a b -> compare (fst a) (fst b)
```

```
fst = \ (a, b) -> a
```

Тем самым ввели функцию упорядочивания на парах, которая будет сравнивать пары по первому элементу. Отметим, что аналогичного эффекта можно добиться с помощью функции `comparing` из модуля `Data.Ord`.

Функция применения

Ещё одной очень полезной функцией является функция применения `($)`. Посмотрим на её определение:

```
($) :: (a -> b) -> a -> b
f $ x = f x
```

На первый взгляд её определение может показаться бессмысленным. Зачем нам специальный знак для применения, если у нас уже есть пробел? Для ответа на этот вопрос нам придётся познакомиться с приоритетом инфиксных операций.

Приоритет инфиксных операций

В Haskell очень часто используются бинарные операции для составления функций “на лету”. В этом помогает и частичное применение, мы можем в одном выражении применить к функции часть аргументов, построить из неё новую функцию с помощью какой-нибудь такой бинарной операции и всё это передать в другую функцию!

Для сокращения числа скобок нам понадобится разобраться в понятии приоритета операции. Так например в выражении

```
> 2 + 3 * 10
32
```

Мы полагаем, что умножение имеет больший приоритет чем сложение и со скобками это выражение будет выглядеть так:

```
> 2 + (3 * 10)
32
```

Фраза “больший приоритет” означает: сначала умножение потом сложение. Мы всегда можем изменить поведение по умолчанию с помощью скобок:

```
> (2 + 3) * 10
50
```

В Haskell приоритет функций складывается из двух понятий: старшинство и ассоциативность. Старшинство определяется числами, они могут быть от 0 до 9. Чем больше это число, тем выше приоритет функций.

Старшинство используется вычислителем для группировки разных операций, например (+) имеет старшинство 6, а (*) имеет старшинство 7. Поэтому интерпретатор сначала ставит скобки вокруг

выражения с (*), а затем вокруг (+). Считается, что обычное префиксное применение имеет высший приоритет 10. Нельзя задать приоритет выше применения, это значит, что операция “пробел” будет всегда выполняться первой.

Ассоциативность используется для группировки одинаковых операций, например мы видим:

```
1+2+3+4
```

Как нам быть? Мы можем группировать скобки слева направо:

```
((1+2)+3)+4
```

Или справа налево:

```
1+(2+(3+4))
```

Ответ на этот вопрос даёт ассоциативность, она бывает левая и правая. Например операции (+) (-) и (*) являются лево-ассоциативными, а операция возведения в степень (^) является право-ассоциативной.

```
1 + 2 + 3 == (1 + 2) + 3
1 ^ 2 ^ 3 == 1 ^ (2 ^ 3)
```

Приоритет функции можно узнать в интерпретаторе с помощью команды

```
:i:
```

```
*FunNat> :m Prelude
Prelude> :i (+)
class (Eq a, Show a) => Num a where
  (+) :: a -> a -> a
  ...
      -- Defined in GHC.Num
infixl 6 +
Prelude> :i (*)
class (Eq a, Show a) => Num a where
  ...
  (*) :: a -> a -> a
  ...
      -- Defined in GHC.Num
infixl 7 *
Prelude> :i (^)
(^) :: (Num a, Integral b) => a -> b -> a      -- Defined in GHC.Real
infixr 8 ^
```

Приоритет указывается в строчках `infixl 6 +` и `infixl 7 *`. Цифра указывает на старшинство операции, а суффикс `l` (от англ. `left` – левая) или `r` (от англ. `right` – правая) на ассоциативность.

Если мы создали свою функцию, мы можем определить для неё ассоциативность. Для этого мы пишем в коде:

```
module Fixity where

import Prelude(Num(..))

infixl 4 ***
infixl 5 +++
infixr 5 `neg`

(***) = (*)
(+++) = (+)
neg    = (-)
```

Мы ввели новые операции и поменяли старшинство операций сложения и умножения местами и изменили ассоциативность у вычитания.

Проверим в интерпретаторе:

```
Prelude> :l Fixity
[1 of 1] Compiling Fixity                ( Fixity.hs, interpreted )
Ok, modules loaded: Fixity.
*Fixity> 1 + 2 * 3
7
*Fixity> 1 +++ 2 *** 3
9
*Fixity> 1 - 2 - 3
-4
*Fixity> 1 `neg` 2 `neg` 3
2
```

Посмотрим как это вычислялось:

```
1 + 2 * 3 == 1 + (2 * 3)
1 +++ 2 *** 3 == (1 +++ 2) *** 3

1 - 2 - 3 == (1 - 2) - 3
1 `neg` 2 `neg` 3 == 1 `neg` (2 `neg` 3)
```

Также в Haskell есть директива `infix` это тоже самое, что и `infixl`.

Приоритет функции композиции

Посмотрим на приоритет функции композиции:

```
Prelude> :i (.)
(.) :: (b -> c) -> (a -> b) -> a -> c    -- Defined in GHC.Base
infixr 9 .
```

Она имеет высший приоритет. Она очень часто используется при определении функции в бесточечном стиле. Такая функция похожа на конвейер функций:

```
fun a = fun1 a . fun2 (x1 + x2) . fun3 . (+x1)
```

Приоритет функции применения

Теперь посмотрим на полное определение функции применения:

```
infixr 0 $

($) :: (a -> b) -> a -> b
f $ x = f x
```

Ответ на вопрос о полезности этой функции кроется в её приоритете. Ей назначен самый низкий приоритет. Она будет исполняться в последнюю очередь. Очень часто возникают ситуации вроде:

```
foldNat zero succ (Succ b) = succ (foldNat zero succ b)
```

С помощью функции применения мы можем переписать это определение так:

```
foldNat zero succ (Succ b) = succ $ foldNat zero succ b
```

Если бы мы написали без скобок:

```
... = succ foldNat zero succ b
```

То выражение было бы сгруппировано так:

```
... = (((succ foldNat) zero) succ) b
```

Но поскольку мы поставили барьер в виде операции (\$) с низким приоритетом, группировка скобок произойдёт так:

```
... = (succ $ ((foldNat zero) succ) b)
```

Это как раз то, что нам нужно. Преимущество этого подхода проявляется особенно ярко если у нас несколько вложенных функций на конце выражения:

```
xs :: [Int]
xs = reverse $ map ((+1) . (*10)) $ filter even $ ns 40

ns :: Int -> [Int]
ns 0 = []
ns n = n : ns (n - 1)
```

В списке `xs` мы сначала создаём в функции `ns` убывающий список чисел, затем оставляем лишь чётные, потом применяем два арифметических действия ко всем элементам списка, затем переворачиваем список.

Проверим работает ли это в интерпретаторе, заодно поупражняемся в композиционном стиле:

```
Prelude> let ns n = if (n == 0) then [] else n : ns (n - 1)
Prelude> let even x = 0 == mod x 2
Prelude> let xs = reverse $ map ((+1) . (*10)) $ filter even $ ns 20
Prelude> xs
[21,41,61,81,101,121,141,161,181,201]
```

Если бы не функция применения нам пришлось бы написать это выражение так:

```
xs = reverse (map ((+1) . (*10)) (filter even (ns 40)))
```

Функциональный калькулятор

Мне бы хотелось сделать акцент на одном из вступительных предложений этой главы:

За счёт развитых средств составления новых функций в Haskell пользователь определяет лишь базовые функции, получая остальные “на лету” применением двух-трёх операций, это выглядит примерно как $(2+3)*5$, где вместо чисел стоят базовые функции, а операции `+` и `*` составляют новые функции из простейших.

Такие обобщённые функции как `id`, `const`, `(.)`, `map` `filter` позволяют очень легко комбинировать различные функции. Бесточечный стиль записи функций превращает функции в простые значения или

значения-константы, которые можно подставлять в другие функции. В этом разделе мы немного потренируемся в перегрузке численных значений и превратим числа в функции, функции и в самом деле станут константами. Мы определим экземпляр `Num` для функций, которые возвращают числа. Смысл этих операций заключается в том, что теперь мы применяем обычные операции сложения умножения к функциям, аргумент которых совпадает по типу. Например для того чтобы умножить функции `\t -> t+2` и `\t -> t+3` мы составляем новую функцию `\t -> (t+2) * (t+3)`, которая получает на вход значение `t` применяет его к каждой из функций и затем умножает результаты:

```
module FunNat where
```

```
instance Num a => Num (t -> a) where
```

```
  (+) = fun2 (+)
```

```
  (*) = fun2 (*)
```

```
  (-) = fun2 (-)
```

```
  abs      = fun1 abs
```

```
  signum   = fun1 signum
```

```
  fromInteger = const . fromInteger
```

```
fun1 :: (a -> b) -> ((t -> a) -> (t -> b))
```

```
fun1 = (.)
```

```
fun2 :: (a -> b -> c) -> ((t -> a) -> (t -> b) -> (t -> c))
```

```
fun2 op a b = \t -> a t `op` b t
```

Функции `fun1` и `fun2` превращают функции, которые принимают значения, в функции, которые принимают другие функции. Загрузим модуль `FunNat` в интерпретатор и посмотрим что же у нас получилось:

```
Prelude> :l FunNat.hs
```

```
[1 of 1] Compiling FunNat                ( FunNat.hs, interpreted )
```

```
Ok, modules loaded: FunNat.
```

```
*FunNat> 2 2
```

```
2
```

```
*FunNat> 2 5
```

```
2
```

```
*FunNat> (2 + (+1)) 0
```

```
3
```

```
*FunNat> ((+2) * (+3)) 1
```

```
12
```

На первый взгляд кажется что выражение `2 2` не должно пройти проверку типов, ведь мы применяем значение к константе. Но на самом деле `2` это не константа, а значение `2 :: Num a => a` и подспудно к двойке применяется функция `fromInteger`. Поскольку в нашем модуле мы определили экземпляр `Num` для функций, второе число `2` было конкретизировано по умолчанию до `Integer`, а первое число `2` было конкретизировано до `Integer -> Integer`. Компилятор вывел из контекста, что под `2` мы понимаем функцию. Функция была создана с помощью метода `fromInteger`. Эта функция принимает любое значение и возвращает двойку.

Далее мы складываем и перемножаем функции словно это обычные значения. Что интересно мы можем составлять и такие выражения:

```
*FunNat> let f = ((+) - (*))
*FunNat> f 1 2
1
```

Как была вычислена эта функция? Мы определили экземпляр функций для значений типа `Num a => t -> a`. Если мы вспомним, что функция двух аргументов на самом деле является функцией одного аргумента: `Num a => t1 -> (t2 -> a)`, мы заметим, что тип `Num a => (t2 -> a)` принадлежит `Num`, теперь если мы обозначим его за `a'`, то мы получим тип `Num a' => t1 -> a'`, это совпадает с нашим исходным экземпляром.

Получается, что за счёт механизма частичного применения мы одним махом определили экземпляры `Num` для функций *любого* числа аргументов, которые возвращают значение типа `Num`.

Итак функция `f` имеет вид:

```
\t1 t2 -> (t1 + t2) - (t1 * t2)
```

Теперь давайте составим несколько выражений с обобщёнными функциями. Составим функцию, которая принимает один аргумент, умножает его на два, вычитает 10 и берёт модуль числа.

```
*FunNat> let f = abs $ id * 2 - 10
*FunNat> f 2
6
*FunNat> f 10
10
```

Давайте посмотрим как была составлена эта функция:

```
abs $ id * 2 - 10

=> abs $ (id * 2) - 10           -- приоритет умножения
=> abs $ (\x -> x * \x -> 2) - 10 -- развернём id и 2
=> abs $ (\x -> x * 2) - 10      -- по определению (*) для функций
=> abs $ (\x -> x * 2) - \x -> 10 -- развернём 10
=> abs $ \x -> (x * 2) - 10      -- по определению (-) для функций
=> \x -> abs x . \x -> (x * 2) - 10 -- по определению abs для функций
=> \x -> abs ((x * 2) - 10)     -- по определению (.)

=> \x -> abs ((x * 2) - 10)
```

Функция возведения в квадрат:

```
*FunNat> let f = id * id
*FunNat> map f [1,2,3,4,5]
[1,4,9,16,25]
*FunNat> map (id * id - 1) [1,2,3,4,5]
[0,3,8,15,24]
```

Обратите внимание на краткость записи. В этом выражении `(id * id - 1)` проявляется основное преимущество бесточечного стиля, избавившись от аргументов, мы можем пользоваться функциями так, словно это простые значения. Этот приём используется в Haskell очень активно. Пока нам встретились лишь две инфиксных операции для функций (это композиция и применение с низким приоритетом), но в будущем вы столкнётесь с целым морем подобных операций. Все они служат одной цели, они прячут аргументы функции, позволяя быстро составлять функции на лету из примитивов.

Возведём в четвёртую степень:

```
*FunNat> map (f . f) [1,2,3,4,5]
[1,16,81,256,625]
```

Составим функцию двух аргументов, которая будет вычислять сумму квадратов двух аргументов:

```
*FunNat> let x = const
*FunNat> let y = flip const
*FunNat> let d = x * x + y * y
*FunNat> d 1 2
5
*FunNat> d 3 2
13
```

Так мы составили функцию, ни прибегая к помощи аргументов. Эти выражения могут стать частью других выражений:

```
*FunNat> filter ((<10) . d 1) [1,2,3,4,5]
[1,2]
*FunNat> zipWith d [1,2,3] [3,2,1]
[10,8,10]
*FunNat> foldr (x*x - y*y) 0 [1,2,3,4]
-3721610024
```

Функции, возвращающие несколько значений

Как было сказано ранее функции, которые возвращают несколько значений, реализованы в Haskell с помощью кортежей. Например функция, которая расщепляет поток на голову и хвост выглядит так:

```
decons :: Stream a -> (a, Stream a)
decons (a :& as) = (a, as)
```

Здесь функция возвращает сразу два значения. Но всегда ли уместно пользоваться кортежами? Для композиции функций, которые возвращают несколько значений нам придётся разбирать возвращаемые значения с помощью сопоставления с образцом и затем использовать эти значения в других функциях. Посудите сами если у нас есть функции:

```
f :: a -> (b1, b2)
g :: b1 -> (c1, c2)
h :: b2 -> (c3, c4)
```

Мы уже не сможем комбинировать их так просто как если бы это были обычные функции без кортежей.

```
q x = (\(a, b) -> (g a, h b)) (f x)
```

В случае пар нам могут прийти на помощь функции `first` и `second`:

```
q = first g . second h . f
```

Если мы захотим составить какую-нибудь другую функцию из `q`, то ситуация заметно усложнится. Функции, возвращающие кортежи, сложнее комбинировать в бесточечном стиле. Здесь стоит вспомнить правило Unix.

Пишите функции, которые делают одну вещь, но делают её хорошо.

Функция, которая возвращает кортеж пытается сделать сразу несколько дел. И теряет в гибкости, ей трудно взаимодействовать с другими функциями. Старайтесь чтобы таких функций было как можно меньше.

Если функция возвращает несколько значений, попробуйте разбить её на несколько, которые возвращают лишь одно значение. Часто бывает так, что эти значения тесно связаны между собой и такую функцию не удаётся разбить на несколько составляющих. Если у вас появляется много таких функций, то это повод задуматься о создании нового типа данных.

Например в качестве точки на плоскости можно использовать пару (`Float`, `Float`). В этом случае, если вы начнёте писать модуль на геометрическую тему у вас появится много функций, которые принимают и возвращают точки:

```
rotate      :: Float -> (Float, Float) -> (Float, Float)
norm        :: (Float, Float) -> (Float, Float)
translate    :: (Float, Float) -> (Float, Float) -> (Float, Float)
...
```

Все они стараются делать несколько дел одновременно, возвращая кортежи. Но мы можем изменить ситуацию определением новых типов:

```
data Point  = Point  Float Float
data Vector = Vector Float Float
data Angle  = Angle  Float
```

Объявления функций станут более краткими и наглядными.

```
rotate      :: Angle  -> Point -> Point
norm        :: Point  -> Point
translate    :: Vector -> Point -> Point
...
```

Комбинатор неподвижной точки

Познакомимся с функцией `fix` или комбинатором неподвижной точки. По хорошему об этой функции следовало бы рассказать в разделе обобщённые функции. Но я пропустил её нарочно, для простоты

изложения. В этом разделе градус сложности резко подскакивает, если вы ранее не встречались с этой функцией она может показаться вам очень необычной. Для начала посмотрим на её тип:

```
Prelude> :m +Data.Function
Prelude Data.Function> :t fix
fix :: (a -> a) -> a
```

Странно, `fix` принимает функцию и возвращает значение, обычно всё происходит наоборот. Теперь посмотрим на определение:

```
fix f = let x = f x
        in x
```

Если вы запутались, то по смыслу это определение равносильно такому:

```
fix f = f (fix f)
```

Функция `fix` берёт функцию и начинает бесконечно нанизывать её саму на себя. Так мы получаем, что-то вроде:

```
f (f (f (f (...))))
```

Зачем нам такая функция? Помните в самом конце четвёртой главы в упражнениях мы составляли бесконечные потоки. Мы делали это так:

```
data Stream a = a :& Stream a

constStream :: a -> Stream a
constStream a = a :& constStream a
```

Если смотреть на функцию `constStream` очень долго, то рано или поздно в ней проглянет функция `fix`. Я нарочно не буду выписывать, а вы мысленно обозначьте `(a :&)` за `f` и `constStream a` за `fix f`. Получилось?

Через `fix` можно очень просто определить бесконечность для `Nat`, бесконечность это цепочка `Succ`, которая никогда не заканчивается `Zero`. Оказывается, что в Haskell мы можем составлять выражения с такими значениями (как вычислителю удаётся справиться с бесконечными выражениями мы обсудим попозже):

```
ghci Nat
*Nat>m + Data.Function
*Nat Data.Function> let infinity = fix Succ
*Nat Data.Function> infinity < Succ Zero
False
```

С помощью функции `fix` можно выразить любую рекурсивную функцию. Посмотрим на примере функции `foldNat`, у нас есть рекурсивное определение:

```
foldNat :: a -> (a -> a) -> Nat -> a
foldNat z s Zero      = z
foldNat z s (Succ n) = s (foldNat z s n)
```

Необходимо привести его к виду:

```
x = f x
```

Слева и справа мы видим повторяются выражения `foldNat z s`, обозначим их за `x`:

```
x :: Nat -> a
x Zero      = z
x (Succ n)  = s (x n)
```

Теперь перенесём первый аргумент в правую часть, сопоставление с образцом превратится в `case`-выражение:

```
x :: Nat -> a
x = \nat -> case nat of
    Zero      -> z
    Succ n    -> s (x n)
```

В правой части вынесем `x` из выражения с помощью лямбда функции:

```
x :: Nat -> a
x = (\t -> \nat -> case nat of
    Zero      -> z
    Succ n    -> s (t n)) x
```

Смотрите, для этого мы обозначили вхождение `x` в выражении справа за `t` и создали лямбда-функцию с таким аргументом.

Получилось! Мы пришли к виду комбинатора неподвижной точки:

```

x :: Nat -> a
x = f x
  where f = \t -> \nat -> case nat of
                        Zero    -> z
                        Succ n  -> s (t n)

```

Приведём в более человеческий вид:

```

foldNat :: a -> (a -> a) -> (Nat -> a)
foldNat z s = fix f
  where f t = \nat -> case nat of
                        Zero    -> z
                        Succ n  -> s (t n)

```

Краткое содержание

Основные функции высшего порядка

Мы познакомились с функциями из модуля `Data.Function`. Их можно разбить на несколько типов:

- Примитивные функции (генераторы функций).

```

id      = \x -> x
const a = \_ -> a

```

- Функции, которые комбинируют функции или функции и значения:

```

f . g = \x -> f (g x)
f $ x = f x

```

```

(.*.) `on` f = \x y -> f x .* f y

```

- Преобразователи функций, принимают функцию и возвращают функцию:

```

flip f = \x y -> f y x

```

- Комбинатор неподвижной точки:

```

fix f = let x = f x
        in  x

```

Приоритет инфиксных операций

Мы узнали о специальном синтаксисе для задания приоритета применения функций в инфиксной форме:

```
infixl 3 #  
infixr 6 `op`
```

Приоритет складывается из двух частей: старшинства (от 1 до 9) и ассоциативности (бывает левая и правая). Старшинство определяет распределение скобок между разными функциями:

```
infixl 6 +  
infixl 7 *
```

$1 + 2 * 3 == 1 + (2 * 3)$

А ассоциативность – между одинаковыми:

```
infixl 6 +  
infixr 8 ^
```

$1 + 2 + 3 == (1 + 2) + 3$
 $1 ^ 2 ^ 3 == 1 ^ (2 ^ 3)$

Мы узнали, что функции (\$) и (.) стоят на разных концах шкалы приоритетов функций и как этим пользоваться.

Упражнения

- Просмотрите написанные вами функции, или функции из примеров. Можно ли их переписать с помощью основных функций высшего порядка? Если да, то перепишите. Попробуйте определить их в бесточечном стиле.
- В прошлой главе у нас было упражнение о потоках. Сделайте поток экземпляром класса `Num`. Для этого поток должен содержать значения из класса `Num`. Методы из класса `Num` применяются поэлементно. Так сложение двух потоков будет выглядеть так:

```
(a1 :& a2 :& a3 :& ...) + (b1 :& b2 :& b3) ==  
== (a1 + b1 :& a2 + b2 :& a3 + b3 :& ...)
```

- Определите приоритет инфиксной операции (`:&`) так чтобы вам было удобно использовать её в сочетании с арифметическими операциями.
- Рассмотрим такой тип:

```
data St a b = St (a -> (b, St a b))
```

Этот тип хранит функцию, которая позволяет преобразовывать потоки значений. Определите функцию применения:

```
ap :: St a b -> [a] -> [b]
```

Она принимает ленту входящих значений и возвращает ленту выходов. Определите для этого типа несколько основных функций высшего порядка. Чтобы не возникало конфликта имён с модулем `Data.Function` мы не будем его импортировать. Вместо него мы импортируем модуль `Control.Category`. Он содержит класс:

```
class Category cat where
    id :: cat a a
    (.) :: cat b c -> cat a b -> cat a c
```

Если присмотреться к типам функций, можно понять, что тип-экземпляр `cat` принимает два параметра. Совсем как тип функции `(a -> b)`. Формально его можно записать в префиксной форме так `(->) a b`. Получается, что тип `cat` это что-то вроде функции. Это некоторые сущности, у которых есть понятия тождества и композиции.

Для обычных функций экземпляр класса `Category` уже определён. Но в этом модуле у нас есть ещё и необычные функции, функции которые преобразуют ленты значений. Функции `id` и `(.)` мы определим, сделав наш тип `St` экземпляром класса `Category`. Также определите постоянный преобразователь. Он на любой вход возвращает одно и то же число, и преобразователь, который будет накапливать сумму поступающих на вход значений, по-другому такой преобразователь называют интегратором:

```
const    :: a -> St b a
integral :: Num a => St a a
```

- Перепишите с помощью `fix` несколько стандартных функций для списков. Например `map`, `foldr`, `foldl`, `zip`, `repeat`, `cycle`, `iterate`.

Старайтесь найти наиболее краткое выражение, пользуйтесь функциями высшего порядка и частичным применением. Например рассмотрим функцию `repeat`:

```
repeat :: a -> [a]
repeat a = a : repeat a
```

Запишем с `fix`:

```
repeat a = fix $ \xs -> a : xs
```

Заметим, что мы можем избавиться от аргумента `xs` с помощью сечения:

```
repeat a = fix (a:)
```

Но мы можем пойти ещё дальше, если вспомним, что функция двух аргументов `(:)` является функцией от одного аргумента `(:) :: a -> ([a] -> [a])`, которая возвращает функцию одного аргумента:

```
repeat = fix . (:)
```

Смотрите в этом выражении мы составили композицию двух функций. Функция `(:)` примет первый аргумент и вернёт функцию, как раз то, что и нужно для `fix`.

Функторы и монады: теория

Мы научились комбинировать функции наиболее общего типа $a \rightarrow b$. В этой главе мы посмотрим на специальные функции и способы их комбинирования. Специальными функциями мы будем называть такие функции, результат которых имеет некоторую известную нам структуру. Среди них функции, которые могут вычислить значение или упасть, или функции, которые возвращают сразу несколько вариантов значений. Для составления таких функций из простейших в Haskell предусмотрено несколько классов типов. Это функторы и монады. Их мы и рассмотрим в этой главе.

Композиция функций

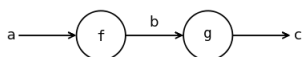
Центральной функцией этой главы будет функция композиции. Вспомним её определение для функций общего типа:

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

Композиция двух функций f и g это такая функция, в которой мы сначала применяем g , а затем f . Для того чтобы тип функции стал более наглядным, мы определим эту функцию немного по-другому. Мы поменяем аргументы местами.

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

Мы будем изображать функции кружками, а значения – стрелками. Значения словно текут от узла к узлу по стрелкам. Поскольку тип стрелки выходящей из f совпадает с типом стрелки входящей в g мы можем соединить их и получить составную функцию $(f >> g)$.



Класс Category

С помощью операции композиции можно обобщить понятие функции. Для этого существует класс **Category**:

```
class Category cat where
  id  :: cat a a
  (>>) :: cat a b -> cat b c -> cat a c
```

Функция `cat` это тип с двумя параметрами, в котором выделено специальное значение `id`, которое оставляет аргумент без изменений. Также мы можем составлять из простых функций сложные с помощью композиции, если функции совпадают по типу. Здесь мы для наглядности также заменили метод `(.)` на `(>>)`, но суть остаётся прежней. Для любого экземпляра класса должны выполняться свойства:

```
f >> id == f
id >> f == f

f >> (g >> h) == (f >> g) >> h
```

Первые два свойства говорят о том, что `id` является нейтральным элементом для `(>>)` слева и справа. Третье свойство говорит о том, что нам не важно в каком порядке проводить композицию. Можно проверить, что эти правила выполнены для функций.

Специальные функции

Все специальные функции, которые мы рассмотрим в этой главе будут иметь один и тот же тип:

```
a -> m b
```

Смотрите вместо произвольного типа `b` функция возвращает `m b`. Единственное, что будет меняться от раздела к разделу это тип `m`. Добавив этот тип к результату, мы сузили область значений функции. Простым примером таких функций могут быть функции, которые возвращают списки:

```
a -> [b]
```

Если раньше наши функции могли возвращать произвольное значение `b`, то теперь мы знаем, что все результирующие значения таких функций будут списками.

При этом для каждого такого `m` мы попытаемся построить свой замкнутый мир специальных функций `a -> m b`. Он будет жить внутри вселенной всех произвольных функций типа `a -> b`. В этом нам поможет специальный класс типов, который называется категорией Клейсли (эта конструкция носит имя математика Хенрика Клейсли).

```
class Kleisli m where
  idK  :: a -> m a
  (*>) :: (a -> m b) -> (b -> m c) -> (a -> m c)
```

Этот класс является классом `Category` в мире наших специальных функций. Если мы сотрём все буквы `m`, то мы получим обычные типы для тождества и композиции. В этом мире должны выполняться те же правила:

```
f *> idK == f
idK *> f  == f

f *> (g *> h) == (f *> g) *> h
```

Взаимодействие с внешним миром

С помощью класса `Kleisli` мы можем составлять из одних специальных функций другие. Но как мы сможем комбинировать специальные функции с обычными?

Поскольку слева у нашей специальной функции обычный общий тип, то с этой стороны мы можем воспользоваться обычной функцией композиции `>>`. Но как быть при композиции справа? Нам нужна функция типа:

```
(a -> m b) -> (b -> c) -> (a -> m c)
```

Оказывается мы можем составить её из методов класса `Kleisli`. Мы назовём эту функцию композиции `(+>)`.

```
(+>) :: Kleisli m => (a -> m b) -> (b -> c) -> (a -> m c)
f +> g = f *> (g >> idK)
```

С помощью метода `idK` мы можем погрузить в мир специальных функций любую обычную функцию.

Три композиции

У нас появилось много композиций целых три:

аргументы			результат	
обычная	>>	обычная	==	обычная
специальная	+>	обычная	==	специальная
специальная	*>	специальная	==	специальная

При этом важно понимать, что по смыслу это три одинаковые функции. Они обозначают операцию последовательного применения функций. Разные значки отражают разные типы функций аргументов. Определим модуль `Kleisli.hs`

```
module Kleisli where

import Prelude hiding (id, (>>))

class Category cat where
  id    :: cat a a
  (>>) :: cat a b -> cat b c -> cat a c

class Kleisli m where
  idK    :: a -> m a
  (*>) :: (a -> m b) -> (b -> m c) -> (a -> m c)

  (+>) :: Kleisli m => (a -> m b) -> (b -> c) -> (a -> m c)
  f +> g = f *> (g >> idK)

-- Экземпляр для функций

instance Category (->) where
  id    = \x -> x
  f >> g = \x -> g (f x)
```

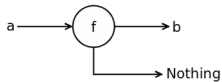
Мы не будем импортировать функцию `id`, а определим её в классе `Category`. Также в `Prelude` уже определена функция `(>>)` мы спрячем её с помощью специальной директивы `hiding` для того, чтобы она нам не мешалась. Далее мы будем дополнять этот модуль экземплярами класса `Kleisli` и примерами.

Примеры специальных функций

Частично определённые функции

Частично определённые функции – это такие функции, которые определены не для всех значений аргументов. Примером такой функции может быть функция поиска предыдущего числа для натуральных чисел. Поскольку числа натуральные, то для нуля такого числа нет. Для описания этого поведения мы можем воспользоваться специальным типом `Maybe`. Посмотрим на его определение:

```
data Maybe a = Nothing | Just a
  deriving (Show, Eq, Ord)
```



Частично определённая функция

Частично определённая функция имеет тип `a -> Maybe b`, если всё в порядке и значение было вычислено, она вернёт (`Just a`), а в случае ошибки будет возвращено значение `Nothing`. Теперь мы можем определить нашу функцию так:

```
pred :: Nat -> Maybe Nat
pred Zero      = Nothing
pred (Succ a)  = Just a
```

Для `Zero` предыдущий элемент не определён .

Составляем функции вручную

Значение функции `pred` завернуто в упаковку `Maybe`, и для того чтобы воспользоваться им нам придётся разворачивать его каждый раз. Как будет выглядеть функция извлечения дважды предыдущего натурального числа:

```
pred2 :: Nat -> Maybe Nat
pred2 x =
  case pred x of
    Just (Succ a) -> Just a
    _              -> Nothing
```

Если мы захотим определить `pred3`, мы заменим `pred` в `case`-выражении на `pred2`. Не такое уж и длинное решение, но всё же мы теряем все преимущества гибких функций, все преимущества бесточечного стиля. Нам бы хотелось написать так:

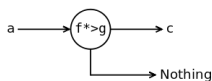
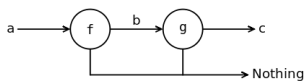
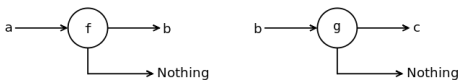
```
pred2 :: Nat -> Maybe Nat
pred2 = pred >> pred
```

```
pred3 :: Nat -> Maybe Nat
pred3 = pred >> pred >> pred
```

Но компилятор этого не допустит.

Композиция

Для того чтобы понять как устроена композиция частично определённых функций изобразим её вычисление графически. Сверху изображены две частично определённых функции. Если функция `f` вернула значение, то оно подставляется в следующую частично определённую функцию. Если же первая функция не смогла вычислить результат и вернула `Nothing`, то считается что вся функция $(f*>g)$ вернула `Nothing`.



Теперь давайте закодируем это определение в Haskell. При этом мы воспользуемся нашим классом `Kleisli`. Аналогом функции `id` для частично определённых функций будет функция, которая просто заворачивает значение в конструктор `Just`.

```
instance Kleisli Maybe where
    idK    = Just
    f *> g = \a -> case f a of
                        Nothing -> Nothing
                        Just b  -> g b
```

Смотрите, в `case`-выражении мы возвращаем `Nothing`, если функция `f` вернула `Nothing`, а если ей удалось вычислить значение и она вернула `(Just b)` мы передаём это значение в следующую функцию, то есть составляем выражение `(g b)`.

Сохраним это определение в модуле `Kleisli`, а также определение для функции `pred` и загрузим модуль в интерпретатор. Перед этим нам придётся добавить в список функций, которые мы не хотим импортировать из `Prelude` функцию `pred`, она также уже определена в `Prelude`. Для определения нашей функции нам потребуется модуль `Nat`, который мы уже определили. Скопируем файл `Nat.hs` в ту же директорию, в которой содержится файл `Kleisli.hs` и подключим этот модуль. Шапка модуля примет вид:

```
module Kleisli where

import Prelude hiding(id, (>>), pred)
import Nat
```

Добавим определение экземпляра `Kleisli` для `Maybe` в модуль `Kleisli` а также определение функции `pred`. Сохраним обновлённый модуль и загрузим в интерпретатор.

```
*Kleisli> :load Kleisli
[1 of 2] Compiling Nat                ( Nat.hs, interpreted )
[2 of 2] Compiling Kleisli             ( Kleisli.hs, interpreted )
Ok, modules loaded: Kleisli, Nat.
*Kleisli> let pred2 = pred *> pred
*Kleisli> let pred3 = pred *> pred *> pred
*Kleisli> let two   = Succ (Succ Zero)
*Kleisli>
*Kleisli> pred two
Just (Succ Zero)
*Kleisli> pred3 two
Nothing
```

Обратите внимание на то, как легко определяются производные функции. Желаемое поведение для частично определённых функций закодировано в функции `(*>)` теперь нам не нужно заворачивать значения и разворачивать их из типа `Maybe`.

Приведём пример функции, которая составлена из частично определённой функции и обычной. Определим функцию `beside`, которая вычисляет соседей для данного числа Пеано.

```

*Kleisli> let beside = pred +> \a -> (a, a + 2)
*Kleisli> beside Zero
Nothing
*Kleisli> beside two
Just (Succ Zero,Succ (Succ (Succ Zero)))
*Kleisli> (pred *> beside) two
Just (Zero,Succ (Succ Zero))

```

В выражении

```
pred +> \a -> (a, a + 2)
```

Мы сначала вычисляем предыдущее число, и если оно есть составляем пару из `\a -> (a, a+2)`, в пару попадёт данное число и число, следующее за ним через одно. Поскольку сначала мы вычислили предыдущее число в итоговом кортеже окажется предыдущее число и следующее.

Итак с помощью функций из класса `Kleisli` мы можем составлять частично определённые функции в бесточечном стиле. Обратите внимание на то, что все функции кроме `pred` были составлены в интерпретаторе.

Отметим, что в `Prelude` определена специальная функция `maybe`, которая похожа на функцию `foldr` для списков, она заменяет в значении типа `Maybe` конструкторы на функции. Посмотрим на её определение:

```

maybe      :: b -> (a -> b) -> Maybe a -> b
maybe n f Nothing = n
maybe n f (Just x) = f x

```

С помощью этой функции мы можем переписать определение экземпляра `Kleisli` так:

```

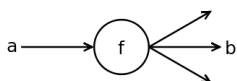
instance Kleisli Maybe where
    idM      = Just
    f *> g    = f >> maybe Nothing g

```

Многозначные функции

Многозначные функции ветрены и непостоянны. Для некоторых значений аргументов они возвращают одно значение, для иных десять, а для третьих и вовсе ничего. В Haskell такие функции

имеют тип `a -> [b]`. Функция возвращает список ответов. На рисунке изображена схема многозначной функции.



Многозначная функция

Приведём пример. Системы Линденмайера (или L-системы) моделируют развитие живого организма. Считается, что организм состоит из последовательности букв (или клеток). В каждый момент времени одна буква заменяется на новую последовательность букв, согласно определённым правилам. Так организм живёт и развивается. Приведём пример:

Аксиомы

`$a \rightarrow ab$`

`$b \rightarrow a$`

Вывод

`a`

`ab`

`aba`

`$abaab$`

`$abaababab$`

У нас есть два правила размножения клеток-букв в организме. На каждом этапе мы во всём слове заменяем букву `a` на слово `ab` и букву `b` на `a`. Начав с одной буквы `a`, мы за несколько шагов пришли к более сложному слову.

Опишем этот процесс в Haskell. Для этого определим правила развития организма в виде многозначной функции:

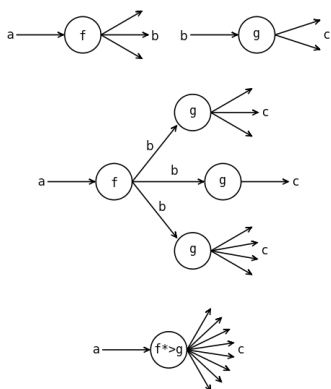
```
next :: Char -> String
```

```
next 'a' = "ab"
```

```
next 'b' = "a"
```

Напомню, что строки в Haskell являются списками символов. Теперь нам нужно применить многозначную функцию к выходу многозначной функции. Для этого мы воспользуемся классом `Kleisli`.

Композиция



Композиция многозначных функций

Определим экземпляр класса `Kleisli` для списков. На рисунке изображена схема композиции в случае многозначных функций. После применения первой функции `f` мы применяем функцию `g` к каждому элементу списка, который был получен из `f`. Так у нас получится список списков. Но нам нужен список, для этого мы после применения `g` объединяем все значения в один большой список. Отметим, что функции `f` и `g` в зависимости от значений могут возвращать разное число значений, поэтому на выходе у функций `g` разное число стрелок.

Закодируем эту схему в Haskell:

```
instance Kleisli [] where
    idK      = \a -> [a]
    f *> g   = f >> map g >> concat
```

Функция тождества принимает одно значение и погружает его в список. В композиции мы сначала применяем `f`, затем к каждому элементу списка результата применяем `g`, так у нас получается список списков. После чего мы сворачиваем его в один список с помощью функции `concat`.

Вспомним тип функций `map` и `concat`:

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

С помощью композиции мы можем получить `n`-тое поколение так:

```
generate :: Int -> (a -> [a]) -> (a -> [a])
generate 0 f = idK
generate n f = f *> generate (n - 1) f
```

Или мы можем воспользоваться функцией `iterate` и написать это определение так:

```
generate :: Int -> (a -> [a]) -> (a -> [a])
generate n f = iterate (*> f) idK !! n
```

Функция `iterate` принимает функцию вычисления следующего элемента и начальное значение и строит бесконечный список итераций:

```
iterate :: (a -> a) -> a -> [a]
iterate f a = [a, f a, f (f a), f (f (f a)), ...]
```

Если мы подставим наши аргументы то мы получим список:

```
[id, f, f*>f, f*>f*>f, f*>f*>f*>f, ...]
```

Проверим как работает эта функция в интерпретаторе. Для этого мы сначала дополним наш модуль `Kleisli` определением экземпляра для списков и функциями `next` и `generate`:

```
*Kleisli> :reload
[2 of 2] Compiling Kleisli          ( Kleisli.hs, interpreted )
Ok, modules loaded: Kleisli, Nat.
*Kleisli> let gen n = generate n next 'a'
*Kleisli> gen 0
"a"
*Kleisli> gen 1
"ab"
*Kleisli> gen 2
"aba"
*Kleisli> gen 3
"abaab"
*Kleisli> gen 4
"abaababa"
```

Правила L-системы задаются многозначной функцией. Функция `generate` позволяет по такой функции строить произвольное поколение развития буквенного организма.

Применение функций

Давайте определим в терминах композиции ещё одну полезную функцию. А именно функцию применения. Вспомним её тип:

```
($) :: (a -> b) -> a -> b
```

Эту функцию можно определить через композицию, если у нас есть в наличии постоянная функция и единичный тип. Мы будем считать, что константа это функция из единичного типа в значение. Превратив константу в функцию мы можем составить композицию:

```
($) :: (a -> b) -> a -> b  
f $ a = (const a >> f) ()
```

В самом конце мы подставляем специальное значение `()`. Это значение единичного типа (unit type) или кортежа с нулём элементов. Единичный тип имеет всего одно значение, которым мы и воспользовались в этом определении. Зачем такое запутанное определение, вместо привычного `(f a)`? Оказывается точно таким же способом мы можем определить применение в нашем мире специальных функций `a -> m b`.

Применение в этом мире происходит особенным образом. Необходимо помнить о том, что второй аргумент функции применения, значение, которое мы подставляем в функцию, также было получено из какой-то другой функции. Поэтому оно будет иметь такую же форму, что и значения справа от стрелки. В нашем случае это `m b`.

Посмотрим на типы специальных функций применения:

```
(* $) :: (a -> m b) -> m a -> m b  
(+ $) :: (a -> b) -> m a -> m b
```

Функция `* $` применяет специальную функцию к специальному значению, а функция `+ $` применяет обычную функцию к специальному значению. Определения выглядят также как и в случае обычной функции применения, мы только меняем знаки для композиции:

```
f $ a = (const a >> f) ()  
f * $ a = (const a *> f) ()  
f + $ a = (const a +> f) ()
```

Теперь мы можем не только нанизывать специальные функции друг на друга но и применять их к значениям. Добавим эти определения в модуль `Kleisli` и посмотрим как происходит применение в интерпретаторе. Одна тонкость заключается в том, что мы

определяли применение в терминах класса `Kleisli`, поэтому правильно было написать типы новых функций так:

```
infixr 0 +$, *$
```

```
(*$) :: Kleisli m => (a -> m b) -> m a -> m b
(+)$ :: Kleisli m => (a -> b) -> m a -> m b
```

Также мы определили приоритет выполнения операций.

Загрузим в интерпретатор:

```
*Kleisli> let three = Succ (Succ (Succ Zero))
*Kleisli> pred *$ pred *$ idK three
Just (Succ Zero)
*Kleisli> pred *$ pred *$ idK Zero
Nothing
```

Обратите внимание на то как мы погружаем в мир специальных функций обычное значение с помощью функции `idK`.

Вычислим третье поколение L-системы:

```
*Kleisli> next *$ next *$ next *$ idK 'a'
"abaab"
```

Мы можем использовать и другие функции на списках:

```
*Kleisli> next *$ tail $ next *$ reverse $ next *$ idK 'a'
"aba"
```

Применение функций многих переменных

С помощью функции `+$` мы можем применять к специальным значениям обычные функции одного аргумента. А что если нам захочется применить функцию двух аргументов?

Например если мы захотим сложить два частично определённых числа:

```
?? (+) (Just 2) (Just 2)
```

На месте `??` должна стоять функция типа:

```
?? :: (a -> b -> c) -> m a -> m b -> m c
```

Оказывается с помощью методов класса `Kleisli` мы можем определить такую функцию для любой обычной функции, а не только для функции двух аргументов. Мы будем называть такие функции словом `liftN`,

где **N** – число, указывающее на арность функции. Функция (`liftN f`) “поднимает” (от англ. `lift`) обычную функцию `f` в мир специальных функций.

Функция `lift1` у нас уже есть, это просто функция `+$`. Теперь давайте определим функцию `lift2`:

```
lift2 :: Kleisli m => (a -> b -> c) -> m a -> m b -> m c
lift2 f a b = ...
```

Поскольку функция двух аргументов на самом деле является функцией одного аргумента мы можем применить первый аргумент с помощью функции `lift1`, посмотрим что у нас получится:

```
lift1      :: (a' -> b') -> m' a' -> m' b'
f          :: (a -> b -> c)
a          :: m a
```

```
lift1 f a  :: m (b -> c)  -- m' == m, a' == a, b' == b -> c
```

Теперь в нашем определении для `lift2` появится новое слагаемое `g`:

```
lift2 :: Kleisli m => (a -> b -> c) -> m a -> m b -> m c
lift2 f a b = ...
    where g = lift1 f a
```

Один аргумент мы применили, осталось применить второй. Нам нужно составить выражение `(g b)`, но для этого нам нужна функция типа:

```
m (b -> c) -> m b -> m c
```

Эта функция применяет к специальному значению функцию, которая завернута в тип `m`. Посмотрим на определение этой функции, мы назовём её `$$`:

```
($$) :: Kleisli m => m (a -> b) -> m a -> m b
mf $$ ma = ( +$ ma) *$ mf
```

Вы можете убедиться в том, что это определение проходит проверку типов. Посмотрим как эта функция работает в интерпретаторе на примере частично определённых и многозначных функций, для этого давайте добавим в модуль `Kleisli` это определение и загрузим его в интерпретатор:

```

*Kleisli> :reload Kleisli
Ok, modules loaded: Kleisli, Nat.
*Kleisli> Just (+2) $$ Just 2
Just 4
*Kleisli> Nothing $$ Just 2
Nothing
*Kleisli> [(+1), (+2), (+3)] $$ [10,20,30]
[11,21,31,12,22,32,13,23,33]
*Kleisli> [(+1), (+2), (+3)] $$ []
[]

```

Обратите внимание на то, что в случае списков были составлены все возможные комбинации применений. Мы применили первую функцию из списка ко всем аргументам, потом вторую функцию, третью и объединили все результаты в список.

Теперь мы можем закончить наше определение для lift2:

```

lift2 :: Kleisli m => (a -> b -> c) -> m a -> m b -> m c
lift2 f a b = f' $$ b
  where f' = lift1 f a

```

Мы можем записать это определение более кратко:

```

lift2 :: Kleisli m => (a -> b -> c) -> m a -> m b -> m c
lift2 f a b = lift1 f a $$ b

```

Теперь давайте добавим это определение в модуль Kleisli и посмотрим в интерпретаторе как работает эта функция:

```

*Kleisli> :reload
[2 of 2] Compiling Kleisli          ( Kleisli.hs, interpreted )
Ok, modules loaded: Kleisli, Nat.
*Kleisli> lift2 (+) (Just 2) (Just 2)
Just 4
*Kleisli> lift2 (+) (Just 2) Nothing
Nothing

```

Как на счёт функций трёх и более аргументов? У нас уже есть функции lift1 и lift2 определим функцию lift3:

```

lift3 :: Kleisli m => (a -> b -> c -> d) -> m a -> m b -> m c -> m d
lift3 f a b c = ...

```

Первые два аргумента мы можем применить с помощью функции lift2. Посмотрим на тип получившегося выражения:

```
lift2      :: Kleisli m => (a' -> b' -> c') -> m a' -> m b' -> m c'
f         :: a -> b -> c -> d
```

```
lift2 f a b :: m (c -> d)  -- a' == a, b' == b, c' == c -> d
```

У нас опять появился тип `m (c -> d)` и к нему нам нужно применить значение `m c`, чтобы получить `m d`. Этим как раз и занимается функция `$$`. Итак итоговое определение примет вид:

```
lift3 :: Kleisli m => (a -> b -> c -> d) -> m a -> m b -> m c -> m d
lift3 f a b c = lift2 f a b $$ c
```

Так мы можем определить любую функцию `liftN` через функции `liftN-1` и `$$`.

Несколько полезных функций

Теперь мы умеем применять к специальным значениям произвольные обычные функции. Определим ещё несколько полезных функций. Первая функция принимает список специальных значений и собирает их в специальный список:

```
import Prelude hiding (id, (>>), pred, sequence)
```

```
sequence :: Kleisli m => [m a] -> m [a]
sequence = foldr (lift2 (:)) (idK [])
```

Мы “спрячем” из `Prelude` одноимённую функцию `sequence`. Посмотрим на примеры:

```
*Kleisli> sequence [Just 1, Just 2, Just 3]
Just [1,2,3]
*Kleisli> sequence [Just 1, Nothing, Just 3]
Nothing
```

Во второй команде вся функция вернула `Nothing` потому что при объединении списка встретилось значение `Nothing`, это равносильно тому, что мы объединяем в один список, значения полученные из функций, которые могут не вычислить результат. Поскольку значение одного из элементов не определено, весь список не определён.

Посмотрим как работает эта функция на списках:

```
*Kleisli> sequence [[1,2,3], [11,22]]
[[1,11],[1,22],[2,11],[2,22],[3,11],[3,22]]
```

Она составляет список всех комбинаций элементов из всех подсписков.

С помощью этой функции мы можем определить функцию `mapK`. Эта функция является аналогом обычной функции `map`, но она применяет специальную функцию к списку значений.

```
mapK :: Kleisli m => (a -> m b) -> [a] -> m [b]
mapK f = sequence . map f
```

Функторы и монады

В этой главе мы выписали вручную все определения для класса `Kleisli`. Мы сделали это потому, что на самом деле в арсенале стандартных средств Haskell такого класса нет. Класс `Kleisli` строит замкнутый мир специальных функций `a -> m b`. Его цель построить язык в языке и сделать программирование со специальными функциями таким же удобным как и с обычными функциями. Мы пользовались классом `Kleisli` исключительно в целях облегчения понимания этого мира. Впрочем никто не мешает нам определить этот класс и пользоваться им в наших программах.

А пока посмотрим, что есть в Haskell и как это соотносится с тем, что мы уже увидели. С помощью класса `Kleisli` мы научились делать три различных операции применения:

Применение:

- обычных функций одного аргумента к специальным значениям (функция `+$`).
- обычных функций произвольного числа аргументов к специальным значениям (функции `+$` и `$$`)
- специальных функций к специальным значениям (функция `*$`).

В Haskell для решения этих задач предназначены три отдельных класса. Это функторы, аппликативные функторы и монады.

Функторы

Посмотрим на определение класса `Functor`:

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

Тип метода `fmap` совпадает с типом для функции `+$`:

```
(+$) :: Kleisli m => (a -> b) -> m a -> m b
```

Нам только нужно заменить `m` на `f` и зависимость от `Kleisli` на зависимость от `Functor`:

Итак в Haskell у нас есть базовая операция `fmap` применения обычной функции к значению из мира специальных функций. В модуле `Control.Applicative` определён инфиксный синоним `<$>` для этой функции.

Аппликативные функторы

Посмотрим на определение класса `Applicative`:

```
class Functor f => Applicative f where
    pure    :: a -> f a
    (<*>)   :: f (a -> b) -> f a -> f b
```

Если присмотреться к типам методов этого класса, то мы заметим, что это наши старые знакомые `idK` и `$$`. Если для данного типа `f` определён экземпляр класса `Applicative`, то из контекста следует, что для него также определён и экземпляр класса `Functor`.

Значит у нас есть функции `fmap` (или `lift1`) и `<*>` (или `$$`). С их помощью мы можем составить функции `liftN`, которые поднимают обычные функции произвольного числа аргументов в мир специальных значений.

Класс `Applicative` определён в модуле `Control.Applicative`, там же мы сможем найти и функции `liftA`, `liftA2`, `liftA3` и символьный синоним `<$>` для функции `fmap`. Функции `liftAn` определены так:

```
liftA2 f a b    = f <$> a <*> b
liftA3 f a b c  = f <$> a <*> b <*> c
```

Видно что эти определения с точностью до обозначений совпадают с теми, что мы уже писали для класса `Kleisli`.

Монады

Посмотрим на определение класса `Monad`

```
class Monad m where
  return :: a -> m a
  (>>=) :: m a -> (a -> m b) -> m b
```

Присмотримся к типам методов этого класса:

```
return :: a -> m a
```

Их типа видно, что это ни что иное как функция `idK`. В классе `Monad` у неё точно такой же смысл. Теперь функция `>>=`, она читается как функция *связывания* (`bind`).

```
(>>=) :: m a -> (a -> m b) -> m b
```

Так возможно совпадение не заметно, но давайте “перевернём” эту функцию:

```
(=<<) :: Monad m => (a -> m b) -> m a -> m b
(=<<) = flip (>>=)
```

Поменяв аргументы местами, мы получили знакомую функцию `*$`. Итак функция связывания это функция применения специальной функции к специальному значению. У неё как раз такой смысл.

В `Prelude` определены экземпляры класса `Monad` для типов `Maybe` и `[]`.

Они определены по такому же принципу, что и наши определения для `Kleisli` только не для композиции, а для применения.

Отметим, что в модуле `Control.Monad` определены функции `sequence` и `mapM`, они несут тот же смысл, что и функции `sequence` и `mapK`, которые мы определяли для класса `Kleisli`.

Свойства классов

Посмотрим на свойства функторов и аппликативных функторов.

Свойства класса Functor

```
fmap id x          == x          -- тождество
fmap f . fmap g    == fmap (f . g) -- композиция
```

Первое свойство говорит о том, что если мы применяем `fmap` к функции тождества, то мы должны снова получить функцию тождества, или по другому можно сказать, что применение функции тождества к специальному значению не изменяет это значение. Второе свойство говорит о том, что последовательное применение к специальному значению двух обычных функций можно записать в виде применения композиции двух обычных функций к специальному значению.

Если всё это звучит туманно, попробуем переписать эти свойства в терминах композиции:

```
mf +> id          == mf
(mf +> g) +> h     == mf +> (g >> h)
```

Первое свойство говорит о том, что тождественная функция не изменяет значение при композиции. Второе свойство указывает на ассоциативность композиции одной специальной функции `mf` и двух обычных функций `g` и `h`.

Свойства класса Applicative

Свойства класса **Applicative**, для наглядности они сформулированы не через методы класса, а через производные функции.

```
fmap f x          == liftA f x          -- связь с Functor

liftA id x         == x                 -- тождество
liftA3 (.) f g x   == f <*> (g <*> x)    -- композиция
liftA f (pure x)   == pure (f x)       -- гомоморфизм
```

Первое свойство говорит о том, что применение специальной функции одного аргумента совпадает с методом `fmap` из класса **Functor**. Свойство тождества идентично аналогичному свойству для класса **Functor**.

Свойство композиции сформулировано хитро, но давайте посмотрим на типы аргументов:

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

```
liftA3 (.) f g x :: m c
```

```
g <*> x           :: m b
f (g <*> x)        :: m c
```

Слева в свойстве стоит `liftA3`, а не `liftA2`, потому что мы сначала применяем композицию `(.)` к двум функциям `f` и `g`, а затем применяем составную функцию к значению `x`.

Последнее свойство говорит о том, что если мы возьмём обычную функцию и обычное значение и поднимем их в мир специальных значений с помощью `lift` и `pure`, то это тоже самое если бы мы просто применили бы функцию `f` к значению в мире обычных значений и затем подняли бы результат в мир специальных значений.

Полное определение классов

На самом деле я немного схитрил. Я рассказал вам только об основных методах классов `Applicative` и `Monad`. Но они содержат ещё несколько дополнительных методов, которые выражаются через остальные. Посмотрим на них, начнём с класса `Applicative`.

```
class Functor f => Applicative f where
  -- | Поднимаем значение в мир специальных значений.
  pure :: a -> f a

  -- | Применение специального значения-функции.
  (<*>) :: f (a -> b) -> f a -> f b

  -- | Константная функция. Отбрасываем первое значение.
  (*>) :: f a -> f b -> f b
  (*>) = liftA2 (const id)

  -- | Константная функция, Отбрасываем второе значение.
  (<*) :: f a -> f b -> f a
  (<*) = liftA2 const
```

Два новых метода (`*`) и (`<*`) имеют смысл константных функций. Первая функция игнорирует значение слева, а вторая функция игнорирует значение справа. Посмотрим как они работают в интерпретаторе:

```
Prelude Control.Applicative> Just 2 *> Just 3
Just 3
Prelude Control.Applicative> Nothing *> Just 3
Nothing
Prelude Control.Applicative> (const id) Nothing Just 3
Just 3
Prelude Control.Applicative> [1,2] <* [1,2,3]
[1,1,1,2,2,2]
```

Значение игнорируется, но способ комбинирования специальных функций учитывается. Так во втором выражении не смотря на то, что мы не учитываем конкретное значение `Nothing`, мы учитываем, что если один из аргументов частично определённой функции не определён, то не определено всё значение. Сравните с результатом выполнения следующего выражения.

По той же причине в последнем выражении мы получили три копии первого списка. Так произошло потому, что второй список содержал три элемента. К каждому из элементов была применена функция `const x`, где `x` пробегает по элементам списка слева от (`<*`).

Аналогичный метод есть и в классе `Monad`:

```
class Monad m where
    return :: a -> m a
    (>=)    :: m a -> (a -> m b) -> m b

    (>>)    :: m a -> m b -> m b
    fail    :: String -> m a

    m >> k  = m >= const k
    fail s  = error s
```

Функция `>>` в классе `Monad`, которую мы прятали из-за символа композиции, является аналогом постоянной функции в классе `Monad`. Она работает так же как и `*`. Функция `fail` используется для служебных нужд Haskell при выводе ошибок. Поэтому мы её здесь не рассматриваем. Для определения экземпляра класса `Monad` достаточно определить методы `return` и `>=`.

Исторические замечания

Напрашивается вопрос. Зачем нам функции `return` и `pure` или `*>` и `>>`? Если вы заглянете в документацию к модулю `Control.Monad`, то там вы найдёте функции `liftM`, `liftM2`, `liftM3`, которые выполняют те же операции, что и аналогичные функции из модуля `Control.Applicative`.

Стандартные библиотеки устроены так, потому что класс `Applicative` появился гораздо позже класса `Monad`. И к появлению этого нового класса уже накопилось огромное число библиотек, которые рассчитаны на прежние имена. Но в будущем возможно прежние классы будут заменены на такие классы:

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b

class Pointed f where
    pure :: a -> f a

class (Functor f, Pointed f) => Applicative f where
    (<*>) :: f (a -> b) -> f a -> f b

    (*>)  :: f a -> f b -> f b
    (<*)  :: f a -> f b -> f a

class Applicative f => Monad f where
    (>>=) :: f a -> (a -> f b) -> f b
```

Краткое содержание

В этой главе мы долгой обходной дорогой шли к понятию монады и функтора. Эти классы служат для облегчения работы в мире специальных функций вида `a -> m b`, в категории Клейсли

С помощью класса `Functor` можно применять специальные значения к обычным функциям одного аргумента:

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

С помощью класса `Applicative` можно применять специальные значения к обычным функциям любого числа аргументов:

```

class Functor f => Applicative f where
    pure      :: a -> f a
    <*>       :: f (a -> b) -> f a -> f b

liftA  :: Applicative f => (a -> b) -> f a -> f b
liftA2 :: Applicative f => (a -> b -> c) -> f a -> f b -> f c
liftA3 :: Applicative f => (a -> b -> c -> d) -> f a -> f b -> f c -> f d
...

```

С помощью класса **Monad** можно применять специальные значения к специальным функциям.

```

class Monad m where
    return :: a -> m a
    (>>=)  :: m a -> (a -> m b) -> m b

```

Функция `return` является функцией `id` в мире специальных функций, а функция `>>=` является функцией применения (`$`), с обратным порядком следования аргументов. Вспомним также класс **Kleisli**, на примере котором мы узнали много нового из жизни специальных функций:

```

class Kleisli m where
    idK      :: a -> m a
    (*>)     :: (a -> m b) -> (b -> m c) -> (a -> m c)

```

Мы узнали несколько стандартных специальных функций:

Частично определённые функции

```

a -> Maybe b
data Maybe a = Nothing | Just a

```

Многозначные функции

```

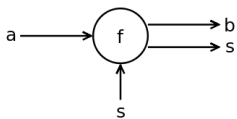
a -> [b]
data [a] = [] | a : [a]

```

Упражнения

В первых упражнениях вам предлагается по картинке специальной функции написать экземпляр классов **Kleisli** и **Monad**.

Функции с состоянием



Функция с состоянием

В Haskell нельзя изменять значения. Новые сложные значения описываются в терминах базовых значений. Но как же тогда мы сможем описать функцию с состоянием? Функцию, которая принимает на вход значение, составляет результат на основе внутреннего состояния и значения аргумента и обновляет состояние. Поскольку мы не можем изменять состояние единственное, что нам остаётся – это принимать значение состояния на вход вместе с аргументом и возвращать обновлённое состояние на выходе. У нас получится такой тип:

```
a -> s -> (b, s)
```

Функция принимает одно значение типа `a` и состояние типа `s`, а возвращает пару, которая состоит из результата типа `b` и обновлённого состояния. Если мы введём синоним:

```
type State s b = s -> (b, s)
```

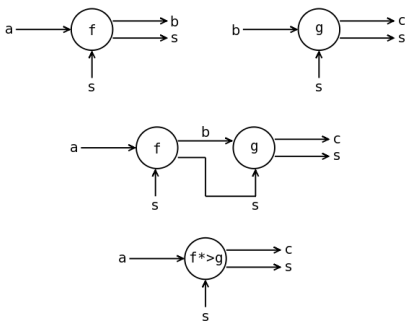
И вспомним о частичном применении, то мы сможем записать тип функции с состоянием так:

```
a -> State s b
```

В Haskell пошли дальше и выделили для таких функций специальный тип:

```
data State s a = State (s -> (a, s))
```

```
runState :: State s a -> s -> (a, s)  
runState (State f) = f
```

Композиция функций с состоянием

Функция `runState` просто извлекает функцию из оболочки `State`.

На рисунке изображена схема функции с состоянием. В сравнении с обычной функцией у такой функции один дополнительный выход и один дополнительный вход типа `s`. По ним течёт и изменяется состояние.

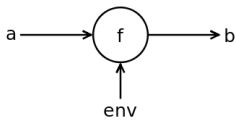
Попробуйте по схеме композиции для функций с состоянием написать экземпляры для классов `Kleisli` и `Monad` для типа `State s`.

Подсказка: В этом определении есть одна хитрость, в отличие от типов `Maybe` и `[a]` у типа `State` два параметра, это параметр состояния и параметр значения. Но мы делаем экземпляр не для `State`, а для `State s`, то есть мы свяжем тип с некоторым произвольным типом `s`.

```
instance Kleisli (State s) where
    ...
```

Функции с окружением

Сначала мы рассмотрим функции с окружением. Функции с окружением – это такие функции, у которых есть некоторое хранилище данных или окружение, из которых они могут читать информацию. Но в отличие от функций с состоянием они не могут это окружение изменять. Функция с окружением похожа на функцию с состоянием без одного выхода для состояния.



Функция с окружением

Функция с окружением принимает аргумент `a` и окружение `env` и возвращает результат `b`:

```
a -> env -> b
```

Как и в случае функций с состоянием выделим для функции с окружением отдельный тип. В Haskell он называется **Reader** (от англ. чтец). Все функции с окружением имеют возможность читать из общего хранилища данных. Например они могут иметь доступ на чтение к общей базе данных.

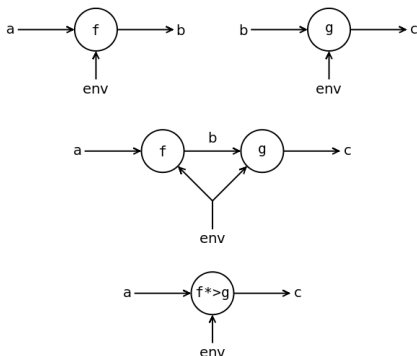
```
data Reader env b = Reader (env -> b)
```

```
runReader :: Reader env b -> (env -> b)
runReader (Reader f) = f
```

Теперь функция с окружением примет вид:

```
a -> Reader env b
```

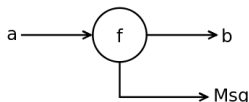
Определите для функций с окружением экземпляр класса **Kleisli**. У нас возникнет цепочка функций, каждая из которых будет нуждаться в значении окружения. Поскольку окружение общее для всех функций мы всем функциям передадим одно и то же значение.



Функция с окружением

Функции-накопители

Функции-накопители при вычислении за ширмой накапливают некоторое значение. Функция-накопитель похожа на функцию с состоянием но без стрелки, по которой состояние подаётся в функцию. Функция-накопитель имеет тип: $a \rightarrow (b, \text{msg})$



Функция-накопитель

Выделим результат функции в отдельный тип с именем `Writer`.

```
data Writer msg b = Writer (b, msg)
```

```
runWriter :: Writer msg b -> (b, msg)
runWriter (Writer a) = a
```

Тип функции примет вид:

```
a -> Writer msg b
```

Значения типа `msg` мы будем называть сообщениями. Смысл функций $a \rightarrow \text{Writer msg b}$ заключается в том, что при вычислении они накапливают в значении `msg` какую-нибудь информацию. Это могут быть отладочные сообщения. Или база данных, которая открыта для всех функций на запись.

Класс Monoid

Как мы будем накапливать результат? Пока мы умеем лишь возвращать из функции пару значений. Одно из них нам нужно передать в следующую функцию, а что делать с другим?

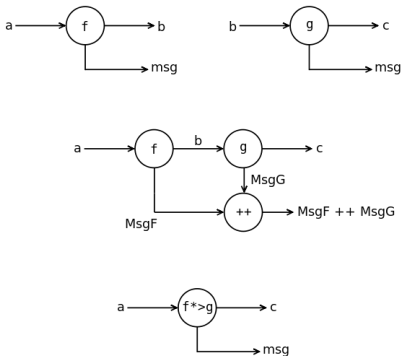
На помощь нам придёт класс `Monoid`, он определён в модуле `Data.Monoid`:

```
class Monoid a where
  mempty  :: a
  mappend :: a -> a -> a
```

В этом классе определено пустое значение `mempty` и бинарная функция соединения двух значений в одно. Этот класс очень похож на класс `Category` и `Kleisli`. Там тоже было значение, которое ничего не делает и операция составления нового значения из двух простейших значений. Даже свойства класса похожи:

```
mempty `mappend` f      = f
f      `mappend` mempty  = f
```

```
f `mappend` (g `mappend` h) = (f `mappend` g) `mappend` h
```



Композиция функций-накопителей

Первые два свойства говорят о том, что значение `mempty` и вправду является пустым элементом относительно операции `mappend`. А третье свойство говорит о том, что порядок при объединении элементов не важен.

Посмотрим на определение экземпляра для списков:

```
instance Monoid [a] where
    mempty  = []
    mappend = (++)
```

Итак пустой элемент это пустой список, а объединение это операция конкатенации списков. Проверим в интерпретаторе:

```
*Kleisli> :m Data.Monoid
Prelude Data.Monoid> [1 .. 4] `mappend` [4, 3 .. 1]
[1,2,3,4,4,3,2,1]
Prelude Data.Monoid> "Hello" `mappend` " World" `mappend` mempty
"Hello World"
```

Напишите экземпляр класса `Kleisli` для функций накопителей по рисунку. При этом будем считать, что тип `msg` является экземпляром класса `Monoid`.

Экземпляры для функторов и монад

Представьте, что у нас нет класса `Kleisli`, а есть лишь `Functor`, `Applicative` и `Monad`. Напишите экземпляры для этих классов для всех рассмотренных в этой главе специальных функций (в том числе и для `Reader` и `Writer`). Экземпляры `Functor` и `Applicative` могут быть определены через `Monad`. Но для тренировки определите экземпляры полностью. Сначала `Functor`, затем `Applicative` и в последнюю очередь `Monad`.

Деревья

Напишите экземпляры классов `Kleisli` и `Monad` для двух типов, которые описывают деревья. Бинарные деревья:

```
data BTree a = BList a | BNode a (BTree a) (BTree a)
```

Деревья с несколькими узлами:

```
data Tree a = Node a [Tree a]
```

Считайте, что списки являются частными случаями деревьев. В этом смысле деревья будут описывать многозначные функции, которые возвращают несколько значений, организованных в иерархическую структуру.

Стандартные функции

Почитайте документацию к модулям `Control.Monad` и `Control.Applicative`. Присмотритесь к функциям, попробуйте применить их в интерпретаторе.

Эквивалентность классов `Kleisli` и `Monad`

Покажите, что классы `Kleisli` и `Monad` эквивалентны. Для этого нужно для произвольного типа `m` с одним параметром `m` определить два экземпляра:

```
instance Kleisli m => Monad    m where
instance Monad    m => Kleisli m where
```

Нужно определить экземпляр одного класса с помощью методов другого.

Свойства класса Monad

Если класс `Monad` эквивалентен `Kleisli`, то в нём должны выполняться точно такие же свойства. Запишите свойства класса `Kleisli` через методы класса `Monad`

Функторы и монады: примеры

В этой главе мы закрепим на примерах то, что мы узнали о монадах и функторах. Напомню, что с помощью монад и функторов мы можем комбинировать специальные функции вида $(a \rightarrow m\ b)$ с другими специальными функциями.

У нас есть функции тождества (`pure`, `return`) и применения (`fmap`, `=<<`):

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b

class Functor f => Applicative f where
  pure   :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b

class Monad m where
  return :: a -> m a
  (>=)   :: m a -> (a -> m b) -> m b

(=<<) :: (a -> m b) -> m a -> m b
(=<<) = flip (>=)
```

Вспомним основные производные функции для этих классов:

Или в терминах класса `Kleisli`:

```
-- Композиция
(>=>) :: Monad m => (a -> m b) -> (b -> m c) -> (a -> m c)
(<=<) :: Monad m => (b -> m c) -> (a -> m b) -> (a -> m c)

-- Константные функции
(*>) :: Applicative f => f a -> f b -> f b
(<*) :: Applicative f => f a -> f b -> f a

-- Применение обычных функций к специальным значениям
(<$>) :: Functor f => (a -> b) -> f a -> f b

liftA  :: Applicative f => (a -> b)          -> f a -> f b
liftA2 :: Applicative f => (a -> b -> c)      -> f a -> f b -> f c
liftA3 :: Applicative f => (a -> b -> c -> d) -> f a -> f b -> f c -> f d

-- Преобразование элементов списка специальной функцией
mapM   :: Monad m => (a -> m b) -> [a] -> m [b]
```

Нам понадобится модуль с определениями типов и экземпляров монад для всех типов, которые мы рассмотрели в предыдущей главе.

Экземпляры для `[]` и `Maybe` уже определены в `Prelude`, а типы `State`, `Reader` и `Writer` можно найти в библиотеках `mtl` и `transformers`. Пока мы не знаем как устанавливать библиотеки, определим эти типы и экземпляры для `Monad` самостоятельно. Возможно, вы уже определили их, выполняя одно из упражнений предыдущей главы, если это так, то сейчас вы можете сверить ответы. Определим модуль `Types`:

```
module Types(
  State(..), Reader(..), Writer(..),
  runState, runWriter, runReader,
  module Control.Applicative,
  module Control.Monad,
  module Data.Monoid)
where

import Data.Monoid
import Control.Applicative
import Control.Monad

-----
-- Функции с состоянием
--
--      a -> State s b

data State s a = State (s -> (a, s))
runState :: State s a -> s -> (a, s)
runState (State f) = f

instance Monad (State s) where
  return a = State $ \s -> (a, s)
  ma >>= mf = State $ \s0 ->
    let (b, s1) = runState ma s0
    in runState (mf b) s1

-----
-- Функции с окружением
--
--      a -> Reader env b

data Reader env a = Reader (env -> a)

runReader :: Reader env a -> env -> a
runReader (Reader f) = f

instance Monad (Reader env) where
  return a = Reader $ const a
  ma >>= mf = Reader $ \env ->
    let b = runReader ma env
    in runReader (mf b) env
```



```

-----
-- Функции-накопители
--
--      Monoid msg => a -> Writer msg b

data Writer msg a = Writer (a, msg)
    deriving (Show)

runWriter :: Writer msg a -> (a, msg)
runWriter (Writer f) = f

instance Monoid msg => Monad (Writer msg) where
    return a    = Writer (a, mempty)
    ma >>= mf   = Writer (c, msgA `mappend` msgF)
        where (b, msgA) = runWriter ma
              (c, msgF) = runWriter $ mf b

```

Я пропустил определения для экземпляров классов `Functor` и `Applicative`, их можно получить из экземпляра для класса `Monad` с помощью стандартных функций `liftM`, `return` и `ap` из модуля `Control.Monad`.

Нам встретилась новая запись в экспорте модуля. Для удобства мы экспортируем модули `Control.Applicative`, `Control.Monad` и `Data.Monoid` целиком. Для этого мы написали ключевое слово `module` перед экспортируемым модулем. Теперь если мы в каком-нибудь другом модуле импортируем модуль `Types` нам станут доступными все функции из этих модулей.

Случайные числа

С помощью монады `State` можно имитировать случайные числа. Мы будем генерировать случайные числа из интервала от 0 до 1 с помощью алгоритма:

```

nextRandom :: Double -> Double
nextRandom = snd . properFraction . (105.947 * )

```

Функция `properFraction` возвращает пару, которая состоит из целой части и остатка числа. Взяв второй элемент пары с помощью `snd`, мы выделяем остаток. Функция `nextRandom` представляет собой генератор случайных чисел, который принимает значение с предыдущего шага и строит по нему следующее значение.

Построим тип для случайных чисел:

```
type Random a = State Double a

next :: Random Double
next = State $ \s -> (s, nextRandom s)
```

Теперь определим функцию, которая прибавляет к данному числу случайное число из интервала от 0 до 1:

```
addRandom :: Double -> Random Double
addRandom x = fmap (+x) next
```

Посмотрим как эта функция работает в интерпретаторе:

```
*Random> runState (addRandom 5) 0.5
(5.5,0.9735000000000014)
*Random> runState (addRandom 5) 0.7
(5.7,0.16289999999999338)
*Random> runState (mapM addRandom [1..5]) 0.5
([1.5,2.9735000000000014,3.139404500000154,4.769488561516319,
 5.5250046269694195],0.6226652135290891)
```

В последней строчке мы с помощью функции `mapM` прибавили ко всем элементам списка разные случайные числа, обновление счётчика происходило за кадром, с помощью функции `mapM` и экземпляра `Monad` для `State`.

Также мы можем определить функцию, которая складывает два случайных числа, одно из интервала $[-1+a, 1+a]$, а другое из интервала $[-2+b, 2+b]$:

```
addRandom2 :: Double -> Double -> Random Double
addRandom2 a b = liftA2 add next next
  where add a b = \x y -> diapr a 1 x + diapr b 1 y
        diapr c r = \x -> x * 2 * r - r + c
```

Функция `diapr` перемещает интервал от 0 до 1 в интервал от $c-r$ до $c+r$. Обратите внимание на то как мы сначала составили обычную функцию `add`, которая перемещает значения из интервала от 0 до 1 в нужный диапазон и складывает. И только в самый последний момент мы применили к этой функции случайные значения. Посмотрим как работает эта функция:

```
*Random> runState (addRandom2 0 10) 0.5
(10.947000000000003,0.13940450000015403)
*Random> runState (addRandom2 0 10) 0.7
```

```
(9.725799999999987,0.2587662999992979)
```

Прибавим два списка и получим сумму:

```
*Random> let res = fmap sum $ zipWithM addRandom2 [1..3] [11 .. 13]
*Random> runState res 0.5
(43.060125804029965,0.969511377766409)
*Random> runState res 0.7
(39.86034841613788,0.26599261421101517)
```

Функция `zipWithM` является аналогом функции `zipWith`. Она устроена также как и функция `mapM`, сначала применяется обычная функция `zipWith`, а затем функция `sequence`.

С помощью типа `Random` мы можем определить функцию подбрасывания монетки:

```
data Coin = Heads | Tails
    deriving (Show)

dropCoin :: Random Coin
dropCoin = fmap drop' next
    where drop' x
        | x < 0.5    = Heads
        | otherwise = Tails
```

У монетки две стороны орёл (`Heads`) и решка (`Tails`). Поскольку шансы на выпадение той или иной стороны равны, мы для определения стороны разделяем интервал от 0 до 1 в равных пропорциях.

Подбросим монетку пять раз:

```
*Random> let res = sequence $ replicate 5 dropCoin
```

Функция `replicate n a` составляет список из `n` повторяющихся элементов `a`. Посмотрим что у нас получилось:

```
*Random> runState res 0.4
([Heads,Heads,Heads,Heads,Tails],0.5184926967068364)
*Random> runState res 0.5
([Tails,Tails,Heads,Tails,Tails],0.6226652135290891)
```

Конечные автоматы

С помощью монады `State` можно описывать конечные автоматы (finite-state machine). Конечный автомат находится в каком-то начальном

состоянии. Он принимает на вход ленту событий. Одно событие происходит за другим. На каждое событие автомат реагирует переходом из одного состояния в другое.

```
type FSM s = State s s
```

```
fsm :: (ev -> s -> s) -> (ev -> FSM s)
fsm transition = \e -> State $ \s -> (s, transition e s)
```

Функция `fsm` принимает функцию переходов состояний `transition` и возвращает функцию, которая принимает состояние и возвращает конечный автомат. В качестве значения конечный автомат `FSM` будет возвращать текущее состояние.

С помощью конечных автоматов можно описывать различные устройства. Лентой событий будет ввод пользователя (нажатие на кнопки, включение/выключение питания).

Приведём простой пример. Рассмотрим колонки, у них есть розетка, кнопка вкл/выкл и регулятор громкости. Возможные состояния:

```
type Speaker = (SpeakerState, Level)
```

```
data SpeakerState = Sleep | Work
    deriving (Show)
```

```
data Level = Level Int
    deriving (Show)
```

Тип колонок складывается из двух значений: состояния и уровня громкости. Колонки могут быть выключенными (`Sleep`) или работать на определённой громкости (`Work`). Считаем, что максимальный уровень громкости составляет 10 единиц, а минимальный ноль единиц. Границы диапазона громкости описываются такими функциями:

```
quieter :: Level -> Level
quieter (Level n) = Level $ max 0 (n-1)
```

```
louder :: Level -> Level
louder (Level n) = Level $ min 10 (n+1)
```

Мы будем обновлять значения уровня громкости не напрямую, а с помощью вспомогательных функций `louder` и `quieter`. Так мы не сможем выйти за пределы заданного диапазона.

Возможные события:

```
data User = Button | Quieter | Louder
  deriving (Show)
```

Пользователь может либо нажать на кнопку вкл/выкл или повернуть реле громкости влево, чтобы приглушить звук (**Quieter**) или вправо, чтобы сделать погромче (**Louder**). Будем считать, что колонки всегда включены в розетку.

Составим функцию переходов:

```
speaker :: User -> FSM Speaker
speaker = fsm $ trans
  where trans Button    (Sleep, n) = (Work, n)
        trans Button    (Work, n) = (Sleep, n)
        trans Louder     (s, n)    = (s, louder n)
        trans Quieter    (s, n)    = (s, quieter n)
```

Мы считаем, что при выключении колонок реле остаётся некотором положении, так что при следующем включении они будут работать на той же громкости. Реле можно крутить и в состоянии **Sleep**. Посмотрим на типичную сессию работы колонок:

```
*FSM> let res = mapM speaker [Button, Louder, Quieter, Quieter, Button]
```

Сначала мы включаем колонки, затем прибавляем громкость, затем дважды делаем тише и в конце выключаем. Посмотрим что получилось:

```
*FSM> runState res (Sleep, Level 2)
([(Sleep,Level 2),(Work,Level 2),(Work,Level 3),(Work,Level 2),
 (Work,Level 1)],(Sleep,Level 1))
*FSM> runState res (Sleep, Level 0)
([(Sleep,Level 0),(Work,Level 0),(Work,Level 1),(Work,Level 0),
 (Work,Level 0)],(Sleep,Level 0))
```

Смотрите, изменив начальное значение, мы изменили весь список значений. Обратите внимание на то, что во втором прогоне мы не ушли в минус по громкости, не смотря на то, что пытались крутить реле за установленный предел.

Определим колонки другого типа. Наши новые колонки будут безопаснее предыдущих. Представьте ситуацию, что мы выключили колонки на высоком уровне громкости. Мы слушали домашнюю запись с низким уровнем звука. Мы выключили и забыли. Потом мы решили

послушать другую мелодию, которая записана с нормальным уровнем звука. При включении колонок нас оглушил шквал звука. Чтобы этого избежать мы решили воспользоваться другими колонками.

Колонки при выключении будут выставлять уровень громкости на ноль и реле можно будет крутить только если колонки включены.

```
safeSpeaker :: User -> FSM Speaker
safeSpeaker = fsm $ trans
  where trans Button (Sleep, _) = (Work, Level 0)
        trans Button (Work, _) = (Sleep, Level 0)
        trans Quieter (Work, n) = (Work, quieter n)
        trans Louder (Work, n) = (Work, louder n)
        trans _ (Sleep, n) = (Sleep, n)
```

При нажатии на кнопку вкл/выкл уровень громкости выводится в положение 0. Колонки реагируют на запросы изменения уровня громкости только в состоянии **Work**. Посмотрим как работают наши новые колонки:

```
*FSM> let res = mapM safeSpeaker [Button, Louder, Quieter, Button, Louder]
```

Мы включаем колонки, делаем по-громче, затем по-тише, затем выключаем и пытаемся изменить громкость после выключения. Посмотрим как они сработают, представим, что мы выключили колонки на уровне громкости 10:

```
*FSM> runState res (Sleep, Level 10)
([(Sleep,Level 10),(Work,Level 0),(Work,Level 1),(Work,Level 0),
 (Sleep,Level 0)],(Sleep,Level 0))
```

Первое значение в списке является стартовым состоянием, которое мы задали. После этого колонки включаются и мы видим, что уровень громкости переключился на ноль. Затем мы увеличиваем громкость, сбавляем её и выключаем. Попытка изменить громкость выключенных колонок не проходит. Это видно по последнему элементу списка и итоговому состоянию колонок, которое находится во втором элементе пары.

Предположим, что колонки работают с самого начала, тогда первым действием мы выключаем их. Посмотрим, что случится дальше:

```
*FSM> runState res (Work, Level 10)
([(Work,Level 10),(Sleep,Level 0),(Sleep,Level 0),(Sleep,Level 0),
 (Work,Level 0)],(Work,Level 1))
```

Дальше мы пытаемся изменить громкость но у нас ничего не выходит.

Отложенное вычисление выражений

В этом примере мы будем выполнять арифметические операции на целых числах. Мы будем их складывать, вычитать и умножать. Но вместо того, чтобы сразу вычислять выражения мы будем составлять их описание. Мы будем кодировать операции конструкторами.

```
data Exp      = Var String
              | Lit Int
              | Neg Exp
              | Add Exp Exp
              | Mul Exp Exp
              deriving (Show, Eq)
```

У нас есть тип `Exp`, который может быть либо переменной `Var` с данным строчным именем, либо целочисленной константой `Lit`, либо одной из трёх операций: вычитанием (`Neg`), сложением (`Add`) или умножением (`Mul`).

Такие типы называют *абстрактными синтаксическими деревьями* (abstract syntax tree, AST). Они содержат описание выражений. Теперь вместо того чтобы сразу проводить вычисления мы будем собирать выражения в значении типа `Exp`. Сделаем экземпляр для `Num`:

```
instance Num Exp where
    negate = Neg
    (+)    = Add
    (*)    = Mul

    fromInteger = Lit . fromInteger

    abs      = undefined
    signum   = undefined
```

Также определим вспомогательные функции для обозначения переменных:

```
var :: String -> Exp
var = Var

n :: Int -> Exp
n = var . show
```

Функция `var` составляет переменную с данным именем, а функция `n` составляет переменную, у которой имя является целым числом. Сохраним эти определения в модуле `Exp`. Теперь у нас всё готово для составления выражений:

```
*Exp> n 1
Var "1"
*Exp> n 1 + 2
Add (Var "1") (Lit 2)
*Exp> 3 * (n 1 + 2)
Mul (Lit 3) (Add (Var "1") (Lit 2))
*Exp> - n 2 * 3 * (n 1 + 2)
Neg (Mul (Mul (Var "2") (Lit 3)) (Add (Var "1") (Lit 2)))
```

Теперь давайте создадим функцию для вычисления таких выражений. Она будет принимать выражение и возвращать целое число.

```
eval :: Exp -> Int
eval (Lit n)      = n
eval (Neg n)      = negate $ eval n
eval (Add a b)    = eval a + eval b
eval (Mul a b)    = eval a * eval b
eval (Var name)  = ???
```

Как быть с конструктором `Var`? Нам нужно откуда-то узнать какое значение связано с переменной. Функция `eval` должна также принимать набор значений для всех переменных, которые используются в выражении. Этот набор значений мы будем называть окружением.

Обратите внимание на то, что в каждом составном конструкторе мы рекурсивно вызываем функцию `eval`, мы словно обходим всё дерево выражения. Спускаемся вниз, до самых листьев в которых расположены либо значения (`Lit`), либо переменные (`Var`). Нам было бы удобно иметь возможность пользоваться окружением из любого узла дерева. В этом нам поможет тип `Reader`.

Представим что у нас есть значение типа `Env` и функция, которая позволяет читать значения переменных по имени:

```
value :: Env -> String -> Int
```

Теперь определим функцию `eval`:


```
eval :: Exp -> Reader Env Int
eval (Lit n)      = pure n
eval (Neg n)      = liftA negate $ eval n
eval (Add a b)    = liftA2 (+) (eval a) (eval b)
eval (Mul a b)    = liftA2 (*) (eval a) (eval b)
eval (Var name) = Reader $ \env -> value env name
```

Определение сильно изменилось, оно стало не таким наглядным.

Теперь значение `eval` стало специальным, поэтому при рекурсивном вызове функции `eval` нам приходится поднимать в мир специальных функций обычные функции вычитания, сложения и умножения. Мы можем записать это выражение

немного по другому:

```
eval :: Exp -> Reader Env Int
eval (Lit n)      = pure n
eval (Neg n)      = negateA $ eval n
eval (Add a b)    = eval a `addA` eval b
eval (Mul a b)    = eval a `mulA` eval b
eval (Var name) = Reader $ \env -> value env name
```

```
addA      = liftA2 (+)
mulA      = liftA2 (*)
negateA   = liftA negate
```

Тип Map

Для того чтобы закончить определение функции `eval` нам нужно определить тип `Env` и функцию `value`. Для этого мы воспользуемся типом `Map`, он предназначен для хранения значений по ключу.

Этот тип живёт в стандартном модуле `Data.Map`. Посмотрим на его описание:

```
data Map k a = ..
```

Первый параметр типа `k` это ключ, а второй это значение. Мы можем создать значение типа `Map` из списка пар ключ значение с помощью функции `fromList`.

Посмотрим на основные функции:

<code>-- Создаём значения типа Map</code>	<code>-- создаём</code>
<code>empty :: Map k a</code>	<code>-- пустой Map</code>
<code>fromList :: Ord k => [(k, a)] -> Map k a</code>	<code>-- по списку (ключ, значение)</code>

```
-- Узнаём значение по ключу
(!)      :: Ord k => Map k a -> k -> a
lookup   :: Ord k => k -> Map k a -> Maybe a

-- Добавляем элементы
insert   :: Ord k => k -> a -> Map k a -> Map k a

-- Удаляем элементы
delete  :: Ord k => k -> Map k a -> Map k a
```

Обратите внимание на ограничение `Ord k` в этих функциях, ключ должен быть экземпляром класса `Ord`. Посмотрим как эти функции работают:

```
*Exp> :m +Data.Map
*Exp Data.Map> :m -Exp
Data.Map> let v = fromList [(1, "Hello"), (2, "Bye")]
Data.Map> v ! 1
"Hello"
Data.Map> v ! 3
"*** Exception: Map.find: element not in the map
Data.Map> lookup 3 v
Nothing
Data.Map> let v1 = insert 3 "Yo" v
Data.Map> v1 ! 3
"Yo"
```

Функция `lookup` является стабильным аналогом функции `!`. В том смысле, что она определена с помощью `Maybe`. Она не приведёт к падению программы, если для данного ключа не найдётся значение.

Теперь мы можем определить функцию `value`:

```
import qualified Data.Map as M(Map, lookup, fromList)

...

type Env = M.Map String Int

value :: Env -> String -> Int
value env name = maybe errorMsg $ M.lookup env name
  where errorMsg = error $ "value is undefined for " ++ name
```

Обычно функции из модуля `Data.Map` включаются с директивой `qualified`, поскольку имена многих функций из этого модуля совпадают с именами из модуля `Prelude`. Теперь все определения из модуля `Data.Map` пишутся с приставкой `M..`

Создадим вспомогательную функцию, которая упростит вычисление выражений:

```
runExp :: Exp -> [(String, Int)] -> Int
runExp a env = runReader (eval a) $ M.fromList env
```

Сохраним определение новых функций в модуле `Exp`. И посмотрим что у нас получилось:

```
*Exp> let env a b = [("1", a), ("2", b)]
*Exp> let exp = 2 * (n 1 + n 2) - n 1
*Exp> runExp exp (env 1 2)
5
*Exp> runExp exp (env 10 5)
20
```

Так мы можем пользоваться функциями с окружением для того, чтобы читать значения из общего источника. Впрочем мы можем просто передавать окружение дополнительным аргументом и не пользоваться монадами:

```
eval :: Env -> Exp -> Int
eval env x = case x of
  Lit n      -> n
  Neg n      -> negate $ eval' n
  Add a b    -> eval' a + eval' b
  Mul a b    -> eval' a * eval' b
  Var name   -> value env name
  where eval' = eval env
```

Накопление результата

Рассмотрим по-подробнее тип `Writer`. Он выполняет задачу обратную к типу `Reader`. Когда мы пользовались типом `Reader`, мы могли в любом месте функции извлекать данные из окружения. Теперь же мы будем не извлекать данные из окружения, а записывать их.

Рассмотрим такую задачу нам нужно обойти дерево типа `Exp` и подсчитать все бинарные операции. Мы прибавляем к накопителю результата единицу за каждый конструктор `Add` или `Mul`. Тип сообщений будет числом. Нам нужно сделать экземпляра класса `Monoid` для чисел.

Напомню, что тип накопителя должен быть экземпляром класса `Monoid`:

```
class Monoid a where
  mempty  :: a
  mappend :: a -> a -> a

  mconcat :: [a] -> a
  mconcat = foldr mappend mempty
```

Но для чисел возможно несколько вариантов, которые удовлетворяют свойствам. Для сложения:

```
instance Num a => Monoid a where
  mempty  = 0
  mappend = (+)
```

И умножения:

```
instance Num a => Monoid a where
  mempty  = 1
  mappend = (*)
```

Для нашей задачи подойдёт первый вариант, но не исключена возможность того, что для другой задачи нам понадобится второй. Но тогда мы уже не сможем определить такой экземпляр. Для решения этой проблемы в модуле `Data.Monoid` определено два типа обёртки:

```
newtype Sum a = Sum { getSum  :: a }
newtype Prod a = Prod { getProd :: a }
```

В этом определении есть два новых элемента. Первый это ключевое слово `newtype`, а второй это фигурные скобки. Что всё это значит?

Тип-обёртка newtype

Ключевое слово `newtype` вводит новый тип-обёртку. Тип-обёртка может иметь только один конструктор, у которого лишь один аргумент. Запись:

```
newtype Sum a = Sum a
```

Это тоже самое, что и

```
data Sum a = Sum a
```

Единственное отличие заключается в том, что в случае `newtype` вычислитель не видит разницы между `Sum a` и `a`. Её видит лишь компилятор. Это означает, что на разворачивание и заворачивание

такого значения в тип обёртку не тратится никаких усилий. Такие типы подходят для решения двух задач:

- Более точная проверка типов.

Например у нас есть типы, которые описывают физические величины, все они являются числами, но у них также есть и размерности. Мы можем написать:

```
type Velocity = Double
type Time     = Double
type Length   = Double
```

```
velocity :: Length -> Time -> Velocity
velocity leng time = leng / time
```

В этом случае мы спокойно можем подставить на место времени путь и наоборот. Но с помощью типов обёрток мы можем исключить эти случаи:

```
newtype Velocity = Velocity Double
newtype Time     = Time     Double
newtype Length   = Length   Double
```

```
velocity :: Length -> Time -> Velocity
velocity (Length leng) (Time time) = Velocity $ leng / time
```

В этом случае мы проводим проверку по размерностям, компилятор не допустит смешивания данных.

- Определение нескольких экземпляров одного класса для одного типа. Этот случай мы как раз и рассматриваем для класса `Monoid`. Нам нужно сделать два экземпляра для одного и того же типа `Num a => a`.

Сделаем две обёртки!

```
newtype Sum a = Sum a
newtype Prod a = Prod a
```

Тогда мы можем определить два экземпляра для двух разных типов:

Один для `Sum`:

```
instance Num a => Monoid (Sum a) where
    mempty  = Sum 0
    mappend (Sum a) (Sum b) = Sum (a + b)
```

А другой для `Prod`:

```
instance Num a => Monoid (Prod a) where
    mempty  = Prod 1
    mappend (Prod a) (Prod b) = Prod (a * b)
```

Записи

Вторая новинка заключалась в фигурных скобках. С помощью фигурных скобок в Haskell обозначаются *записи* (records). Запись это произведение типа, но с выделенными именами для полей.

Например мы можем сделать тип для описания паспорта:

```
data Passport = Person {
    surname      :: String,      -- Фамилия
    givenName    :: String,      -- Имя
    nationality   :: String,      -- Национальность
    dateOfBirth  :: Date,        -- Дата рождения
    sex          :: Bool,        -- Пол
    placeOfBirth :: String,      -- Место рождения
    authority     :: String,      -- Место выдачи документа
    dateOfIssue   :: Date,        -- Дата выдачи
    dateOfExpiry  :: Date,        -- Дата окончания срока
    } deriving (Eq, Show)        -- действия

data Date = Date {
    day    :: Int,
    month  :: Int,
    year   :: Int
    } deriving (Show, Eq)
```

В фигурных скобках через запятую мы указываем поля. Поле состоит из имени и типа. Теперь нам доступны две операции:

- Чтение полей

```
hello :: Passport -> String
hello p = "Hello, " ++ givenName p ++ "!"
```

Для чтения мы просто подставляем в имя поля данное значение. В этой функции мы приветствуем человека и обращаемся к нему по имени. Для того, чтобы узнать его имя мы подсмотрели в паспорт, в поле `givenName`.

- Обновление полей. Для обновления полей мы пользуемся таким синтаксисом:

```
value { fieldName1 = newValue1, fieldName2 = newValue2, ... }
```

Мы присваиваем в значении `value` полю с именем `fieldName` новое значение `newFieldValue`. К примеру продлим срок действия паспорта на десять лет:

```
prolongate :: Passport -> Passport
prolongate p = p{ dateOfExpiry = newDate }
  where newDate = oldDate { year = year oldDate + 10 }
        oldDate = dateOfExpiry p
```

Вернёмся к типам `Sum` и `Prod`:

```
newtype Sum a = Sum { getSum :: a }
newtype Prod a = Prod { getProd :: a }
```

Этой записью мы определили два типа-обёртки. У нас есть две функции, которые заворачивают обычное значение, это `Sum` и `Prod`. С помощью записей мы тут же в определении типа определили функции которые разворачивают значения, это `getSum` и `getProd`.

Вспомним определение для типа `State`:

```
data State s a = State (s -> (a, s))

runState :: State s a -> (s -> (a, s))
runState (State f) = f
```

Было бы гораздо лучше определить его так:

```
newtype State s a = State{ runState :: s -> (a, s) }
```

Накопление чисел

Но вернёмся к нашей задаче. Мы будем накапливать сумму в значении типа `Sum`. Поскольку нас интересует лишь значение накопителя, наша функция будет возвращать значение единичного типа `()`.

```
countBiFuns :: Exp -> Int
countBiFuns = getSum . execWriter . countBiFuns'
```

```

countBiFuns' :: Exp -> Writer (Sum Int) ()
countBiFuns' x = case x of
  Add a b -> tell (Sum 1) *> bi a b
  Mul a b -> tell (Sum 1) *> bi a b
  Neg a    -> un a
  _        -> pure ()
  where bi a b = countBiFuns' a *> countBiFuns' b
        un     = countBiFuns'

tell :: Monoid a => a -> Writer a ()
tell a = Writer ((), a)

execWriter :: Writer msg a -> msg
execWriter (Writer (a, msg)) = msg

```

Первая функция `countBiFuns` извлекает значение из типов `Writer` и `Sum`. А вторая функция `countBiFuns'` вычисляет значение.

Мы определили две вспомогательные функции `tell`, которая записывает сообщение в накопитель и `execWriter`, которая возвращает лишь сообщение. Это стандартные для `Writer` функции.

Посмотрим как работает эта функция:

```

*Exp> countBiFuns (n 2)
0
*Exp> countBiFuns (n 2 + n 1 + 2 + 3)
3

```

Накопление логических значений

В модуле `Data.Monoid` определены два типа для накопления логических значений. Это типы `All` и `Any`. С помощью типа `All` мы можем проверить выполняется ли некоторое свойство для всех значений. А с помощью типа `Any` мы можем узнать, что существует хотя бы один элемент, для которых это свойство выполнено.

Посмотрим на определение экземпляров класса `Monoid` для этих типов:

```

newtype All = All { getAll :: Bool }

instance Monoid All where
  mempty = All True
  All x `mappend` All y = All (x && y)

```


В типе `All` мы накапливаем значения с помощью логического “и”. Нейтральным элементом является конструктор `True`. Итоговое значение накопителя будет равно `True` только в том случае, если все накапливаемые сообщения были равны `True`.

В типе `Any` всё наоборот:

```
instance Monoid Any where
  mempty = Any False
  Any x `mappend` Any y = Any (x || y)
```

Посмотрим как работают эти типы. Составим функцию, которая проверяет отсутствие оператора минус в выражении:

```
noNeg :: Exp -> Bool
noNeg = not . getAny . execWriter . anyNeg
anyNeg :: Exp -> Writer Any ()
anyNeg x = case x of
  Neg _    -> tell (Any True)
  Add a b  -> bi a b
  Mul a b  -> bi a b
  _        -> pure ()
  where bi a b = anyNeg a *> anyNeg b
```

Функция `anyNeg` проверяет есть ли в выражении хотя бы один конструктор `Neg`. В функции `noNeg` мы извлекаем результат и берём его отрицание, чтобы убедиться в том что в выражении не встретилось ни одного конструктора `Neg`.

```
*Exp> noNeg (n 2 + n 1 + 2 + 3)
True
*Exp> noNeg (n 2 - n 1 + 2 + 3)
False
```

Накопление списков

Экземпляр класса `Monoid` определён и для списков. Предположим у нас есть дерево, в каждом узле которого находятся числа, давайте соберём все числа больше 5, но меньше 10. Деревья мы возьмём из модуля `Data.Tree`:

```
data Tree a = Node
  { rootLabel :: a           -- значение метки
  , subForest :: Forest a    -- ноль или несколько дочерних деревьев
  }

type Forest a = [Tree a]
```

Интересный тип. Тип `Tree` определён через `Forest`, а `Forest` определён через `Tree`. По этому типу мы видим, что каждый узел содержит некоторое значение типа `a`, и список дочерних деревьев.

Составим дерево:

```
*Exp> :m Data.Tree
Prelude Data.Tree> let t a = Node a []
Prelude Data.Tree> let list a = Node a []
Prelude Data.Tree> let bi v a b = Node v [a, b]
Prelude Data.Tree> let un v a = Node v [a]
Prelude Data.Tree>
Prelude Data.Tree> let tree1 = bi 10 (un 2 $ un 6 $ list 7) (list 5)
Prelude Data.Tree> let tree2 = bi 12 tree1 (bi 8 tree1 tree1)
```

Теперь составим функцию, которая будет обходить дерево, и собирать числа из заданного диапазона:

```
type Diap a = (a, a)
```

```
inDiap :: Ord a => Diap a -> Tree a -> [a]
inDiap d = execWriter . inDiap' d
```

```
inDiap' :: Ord a => Diap a -> Tree a -> Writer [a] ()
inDiap' d (Node v xs) = pick d v *> mapM_ (inDiap' d) xs
  where pick (a, b) v
        | (a <= v) && (v <= b) = tell [v]
        | otherwise           = pure ()
```

Как и раньше у нас две функции, одна выполняет вычисления, другая извлекает результат из `Writer`. В функции `pick` мы проверяем число на принадлежность интервалу, если это так мы добавляем число к результату, а если нет пропускаем его, добавляя нейтральный элемент (в функции `pure`). Обратите внимание на то как мы обрабатываем список дочерних поддеревьев. Функция `mapM_` является аналогом функции `mapM`, Она используется, если результат функции не важен, а важны те действия, которые происходят при преобразовании списка. В нашем случае это накопление результата. Посмотрим на определение этой функции:

```
mapM_ :: Monad m => (a -> m b) -> [a] -> m ()
mapM_ f = sequence_ . map f

sequence_ :: Monad m => [m a] -> m ()
sequence_ = foldr (>>) (return ())
```

Основное отличие состоит в функции `sequence_`. Раньше мы собирали значения в список, а теперь отбрасываем их с помощью константной функции `>>`. В конце мы возвращаем значение единичного типа `()`.

Теперь сохраним в модуле `Tree` определение функции и вспомогательные функции создания деревьев `in`, `bi`, и `list` и посмотрим как наша функция работает:

```
*Tree> inDiap (4, 10) tree2
[10,6,7,5,8,10,6,7,5,10,6,7,5]
*Tree> inDiap (5, 8) tree2
[6,7,5,8,6,7,5,6,7,5]
*Tree> inDiap (0, 3) tree2
[2,2,2]
```

Монада изменяемых значений ST

Возможно читатели, для которых “родным” является один из императивных языков, немного заскучили по изменяемым значениям. Мы говорили, что в Haskell ничего не изменяется, мы даём всё более и более сложные имена статическим значениям, а потом вычислитель редуцирует имена к настоящим значениям. Но есть алгоритмы, которые очень элегантно описываются в терминах изменяемых значений. Примером такого алгоритма может быть быстрая сортировка. Задача состоит в перестановке элементов массива так, чтобы на выходе любой последующий элемент массива был больше предыдущего (для списков эту задачу решают функции `sort` и `sortBy`).

Само по себе явление обновления значения является побочным эффектом. Оно ломает представление о статичности мира, у нас появляются фазы: до обновления и после обновления. Но представьте, что обновление происходит локально, мы постоянно меняем только одно значение, при этом за время обновления ни одна другая переменная *не может* пользоваться промежуточными значениями и обновления происходят с помощью *чистых* функций. Представьте функцию, которая принимает значение, выделяет внутри себя память, и при построении результата начинает обновлять значение внутри этой памяти (с помощью чистых функций) и считать что-то ещё полезное на основе этих обновлений, как только вычисления

закончатся, память стирается, и возвращается значение. Будет ли такая функция чистой? Интуиция подсказывает, что да. Это было доказано, но для реализации этого требуется небольшой трюк на уровне типов. Получается, что не смотря на то, что функция содержит побочные эффекты, она является чистой, поскольку все побочные эффекты локальны, они происходят только внутри вызова функции и только в самой функции.

Для симуляции обновления значения в Haskell нам нужно решить две проблемы. Как упорядочить обновление значения? И как локализовать его? В императивных языках порядок вычисления выражений строго связан с порядком следования выражений, на примитивном уровне, грубо упрощая, можно сказать, что вычислитель читает код как ленту и выполняет выражение за выражением. В Haskell всё совсем по-другому. Мы можем писать функции в любом порядке, также в любом порядке мы можем объявлять локальные переменные в **where** или **let**-выражениях. Компилятор определяет порядок редукции синонимов по функциональным зависимостям. Синоним `f` не будет раскрыт раньше синонима `g` только в том случае, если результат `g` требуется в `f`. Но с обновлением значения этот вариант не пройдет, посмотрим на выражение:

```
fun :: Int -> Int
fun arg =
    let mem = new arg
        x   = read mem
        y   = x + 1
        ??  = write mem y
        z   = read mem
    in z
```

Предполагается, что в этой функции мы получаем значение `arg`, выделяем память `mem` с помощью специальной функции `new`, которая принимает начальное значение, которое будет храниться в памяти. Затем читаем из памяти, прибавляем к значению единицу, снова записываем в память, потом опять читаем из памяти, сохранив значение в переменной `z`, и в самом конце возвращаем ответ. Налицо две проблемы: `z` не зависит от `y`, поэтому мы можем считать значение `z` в любой момент после инициализации памяти и вторая проблема: что должна возвращать функция `write`?

Для того чтобы упорядочить эти вычисления мы воспользуемся типом `State`. Каждое выражение будет принимать фиктивное состояние и возвращать его. Тогда функция `fun` запишется так:

```
fun :: Int -> State s Int
fun arg = State $ \s0 ->
  let (mem, s1)  = runState (new arg)          s0
      ((), s2)   = runState (write mem arg)     s1
      (x, s3)    = runState (read mem)          s2
      y          = x + 1
      ((), s4)   = runState (write mem y)       s3
      (z, s5)    = runState (read mem)          s4
  in (z, s5)
```

```
new      :: a -> State s (Mem a)
write    :: Mem a -> a -> State s ()
read     :: Mem a -> State s a
```

Тип `Mem` параметризован типом значения, которое хранится в памяти. В этом варианте мы не можем изменить порядок следования выражений, поскольку нам приходится передавать состояние. Мы могли бы записать это выражение гораздо короче с помощью методов класса `Monad`, но мне хотелось подчеркнуть как передача состояния навязывает порядок вычисления. Функция `write` теперь возвращает пустой кортеж. Но порядок не теряется за счёт состояния. Пустой кортеж намекает на то, что единственное назначение функции `write` – это обновление состояния.

Однако этого не достаточно. Мы хотим, чтобы обновление значения было скрыто от пользователя в *чистой* функции. Мы хотим, чтобы тип функции `fun` не содержал типа `State`. Для этого нам откуда-то нужно взять начальное значение состояния. Мы можем решить эту проблему, зафиксировав тип `s`. Пусть это будет тип `FakeState`, скрытый от пользователя.

```
module Mutable(
  Mutable, Mem, purge,
  new, read, write)
where

newtype Mutable a = Mutable (State FakeState a)

data FakeState = FakeState

purge :: Mutable a -> a
purge (Mutable a) = fst $ runState a FakeState
```

```
new      :: a -> Mutable (Mem a)
read     :: Mem a -> Mutable a
write    :: Mem a -> a -> Mutable ()
```

Мы предоставим пользователю лишь тип `Mutable` без конструктора и функцию `purge`, которая “очищает” значение от побочных эффектов и примитивные функции для работы с памятью. Также мы определим экземпляры классов типа `State` для `Mutable`, сделать это будет совсем не трудно, ведь `Mutable` – это просто обёртка. С помощью этих экземпляров пользователь сможет комбинировать вычисления, которые связаны с изменением памяти. Пока вроде всё хорошо, но обеспечиваем ли мы локальность изменения значений? Нам важно, чтобы, один раз начав работать с памятью типа `Mem`, мы не смогли бы нигде воспользоваться этой памятью после выполнения функции `purge`. Оказывается, что мы можем разрушить локальность.

Посмотрите на пример:

```
let mem = purge allocate
in  purge (read mem)
```

Мы возвращаем из функции `purge` ссылку на память и спокойно пользуемся ею в другой ветке `Mutable`-вычислений. Можно ли этого избежать? Оказывается, что можно. Причём решение весьма элегантно. Мы можем построить типы `Mem` и `Mutable` так, чтобы ссылке на память не удалось просочиться через функцию `purge`. Для этого мы вернёмся к общему типу `State` с двумя параметрами. Причём первый параметр мы прицепим и к `Mem`:

```
data      Mem      s a = ..
newtype   Mutable  s a = ..

new       :: a -> Mutable s (Mem s a)
write     :: Mem s a -> a -> Mutable s ()
read      :: Mem s a -> Mutable s a
```

Теперь при создании типы `Mem` и `Mutable` связаны общим параметром `s`. Посмотрим на тип функции `purge`

```
purge :: (forall s. Mutable s a) -> a
```

Она имеет необычный тип. Слово `forall` означает “для любых”. Это слово называют квантором всеобщности. Этим мы говорим, что

функция извлечения значения не может делать никаких предположений о типе фиктивного состояния. Как дополнительный `forall` может нам помочь? Функция `purge` забывает тип фиктивного состояния `s` из типа `Mutable`, но в случае типа `Mem`, этот параметр продолжает своё путешествие по программе в типе значения `v :: Mem s a`. По типу `v` компилятор может сказать, что существует такое `s`, для которого значение `v` имеет смысл (правильно типизировано). Но оно не любое! Функцию `purge` с трюком интересует не некоторый тип, а все возможные типы `s`, поэтому пример не пройдёт проверку типов. Компилятор будет следить за “чистотой” наших обновлений.

При таком подходе остаётся вопрос: откуда мы возьмём начальное значение, ведь теперь у нас нет типа `FakeState`? В Haskell специально для этого типа было сделано исключение. Мы возьмём его из воздуха. Это чисто фиктивный параметр, нам главное, что он скрыт от пользователя, и он нигде не может им воспользоваться. Поскольку у нас нет конструктора `Mutable` мы никогда не сможем добраться до внутренней функции типа `State` и извлечь состояние. Состояние скрыто за интерфейсом класса `Monad` и отбрасывается в функции `purge`.

Тип ST

Выше я пользовался вымышленными типами для упрощения объяснений, на самом деле в Haskell за обновление значений отвечает тип `ST` (сокращение от `state transformer`). Он живёт в модуле `Control.Monad.ST`. Из документации видно, что у него два параметра, и нет конструкторов:

```
data ST s a
```

Это наш тип `Mutable`, теперь посмотрим на тип `Mem`. Он называется `ST`-ссылкой и определён в модуле `Data.STRef` (сокращение от `ST reference`). Посмотрим на основные функции:

```
newSTRef    :: a -> ST s (STRef s a)
readSTRef   :: STRef s a -> ST s a
writeSTRef  :: STRef s a -> a -> ST s ()
```

Такие функции иногда называют *смышлёными конструкторами* (*smart constructors*) они позволяют строить значение, но скрывают от

пользователя реализацию за счёт скрытия конструкторов типа (модуль экспортирует лишь имя типа `STRef`).

Для иллюстрации этих функций реализуем одну вспомогательную функцию из модуля `Data.STRef`, функцию обновления значения по ссылке:

```
modifySTRef :: STRef s a -> (a -> a) -> ST s ()
modifySTRef ref f = writeSTRef . f =<< readSTRef ref
```

Мы воспользовались тем, что `ST` является экземпляром `Monad`. Также как и для `State` для `ST` определены экземпляры классов `Functor`, `Applicative` и `Monad`. Какое совпадение! Посмотрим на функцию `purge`:

```
runST :: (forall s. ST s a) -> a
```

Императивные циклы

Реализуем `for` цикл из языка C:

```
Result s;
```

```
for (i = 0 ; i < n; i++)
    update(i, s);
```

```
return s;
```

У нас есть стартовое значение счётчика и результата, функция обновления счётчика, предикат останова и функция обновления результата. Мы инициализируем счётчик и затем обновляем счётчик и состояние до тех пор пока предикат счётчика не станет ложным. Напишем чистую функцию, которая реализует этот процесс. В этой функции мы воспользуемся специальным синтаксическим сахаром, который называется `do`-нотация, не пугайтесь это всё ещё Haskell, для понимания этого примера загляните в раздел “сахар для монад” главы~17.

```
module Loop where
```

```
import Control.Monad
```

```
import Data.STRef
```

```
import Control.Monad.ST
```



```

forLoop :: i -> (i -> Bool) -> (i -> i) -> (i -> s -> s) -> s -> s
forLoop i0 pred next update s0 = runST $ do
  refI <- newSTRef i0
  refS <- newSTRef s0
  iter refI refS
  readSTRef refS
  where iter refI refS = do
    i <- readSTRef refI
    s <- readSTRef refS
    when (pred i) $ do
      writeSTRef refI $ next i
      writeSTRef refS $ update i s
      iter refI refS

```

Впрочем, код выше можно понять, если читать его как обычный императивный код. Выражения **do**-блока выполняются последовательно, одно за другим. Сначала мы инициализируем два изменяемых значения: для счётчика цикла и для состояния. Затем в функции `iter` мы читаем значения и выполняем проверку предиката `pred`. Функция `when` – это стандартная функция из модуля `Control.Monad`. Она проверяет предикат, и если он возвращает `True` выполняет серию действий, в которых мы записываем обновлённые значения. Обратите внимание на то, что связка `when-do` это не специальная конструкция языка. Как было сказано `when` – это просто функция, но она ожидает одно действие, а мы хотим выполнить сразу несколько. Следующее за ней **do** начинает блок действий (границы блока определяются по отступам), который будет интерпретироваться как одно действие. В настоящем императивном цикле в обновлении и предикате счётчика может участвовать переменная результата, но это считается признаком дурного стиля, поэтому наши функции определены на типе счётчика. Решим типичную задачу, посчитаем числа от одного до десяти:

```

*Loop> forLoop 1 (<=10) succ (+) 0
55

```

Посчитаем факториал:

```

*Loop> forLoop 1 (<=10) succ (*) 1
3628800
*Loop> forLoop 1 (<=100) succ (*) 1
9332621544394415268169923885626670049071596826
4381621468592963895217599993229915608941463976
1565182862536979208272237582511852109168640000
000000000000000000

```

Теперь напишем while-цикл:

```
Result s;  
  
while (pred(s))  
    update(s);  
  
return s;
```

В этом цикле участвует один предикат и одна функция обновления результата, мы обновляем результат до тех пор пока предикат не станет ложным.

```
whileLoop :: (s -> Bool) -> (s -> s) -> s -> s  
whileLoop pred update s0 = runST $ do  
    ref <- newSTRef s0  
    iter ref  
    readSTRef ref  
    where iter ref = do  
        s <- readSTRef ref  
        when (pred s) $ do  
            writeSTRef ref $ update s  
            iter ref
```

Посчитаем сумму чисел через while-цикл:

```
*Loop> whileLoop ((>0) . fst) (\(n, s) -> (pred n, n + s)) (10, 0)  
(0,55)
```

Первый элемент пары играет роль счётчика, а во втором мы накапливаем результат.

Быстрая сортировка

Реализуем императивный алгоритм быстрой сортировки. Алгоритм быстрой сортировки хорош не только тем, что он работает очень быстро, но и минимальным расходом памяти. Сортировка проводится в самом массиве, с помощью обмена элементов местами. Но для этого нам понадобятся изменяемые массивы. Этот тип определён в модуле `Data.Array.ST`. В Haskell есть несколько типов изменяемых массивов (как впрочем и неизменяемых), это связано с различными нюансами размещения элементов в массивах, о которых мы пока умолчим. Следующий класс определяет общий интерфейс к различным массивам:

```
class (HasBounds a, Monad m) => MArray a e m where
  newArray  :: Ix i => (i, i) -> e -> m (a i e)
  newArray_ :: Ix i => (i, i) -> m (a i e)
```

MArray – это сокращение от mutable (изменяемый) array. Метод `newArray` создаёт массив типа `a`, который завёрнут в тип-монаду `m`. Первый аргумент указывает на диапазон значений индексов массива, а вторым аргументом передаётся элемент, который будет записан во все ячейки массива. Вторая функция записывает в массив элемент `undefined`.

Посмотрим на вспомогательные классы:

```
class Ord a => Ix a where
  range  :: (a, a) -> [a]
  index  :: (a, a) -> a -> Int
  inRange :: (a, a) -> a -> Bool
  rangeSize :: (a, a) -> Int

class HasBounds a where
  bounds :: Ix i => a i e -> (i, i)
```

Класс **Ix** описывает тип индекса из непрерывного диапазона значений. Наверняка по имени функции и типу вы догадаетесь о назначении методов (можете свериться с интерпретатором на типах **Int** или **(Int, Int)**). Класс **HasBounds** обозначает массивы размер, которых фиксирован. Но вернёмся к массивам. Мы можем не только выделять память под массив, но и читать элементы и обновлять их:

```
readArray  :: (MArray a e m, Ix i) => a i e -> i -> m e
writeArray :: (MArray a e m, Ix i) => a i e -> i -> e -> m ()
```

В случае **ST**-ссылок у нас была функция `runST`. Она возвращала значение из памяти, но что будет возвращать аналогичная функция для массива? Посмотрим на неё:

```
freeze :: (Ix i, MArray a e m, IArray b e) => a i e -> m (b i e)
```

Возможно за всеми классами схожесть с функцией `runST` прослеживается не так чётко. Новый класс **IArray** обозначает неизменяемые (immutable) массивы. Функцией `freeze` мы превращаем изменяемый массив в неизменяемый, но завёрнутый в специальный тип-монаду. В нашем случае этим типом будет **ST**. В модуле **Data.Array.ST** определена специальная версия этой функции:

```
runSTArray :: Ix i => (forall s . ST s (STArray s i e)) -> Array i e
```

Здесь `Array` – это обычный неизменяемый массив. Он живёт в модуле `Data.Array` мы можем строить массивы из списков значений, преобразовывать их разными способами, превращать в обратно в списки и многое другое. Об о всём этом можно узнать из документации к модулю. Обратите на появление слова `forall` и в этой функции. Оно несёт тот же смысл, что и в функции `runST`.

Для тренировки напомним функцию, которая меняет местами два элемента массива:

```
module Qsort where

import Data.STRef
import Control.Monad.ST

import Data.Array
import Data.Array.ST
import Data.Array.MArray

swapElems :: Ix i => i -> i -> STArray s i e -> ST s ()
swapElems i j arr = do
    vi <- readArray arr i
    vj <- readArray arr j

    writeArray arr i vj
    writeArray arr j vi
```

Протестируем на небольшом массиве:

```
test :: Int -> Int -> [a] -> [a]
test i j xs = elems $ runSTArray $ do
    arr <- newListArray (0, length xs - 1) xs
    swapElems i j arr
    return arr
```

Тип функции `test` ничем не выдаёт её содержание. Вроде функция как функция:

```
test :: Int -> Int -> [a] -> [a]
```

Посмотрим на то, как она работает:

```
*Qsort> test 0 3 [0,1,2,3,4]
[3,1,2,0,4]
*Qsort> test 0 4 [0,1,2,3,4]
[4,1,2,3,0]
```

Теперь перейдём к сортировке. Суть метода в том, что мы выбираем один элемент массива, называемый осью (pivot) и переставляем остальные элементы массива так, чтобы все элементы меньше осевого были слева от него, а все, что больше оказались справа. Затем мы повторяем эту процедуру на массивах поменьше, тех, что находятся слева и справа от осевого элемента и так пока все элементы не отсортируются. В алгоритме очень хитрая процедура перестановки элементов, наша задача переставить элементы в массиве, то есть не пользуясь никакими дополнительными структурами данных. Я не буду говорить как это делается, просто выпишу код, а вы можете почитать об этом где-нибудь, в любом случае из кода будет понятно как это происходит:

```
qsort :: Ord a => [a] -> [a]
qsort xs = elems $ runSTArray $ do
  arr <- newListArray (left, right) xs
  qsortST left right arr
  return arr
  where left  = 0
        right = length xs - 1

qsortST :: Ord a => Int -> Int -> STArray s Int a -> ST s ()
qsortST left right arr = do
  when (left <= right) $ do
    swapArray left (div (left + right) 2) arr
    vLeft <- readArray arr left
    (last, _) <- forLoop (left + 1) (<= right) succ
                      (update vLeft) (return (left, arr))
    swapArray left last arr
    qsortST left (last - 1) arr
    qsortST (last + 1) right arr
  where update vLeft i st = do
        (last, arr) <- st
        vi <- readArray arr i
        if (vi < vLeft)
          then do
            swapArray (succ last) i arr
            return (succ last, arr)
          else do
            return (last, arr)
```

Это далеко не самый быстрый вариант быстрой сортировки, но самый простой. Мы просто учимся обращаться с изменяемыми массивами. Протестируем:

```
*Qsort> qsort "abracadabra"
"aaaaabbcdrr"
```

```
*Qsort> let x = 1000000
*Qsort> last $ qsort [x, pred x .. 0]
-- двадцать лет спустя
1000000
```

Краткое содержание

Мы посмотрели на примерах как применяются типы `State`, `Reader` и `Writer`. Также мы познакомились с монадой изменяемых значений `ST`. Она позволяет писать в императивном стиле на Haskell. Мы узнали два новых элемента построения типов:

- Типы-обёртки, которые определяются через ключевое слово `newtype`.
- Записи, они являются произведением типов с именованными полями.

Также мы узнали несколько полезных типов:

- `Map` – хранение значений по ключу (из модуля `Data.Map`).
- `Tree` – деревья (из модуля `Data.Tree`).
- `Array` – массивы (из модуля `Data.Array`).
- Типы для накопления результата (из модуля `Data.Monoid`).

Отметим, что экземпляр класса `Monad` определён и для функций. Мы можем записать функцию двух аргументов (`a -> b -> c`) как `(a -> (->) b c)`. Тогда тип `(->) b` будет типом с одним параметром, как раз то, что нужно для класса `Monad`. По смыслу экземпляр класса `Monad` для функций совпадает с экземпляром типа `Reader`. Первый аргумент стрелочного типа `b` играет роль окружения.

Упражнения

- Напишите с помощью типа `Random` функцию игры в кости, два игрока бросают по очереди кости (два кубика с шестью гранями, грани пронумерованы от 1 до 6). Они бросают кубики 10 раз выигрывает тот, у кого в сумме выпадет больше очков. Функция принимает начальное состояние и выводит результат игры: суммарные баллы игроков.

- Напишите с помощью типа `Random` функцию, которая будет создавать случайные деревья заданной глубины. Значение в узле является случайным числом от 0 до 100, также число дочерних деревьев в каждом узле случайно, оно изменяется от 0 до 10.
- Опишите в виде конечного автомата поведение амёбы. Амёба может двигаться на плоскости по четырём направлениям. Если она чувствует свет в определённой стороне, то она ползёт туда. Если по-близости нет света, она ползает в произвольном направлении. Амёба улавливает интенсивность света, если по всем четырём сторонам интенсивность одинаковая, она стоит на месте и греется.
- Казалось бы, зачем нам сохранять вычисления в выражениях, почему бы нам просто не вычислить их сразу? Если у нас есть описание выражения мы можем применить различные техники оптимизации, которые могут сокращать число вычислений. Например нам известно, что двойное отрицание не влияет на аргумент, мы можем выразить это так:

```
instance Num Exp where
    negate (Neg a)  = a
    negate x        = Neg x
    ...
    ...
```

Так мы сократили вычисления на две операции. Возможны и более сложные техники оптимизации. Мы можем учесть ноль и единицу при сложении и умножении или дистрибутивность сложения относительно умножения.

В этом упражнении вам предлагается провести подобную оптимизацию для логических значений. У нас есть абстрактное синтаксическое дерево:

```
data Log      = True
              | False
              | Not Log
              | Or  Log Log
              | And Log Log
```

Напишите функцию, которая оптимизирует выражение `Log`. Эта функция приводит `Log` к конъюнктивной нормальной форме (КНФ). Дерево в КНФ обладает такими свойствами: все узлы с `Or` находятся ближе к корню чем узлы с `And` и все узлы с `And` находятся ближе к корню чем узлы с `Not`. В КНФ выражения имеют вид:

```
(True `And` Not False `And` True) `Or` True `Or` (True `And` False)
(True `And` True `And` False) `Or` True
```

Как бы мы не шли от корня к листу сначала нам будут встречаться только операции `Or`, затем только операции `And`, затем только `Not`.

КНФ замечательна тем, что её вычисление может пройти досрочно. КНФ можно представить так:

```
data Or' a = Or' [a]
data And' a = And' [a]
data Not' a = Not' a
data Lit = True' | False'

type CNF = Or' (And' (Not' Lit))
```

Сначала идёт список выражений разделённых конструктором `Or` (вычислять весь список не нужно, нам нужно найти первый элемент, который вернёт `True`). Затем идёт список выражений, разделённых `And` (опять же его не надо вычислять целиком, нам нужно найти первое выражение, которое вернёт `False`). В самом конце стоят отрицания.

В нашем случае приведение к КНФ состоит из двух этапов:

- Сначала построим выражение, в котором все конструкторы `Or` и `And` стоят ближе к корню чем конструктор `Not`. Для этого необходимо воспользоваться такими правилами:

```
-- удаление двойного отрицания
Not (Not a) ==> a

-- правила де Моргана
Not (And a b) ==> Or (Not a) (Not b)
Not (Or a b) ==> And (Not a) (Not b)
```


- Делаем так чтобы все конструкторы **Or** были бы ближе к корню чем конструкторы **And**. Для этого мы воспользуемся правилом дистрибутивности:

```
And a (Or b c) ==> Or (And a b) (And a c)
```

При этом мы будем учитывать коммутативность **And** и **Or**:

```
And a b == And b a
```

```
Or a b == Or b a
```

- Когда вы закончите определение функции:

```
transform :: Log -> CNF
```

Напишите функцию, которая будет сравнивать вычисление исходного выражения напрямую и вычисление через КНФ. Эта функция будет принимать исходное значение типа **Log** и будет возвращать два числа, число операций необходимых для вычисления выражения:

```
evalCount :: Log -> (Int, Int)
```

```
evalCount a = (evalCountLog a, evalCountCNF a)
```

```
evalCountLog :: Log -> Int
```

```
evalCountLog a = ...
```

```
evalCountCNF :: Log -> Int
```

```
evalCountCNF a = ...
```

При написании этих функций воспользуйтесь функциями-накопителями.

- В модуле **Data.Monoid** определён специальный тип с помощью которого можно накапливать функции. Только функции должны быть специального типа. Они должны принимать и возвращать значения одного типа. Такие функции называют *эндоморфизмами*.

Посмотрим на их определение:

```
newtype Endo a = Endo { appEndo :: a -> a }
```

```
instance Monoid (Endo a) where
```

```
    mempty = Endo id
```

```
    Endo f `mappend` Endo g = Endo (f . g)
```

В качестве нейтрального элемента выступает функция тождества, а функцией объединения значений является функция композиции. Попробуйте переписать примеры из главы накопление чисел с помощью этого типа.

- Реализуйте с помощью монады **ST** какой-нибудь алгоритм в императивном стиле. Например алгоритм поиска корней уравнения методом деления пополам. Если функция f непрерывна и в двух точках a и b ($a < b$) значения функции имеют разные знаки, то это говорит о том, что где-то на отрезке $[a, b]$ уравнение $f(x) = 0$ имеет решение. Мы можем найти его так. Посмотрим какой знак у значения функции в середине отрезка. Если значение равно нулю, то нам повезло и мы нашли решение, если нет, то из двух концов отрезка выберем тот, у которого знак значения функции f отличается от знака значения в середине отрезка. Далее повторим эту процедуру на новом отрезке. И так пока мы не найдём корень или отрезок не стянется в точку. Внутри функции выделите память под концы отрезка и последовательно изменяйте их внутри типа **ST**.

IO

Пока мы не написали ещё ни одной программы, которой можно было бы пользоваться вне интерпретатора. Предполагается, что программа как-то взаимодействует с пользователем (ожидает ввода с клавиатуры) и изменяет состояние компьютера (выводит сообщения на экран, записывает данные в файлы). Но пока что мы не знаем как взаимодействовать с окружающим миром.

Самое время узнать! Сначала мы посмотрим какие проблемы связаны с реализацией взаимодействия с пользователем. Как эти проблемы решаются в Haskell. Потом мы научимся решать несколько типичных задач, связанных с вводом/выводом.

Чистота и побочные эффекты

Когда мы определяем новые функции или константы мы лишь даём новые имена комбинациям значений. В этом смысле у нас ничего не изменяется. По-другому это называется *функциональной чистотой* (referential transparency). Это свойство говорит о том, что мы свободно можем заменить в тексте программы любой синоним на его определение и это никак не скажется на результате.

Функция является чистой, если её выход зависит только от её входов. В любой момент выполнения программы для одних и тех же входов будет один и тот же выход. Это свойство очень ценно. Оно облегчает понимание поведения функции. Оно говорит о том, что функция может зависеть от других функций только *явно*. Если мы видим, что другая функция используется в данной функции, то она используется в этой функции. У нас нет таинственных глобальных переменных, в которые мы можем записывать данные из одной функции и читать их с помощью другой. Мы вообще не можем ничего записывать и ничего читать. Мы не можем изменять состояния, мы можем лишь давать новые имена или строить новые выражения из уже существующих.

Но в этот статичный мир описаний не вписывается взаимодействие с пользователем. Предположим, что мы хотим написать такую

программу: мы набираем на клавиатуре имя файла, нажимаем **Enter** и программа показывает на экране содержимое этого файла, затем мы набираем текст, нажимаем **Enter** и текст дописывается в конец файла, файл сохраняется. Это описание предполагает упорядоченность действий. Мы не можем сначала сохранить текст, затем прочитать обновления. Тогда текст останется прежним.

Ещё один пример. Предположим у нас есть функция `getChar`, которая читает букву с клавиатуры. И функция `print`, которая выводит строку на экран И посмотрим на такое выражение:

```
let c = getChar
in print $ c : c : []
```

О чём говорит это выражение? Возможно, прочитай с клавиатуры букву и выведи её на экран дважды. Но возможен и другой вариант, если в нашем языке все определения это синонимы мы можем записать это выражение так:

```
print $ getChar : getChar : []
```

Это выражение уже говорит о том, что читать с клавиатуры необходимо дважды! А ведь мы сделали обычное преобразование, заменили вхождения синонима на его определение, но смысл изменился. Взаимодействие с пользователем нарушает чистоту функций, нечистые функции называются функциями с побочными эффектами.

Как быть? Можно ли внести в мир описаний порядок выполнения, сохранив преимущества функциональной чистоты? Долгое время этот вопрос был очень трудным для чистых функциональных языков. Как можно пользоваться языком, который не позволяет сделать такие базовые вещи как ввод/вывод?

Монада IO

Где-то мы уже встречались с такой проблемой. Когда мы говорили о типе **ST** и обновлении значений. Там тоже были проблемы порядка вычислений, нам удалось преодолеть их с помощью скрытой передачи фиктивного состояния. Тогда наши обновления были *чистыми*, мы

могли безболезненно скрыть их от пользователя. Теперь всё гораздо труднее. Нам всё-таки хочется взаимодействовать с внешним миром. Для обозначения внешнего мира мы определим специальный тип и назовём его `RealWorld`:

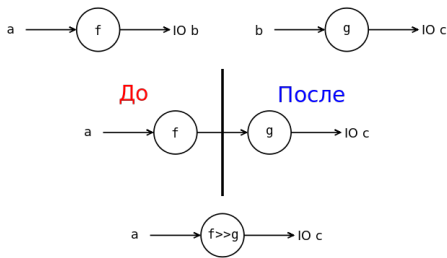
```
module IO(  
    IO  
) where  
  
data RealWorld = RealWorld  
  
newtype IO a = IO (ST RealWorld a)  
  
instance Functor      IO where ...  
instance Applicative  IO where ...  
instance Monad        IO where ...
```

Тип `IO` (от англ. input-output или ввод-вывод) обозначает взаимодействие с внешним миром. Внешний мир словно является состоянием наших вычислений. Экземпляры классов композиции специальных функций такие же как и для `ST` (а следовательно и для `State`). Но при этом, поскольку мы конкретизировали первый параметр типа `ST`, мы уже не сможем воспользоваться функцией `runST`.

Тип `RealWorld` определён в модуле `Control.Monad.ST`, там же можно найти и функцию:

```
stToIO :: ST RealWorld a -> IO a
```

Интересно, что класс `Monad` был придуман как раз для решения проблемы ввода-вывода. Классы типов изначально задумывались для решения проблемы определения арифметических операций на разных числах и функции сравнения на равенство для разных типов, мало кто тогда догадывался, что классы типов сыграют такую роль, станут основополагающей особенностью языка.



Композиция для монады IO

Это рисунок для класса `Kleisli`. Здесь под `>>` понимается композиция, как мы её определяли в главе 6, а не метод класса `Monad`, вспомним определение:

```
class Kleisli m where
  idK  :: a -> m a
  (>>) :: (a -> m b) -> (b -> m c) -> (a -> m c)
```

Композиция специальных функций типа `a -> IO b` вносит порядок вычисления. Считается, что сначала будет вычислена функция слева от композиции, а затем функция справа от композиции. Это происходит за счёт скрытой передачи фиктивного состояния. Теперь перейдём к классу `Monad`. Там композиция заменяется на применение или операция связывания:

```
ma >>= mf
```

Для типа `IO` эта запись говорит о том, что сначала будет выполнено выражение `ma` и результат будет подставлен в выражение `mf` и только затем будет выполнено `mf`. Оператор связывания для специальных функций вида:

```
a -> IO b
```

раскалывает наш статический мир на “до” и “после”. Однажды попав в сети `IO`, мы не можем из них выбраться, поскольку теперь у нас нет функции `runST`. Но это не так страшно. Тип `IO` дробит наш статический мир на кадры. Но мы спокойно можем создавать статические чистые функции и поднимать их в мир `IO` лишь там где это действительно нужно.

Рассмотрим такой пример, программа читает с клавиатуры начальное значение, затем загружает файл настроек. Потом запускается, какая-то сложная функция и в самом конце мы выводим результат на экран.

Схематично мы можем записать эту программу так:

```
program = liftA2 algorithm readInit (readConfig "file") >=> print
```

```
-- функции с побочными эффектами
```

```
readInit    :: IO Int
```

```
readConfig  :: String -> IO Config
```

```
print       :: Show a => a -> IO ()
```

```
-- большая и сложная, но !чистая! функция
```

```
algorithm   :: Int -> Config -> Result
```

Функция `readInit` читает начальное значение, функция `readConfig` читает из файла настройки, функция `print` выводит значение на экран, если это значение можно преобразовать в строку. Функция `algorithm` это большая функция, которая вычисляет какие-то данные. Фактически наша программа это и есть функция `algorithm`. В этой схеме мы добавили взаимодействие с пользователем лишь в одном месте, вся функция `algorithm` построена по правилам мира описаний. Так мы внесли порядок выполнения в программу, сохранив возможность определения чистых функций.

Если у нас будет ещё один “кадр”, ещё одно действие, например как только функция `algorithm` закончила вычисления ей нужны дополнительные данные от пользователя, на основе которых мы сможем продолжить вычисления с помощью какой-нибудь другой функции. Тогда наша программа примет вид:

```
program =
```

```
    liftA2 algorithm2 readInit
```

```
      (liftA2 algorithm1 readInit (readConfig "file"))
```

```
    >=> print
```

```
-- функции с побочными эффектами
```

```
readInit    :: IO Int
```

```
readConfig  :: String -> IO Config
```

```
print       :: Show a => a -> IO ()
```

```
-- большие и сложные, но !чистые! функции
```

```
algorithm1   :: Int -> Config -> Result1
```

```
algorithm2   :: Int -> Result1 -> Result2
```

Теперь у нас два кадра, программа выполняется в два этапа. Каждый из них разделён участками взаимодействия с пользователем. Но тип `IO` присутствует лишь в первых шести строчках, остальные два миллиона строк написаны в мире описаний, исключительно чистыми функциями, которые поднимаются в мир специальных функций с помощью функций `liftA2` и стыкуются с помощью операции связывания `>>=`.

Попробуем тип `IO` в интерпретаторе. Мы будем пользоваться двумя стандартными функциями `getChar` и `print`

```
-- читает символ с клавиатуры
getChar :: IO Char

-- выводит значение на экран
print :: IO ()
```

Функция `print` возвращает значение единичного типа, завёрнутое в тип `IO`, поскольку нас интересует не само значение а побочный эффект, который выполняет эта функция, в данном случае это вывод на экран.

Закодируем два примера из первого раздела. В первом мы читаем один символ и печатаем его дважды:

```
Prelude> :m Control.Applicative
Prelude Control.Applicative> let res = (\c -> c:c:[]) <$> getChar >>= print
Prelude Control.Applicative> res
q"qq"
```

Мы сначала вызываем функцию `getChar` удваиваем результат функцией `\c -> c:c:[]` и затем выводим на экран.

Во втором примере мы дважды запрашиваем символ с клавиатуры а затем печатаем их:

```
Prelude Control.Applicative> let res = liftA2 (\a b -> a:b:[]) getChar getChar
>>= print
Prelude Control.Applicative> res
qw"qw"
```


Как пишутся программы

Мы уже умеем читать с клавиатуры и выводить значения на экран. Давайте научимся писать самостоятельные программы. Программа обозначается специальным именем:

```
main :: IO ()
```

Если модуль называется `Main` или в нём нет директивы `module ... where` и в модуле есть функция `main :: IO ()`, то после компиляции будет сделан исполняемый файл. Его можно запускать независимо от `ghci`. Просто нажимаем дважды мышкой или вызываем из командной строки.

Напишем программу `Hello world`. Единственное, что она делает это выводит на экран приветствие:

```
main :: IO ()
main = print "Hello World!"
```

Теперь сохраним эти строчки в файле `Hello.hs`, перейдём в директорию файла и скомпилируем файл:

```
ghc --make Hello
```

Появились объектный и интерфейсный файлы, а также появился третий бинарный файл. Это либо `Hello` без расширения (в Linux) или `Hello.exe` (в Windows). Запустим этот файл:

```
$ ./Hello
"Hello World!"
```

Получилось! Это наша первая программа. Теперь напишем программу, которая принимает три символа с клавиатуры и выводит их в обратном порядке:

```
import Control.Applicative

f :: Char -> Char -> Char -> String
f a b c = reverse $ [a,b,c]

main :: IO ()
main = print =<< f <$> getChar <*> getChar <*> getChar
```

Сохраним в файле `ReverseIO.hs` и скомпилируем:

```
ghc --make ReverseIO -o rev3
```

Дополнительным флагом `-o` мы попросили компилятор чтобы он сохранил исполняемый файл под именем `rev3`. Теперь запустим в командной строке:

```
$ ./rev3
qwe
"ewq"
```

Набираем три символа и нажимаем ввод. И программа переворачивает ответ. Обратите внимание на то, что с помощью `print` мы выводим не просто строку на экран, а строку как значение. Поэтому добавляются двойные кавычки. Для того чтобы выводить строку существует функция `putStr`. Заменим `print` на `putStr`, перекомпилируем и посмотрим что получится:

```
$ ghc --make ReverseIOstr -o rev3str
[1 of 1] Compiling Main                ( ReverseIOstr.hs, ReverseIOstr.o )
Linking rev3str ...
$ ./rev3str
123
321$
```

Видно, что после вывода не произошёл перенос каретки, терминал приглашает нас к вводу команды сразу за ответом, если перенос нужен, можно воспользоваться функцией `putStrLn`. Обратите внимание на то, что кроме бинарного файла появились ещё два файла с расширениями `.hi` и `.o`. Первый файл называется интерфейсным он описывает какие в модуле определения, а второй файл называется объектным. Он содержит скомпилированный код модуля.

Стоит отметить команду `runhaskell`. Она запускает программу без создания дополнительных файлов. Но в этом случае выполнение программы будет происходить медленнее.

Типичные задачи IO

Вывод на экран

Нам уже встретилось несколько функций вывода на экран. Это функции: `print` (вывод значения из экземпляра класса `Show`), `putStr`

(вывод строки) и `putStrLn` (вывод строки с переносом). Каждый раз когда мы набираем какое-нибудь выражение в строке интерпретатора и нажимаем **Enter**, интерпретатор применяет к выражению функцию `print` и мы видим его на экране.

Из простейших функций вывода на экран осталось не рассмотренной лишь функция `putChar`, но я думаю вы без труда догадаетесь по типу и имени чем она занимается:

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

Функции вывода на экран также можно вызывать в интерпретаторе:

```
Prelude> putStr "Hello" >> putChar ' ' >> putStrLn "World!"  
Hello World!
```

Обратите внимание на применение постоянной функции для монад `>>`. В этом выражении нас интересует не результат, а те побочные эффекты, которые выполняются при композиции специальных функций. Также мы пользовались функцией `>>` в сочетании с монадой **Writer** для накопления результата.

Ввод пользователя

Мы уже умеем принимать от пользователя буквы. Это делается функцией `getChar`. Функцией `getLine` мы можем прочитать целую строчку. Строка читается до тех пор пока мы не нажмём **Enter**.

```
Prelude> fmap reverse $ getLine  
Hello-hello!  
"!olleh-olleH"
```

Есть ещё одна функция для чтения строк, она называется `getContents`. Основное отличие от `getLine` заключается в том, что содержание не читается сразу, а откладывается на потом, когда содержание действительно понадобится. Это ленивый ввод. Для задачи чтения символов с терминала эта функция может показаться странной. Но часто в символы вводятся не вручную, а передаются из другого файла. Например, если мы направим на ввод данные из какого-нибудь большого-большого файла, файл не будет читаться сразу, и память не будет заполнена не нужным пока содержанием. Вместо этого программа отложит считывание на потом и будет

заниматься им лишь тогда, когда оно понадобится в вычислениях. Это может существенно снизить расход памяти. Мы читаем файл в 2Гб моментально (мы делаем вид, что читаем его). А на самом деле сохраняем себе задачу на будущее: читать ввод, когда придёт пора.

Чтение и запись файлов

Для чтения и записи файлов есть три простые функции:

```
type FilePath = String

-- чтение файла
readFile      :: FilePath -> IO String

-- запись строки в файл
writeFile     :: FilePath -> String -> IO ()

-- добавление строки в конце файла
appendFile   :: FilePath -> String -> IO ()
```

Напишем программу, которая сначала запрашивает путь к файлу. Затем показывает его содержание. Затем запрашивает ввод строки из терминала. А после этого добавляет текст в конец файла.

```
main = msg1 >> getLine >>= read >>= append
      where read  file = readFile file >>= putStrLn >> return file
            append file = msg2 >> getLine >>= appendFile file
            msg1      = putStr "input file: "
            msg2      = putStr "input text: "
```

В самом левом вызове `getLine` мы читаем имя файла, затем оно используется в локальной функции `read`. Там мы читаем содержание файла (`readLine`), выводим его на экран (`putStrLn`), и в самом конце мы возвращаем из функции имя файла. Оно нам понадобится в следующей части программы, в которой мы будем читать новые записи и добавлять их в файл. Новая запись читается функцией `getLine` в локальной функции `append`.

Сохраним в модуле `File.hs` и посмотрим, что у нас получилось. Перед этим создадим в текущей директории тестовый пустой файл под именем `test`. В него мы будем добавлять новые записи.

```
*Prelude> :l File
[1 of 1] Compiling File               ( File.hs, interpreted )
Ok, modules loaded: File.
*File> main
```

```
input file: test

input text: Hello!
*File> main
input file: test
Hello!
input text: Hi)
*File> main
input file: test
Hello!Hi)
```

В самом начале наш файл пуст, поэтому сначала мы видим пустую строчку вместо содержания, но потом мы начинаем добавлять в него новые записи.

Ленивое и энергичное чтение файлов

С чтением файлов связана одна тонкость. Функция `readFile` читает содержимое файла в ленивом стиле. Подробнее о ленивой стратегии вычислений мы поговорим в следующей главе. Пока отметим, что `readFile` не читает следующую порцию файла до тех пор пока она не понадобится в программе. Иногда это очень удобно. Например мы можем читать содержание очень большого файла и составлять какую-нибудь статистику на основе прочитанного текста. При этом в памяти будет храниться лишь малая часть файла. Но иногда это свойство мешает. Рассмотрим такую задачу: перевернуть текст в файле под именем "test". Мы должны сначала считать текст из файла, затем перевернуть его и в конце записать в *тот же* файл. Мы могли бы написать эту программу так:

```
module Main where
```

```
main :: IO ()
main = inFile reverse "test"
```

```
inFile :: (String -> String) -> FilePath -> IO ()
inFile fun file = writeFile file . fun =<< readFile file
```

Функция `inFile` обновляет текст файла с помощью некоторого преобразование. Но если мы запустим эту программу:

```
*Main> main
*** Exception: test: openFile: resource busy (file is locked)
```

Мы получили ошибку. Мы пытаемся писать в файл, который уже занят для чтения. Дело в том, что функция `readFile` заняла файл, за счёт чтения по кусочкам. Для решения этой проблемы необходимо воспользоваться энергичной версией функции `readFile`, она будет читать файл целиком. Эта функция живёт в модуле `System.IO.Strict`:

```
import qualified System.IO.Strict as StrictIO

inFile :: (String -> String) -> FilePath -> IO ()
inFile fun file = writeFile file . fun =<< StrictIO.readFile file
```

Функция `main` осталась прежней. Теперь наша программа спокойно переворачивает текст файла.

Аргументы программы

Пока программы, которые мы создавали просили пользователя ввести данные вручную при выполнении программы, они работали в интерактивном режиме, но чаще всего программы принимают какие-нибудь начальные данные, установки или флаги. Читать начальные данные можно с помощью функций из модуля `System.Environment`.

Узнать, что передаётся в программу можно функцией `getArgs :: IO [String]`. Она возвращает список строк. Это те строки, что мы написали за именем программы через пробел при вызове в терминале. Напишем простую программу, которая распечатывает свои аргументы по порядку, в виде пронумерованного списка.

```
module Main where

import System.Environment

main = getArgs >= mapM_ putStrLn . zipWith f [1 .. ]
      where f n a = show n ++ ": " ++ a
```

В локальной функции `f` мы присоединяем к строке номер через двоеточие. Функцией `mapM_` мы пробегаем по списку строк, отображая их с помощью функции `putStrLn`. Обратите внимание на краткость программы, с помощью функции композиции мы легко составили функцию, которая приписывает к аргументам числа, а затем выводит их на экран.

Скомпилируем программу в интерпретаторе и вызовем её.

```

*Main> :! ghc --make Args
[1 of 1] Compiling Main                ( Args.hs, Args.o )
Linking Args ...
*Main> :! ./Args hey hey hey 23 54 "qwe qwe qwe" fin
1: hey
2: hey
3: hey
4: 23
5: 54
6: qwe qwe qwe
7: fin

```

Если мы хотим, чтобы аргумент-строка содержал пробелы мы заключаем его в двойные кавычки.

С помощью функции `getProgName` можно узнать имя программы. Создадим программу, которая здоровается при вызове. И отвечает в зависимости от настроения программы. Настроение задаётся аргументом программы.

```

module Main where

import Control.Applicative
import System.Environment

main = putStrLn =<< reply <$> getProgName <*> getArgs

reply :: String -> [String] -> String
reply name (x:_) = hi name ++ case x of
    "happy"      -> "What a lovely day. What's up?"
    "sad"        -> "Ooohh. Have you got some news for me?"
    "neutral"    -> "How are you?"
reply name _     = reply name ["neutral"]

hi :: String -> String
hi name = "Hi! My name is " ++ name ++ ".\n"

```

В функции `reply` мы составляем реплику программы. Она зависит от имени программы и поступающих на вход аргументов. Посмотрим, что у нас получилось:

```

*Main> :! ghc --make HowAreYou.hs -o ninja
[1 of 1] Compiling Main                ( HowAreYou.hs, HowAreYou.o )
Linking ninja ...
*Main> :! ./ninja happy
Hi! My name is ninja.
What a lovely day. What's up?
*Main> :! ./ninja sad

```

Hi! My name is ninja.

Ooohh. Have you got some news for me?

Вызов других программ

Мы можем вызвать любую программу из нашей программы. Это делается с помощью функции `system`, которая живёт в модуле `System`.

```
system :: String -> IO ExitCode
```

Она принимает строку и запускает её в терминале. Так же как мы делали это с помощью приставки `!` в интерпретаторе. Значение типа `ExitCode` говорит о результате выполнения строки. Он может быть успешным, тогда функция вернёт `ExitSuccess` и закончиться ошибкой, тогда мы сможем узнать код ошибки по значению `ExitFailure Int`.

Случайные значения

Функции для создания случайных значений определены в модуле `System.Random`. Модуль `System.Random` входит в библиотеку `random`. Если в вашей поставке `ghc` его не оказалось, вы можете установить его вручную через интернет, набрав в командной строке `cabal install random`. Сначала давайте разберёмся как генерируются случайные числа. Стандартные случайные числа очень похожи на те, что были у нас, когда мы рассматривали примеры специальных функций. У нас есть генератор случайных чисел типа `g` и с помощью функции `next` мы можем получить обновлённый генератор и случайное целое число:

```
next :: g -> (Int, g)
```

Не правда ли этот тип очень похож на тип результата функций с состоянием. В качестве состояния теперь выступает генератор случайных чисел `g`. Это поведение описывается классом `RandomGen`:

```
class RandomGen g where
  next      :: g -> (Int, g)
  split     :: g -> (g, g)
  getRange  :: g -> (Int, Int)
```

Функция `next` обновляет генератор и возвращает случайное значение типа `Int`. Функция `split` раскалывает один генератор на два.

Функция `genRange` возвращает диапазон значений генерируемых случайных чисел. Первое значение в паре результата `genRange` должно быть всегда меньше второго. Для этого класса определён один экземпляр, это тип `StdGen`. Мы можем создать первый генератор по целому числу с помощью функции `mkStdGen`:

```
mkStdGen :: Int -> StdGen
```

Давайте посмотрим как это происходит в интерпретаторе:

```
Prelude> :m System.Random
Prelude System.Random> let g0 = mkStdGen 0
Prelude System.Random> let (n0, g1) = next g0
Prelude System.Random> let (n1, g2) = next g1
Prelude System.Random> n0
2147482884
Prelude System.Random> n1
2092764894
```

Мы создали первый генератор, а затем начали получать новые. Для того, чтобы получать новые случайные числа, нам придётся таскать везде за собой генератор случайных чисел. Мы можем обернуть его в функцию с состоянием и пользоваться методами классов `Functor`, `Applicative` и `Monad`. Обновление генератора будет происходить за ширмой, во время применения функций. Но у нас есть и другой путь.

Вместо монады `State` мы можем воспользоваться монадой `IO`. Если нам лень определять генератор случайных чисел, мы можем попросить компьютер определить его за нас. В этом случае мы взаимодействуем с компьютером, мы запрашиваем глобальное для системы случайное значение, поэтому возвращаемое значение будет завернуто в тип `IO`. Для этого определены функции:

```
getStdGen :: IO StdGen
newStdGen :: IO StdGen
```

Функция `getStdGen` запрашивает глобальный для системы генератор случайных чисел. Функция `newStdGen` не только запрашивает генератор, но также и обновляет его. Мы пользуемся этими функциями так же как и `mkStdGen`, только теперь мы спрашиваем первый аргумент у компьютера, а не передаём его вручную. Также есть ещё одна полезная функция:

```
getStdRandom    :: (StdGen -> (a, StdGen)) -> IO a
```

Посмотрим, что получится, если передать в неё функцию next:

```
Prelude System.Random> getStdRandom next
1386438055
Prelude System.Random> getStdRandom next
961860614
```

И не надо обновлять никаких генераторов. Но вместо одного неудобства мы получили другое. Теперь значение завёрнуто в оболочку `IO`.

Генератор `StdGen` делает случайные числа из диапазона всех целых чисел. Что если мы хотим получить только числа из некоторого интервала? И как получить случайные значения других типов? Для этого существует класс `Random`. Он является удобной надстройкой над классом `RandomGen`. Посмотрим на его основные методы:

```
class Random a where
  randomR :: RandomGen g => (a, a) -> g -> (a, g)
  random  :: RandomGen g => g -> (a, g)
```

Метод `randomR` принимает диапазон значений, генератор случайных чисел и возвращает случайное число из указанного диапазона и обновлённый генератор. Метод `random` является синонимом метода `next` из класса `RandomGen`, только теперь мы можем получать не только целые числа.

Есть и дополнительные методы. Есть методы, которые позволяют генерировать список всех возможных случайных значений для данного генератора:

```
randomRs :: RandomGen g => (a, a) -> g -> [a]
randoms  :: RandomGen g => g -> [a]
```

За счёт лени мы будем получать новые значения по мере необходимости.

```
randomRIO :: (a, a) -> IO a
randomIO  :: IO a
```

Эти функции выполняют тоже, что и основные функции класса, но им не нужен генератор случайных чисел, они создают его с помощью

функции `getStdRandom`. Экземпляры `Random` определены для `Bool`, `Char`, `Double`, `Float`, `Int` и `Integer`. Например так мы можем подбросить кости десять раз:

```
Prelude System.Random> fmap (take 10 . randomRs (1, 6)) getStdGen
[5,6,5,5,6,4,6,4,4,4]
Prelude System.Random> fmap (take 10 . randomRs (1, 6)) getStdGen
[5,6,5,5,6,4,6,4,4,4]
```

Обратите внимание на то, что функция `getStdGen` не обновляет генератор случайных чисел. Мы запрашиваем глобальное состояние. Поэтому, дважды подбросив кубик, мы получили одни и те же результаты. Генератор будет обновляться, если воспользоваться функцией `newStdGen`:

```
Prelude System.Random> fmap (take 10 . randomRs (1, 6)) newStdGen
[1,1,5,6,5,2,5,5,5,3]
Prelude System.Random> fmap (take 10 . randomRs (1, 6)) newStdGen
[5,4,6,5,5,5,1,5,5,2]
```

Создадим случайные слова из пяти букв:

```
Prelude System.Random> fmap (take 5 . randomRs ('a', 'z')) newStdGen
"maclg"
Prelude System.Random> fmap (take 5 . randomRs ('a', 'z')) newStdGen
"nfjoa"
```

Цитатник

Напишем небольшую программу, которая будет выводить на экран в случайном порядке цитаты. Цитаты хранятся в виде списка пар (автор, высказывание). Сначала мы генерируем случайное число в диапазоне длины списка, затем выбираем цитату под этим номером и выводим её на экран.

```
module Main where
```

```
import Control.Applicative
import System.Random
```

```
main =
  format . (quotes !! ) <$> randomRIO (0, length quotes - 1)
  >>= putStrLn
```

```
format (a, b) = b ++ space ++ a ++ space
  where space = "\n\n"
```

```
quotes = [
    ("Бьёрн Страуструп",
     "Есть лишь два вида языков программирования: те, \
      \ на которые вечно жалуются, и те, которые никогда \
      \ не используются."),
    ("Мохатма Ганди", "Ты должен быть теми изменениями, которые\
      \ ты хочешь видеть вокруг."),
    ("Сократ", "Я знаю лишь то, что ничего не знаю."),
    ("Китайская народная мудрость", "Сохранив спокойствие в минуту\
      \ гнева, вы можете избежать сотни дней сожалений"),
    ("Жан Батист Мольер", "Медленно растущие деревья приносят лучшие плоды"),
    ("Антуан де Сент-Экзюпери", "Жить это значит медленно рождаться"),
    ("Альберт Эйнштейн", "Фантазия важнее знания."),
    ("Тони Хоар", "Внутри любой большой программы всегда есть\
      \ маленькая, что рвётся на свободу"),
    ("Пифагор", "Не гоняйся за счастьем, оно всегда находится в тебе самом"),
    ("Лао Цзы", "Путешествие в тысячу ли начинается с одного шага")]
```

Функция `format` приводит цитату к виду приятному для чтения.
 Попробуем программу в интерпретаторе:

```
Prelude> :! ghc --make Quote -o hi
[1 of 1] Compiling Main                ( Quote.hs, Quote.o )
Linking hi ...
Prelude> :! ./hi
Путешествие в тысячу ли начинается с одного шага
```

Лао Цзы

```
Prelude> :! ./hi
Не гоняйся за счастьем, оно всегда находится в тебе самом
```

Пифагор

Исключения

Мы уже знаем несколько типов, с помощью которых функции могут сказать, что что-то случилось не так. Это типы `Maybe` и `Either`. Если функции не удалось вычислить значение она возвращает специальное значение `Nothing` или `Left` reason, по которому следующая функция может опознать ошибку и предпринять какие-нибудь действия. Так обрабатываются ошибки в чистых функциях. В этом разделе мы узнаем о том, как обрабатываются ошибки, которые происходят при взаимодействии с внешним миром, ошибки, которые происходят внутри типа `IO`.

Ошибки функций с побочными эффектами обрабатываются с помощью специальной функции `catch`, она определена в `Prelude`:

```
catch :: IO a -> (IOError -> IO a) -> IO a
```

Эта функция принимает значение, которое содержит побочные эффекты и функцию, которая обрабатывает исключительные ситуации. К примеру если мы попытаемся прочитать данные из файла, к которому у нас нет доступа, произойдёт ошибка. Мы можем не дать программе упасть и обработать ошибку с помощью функции `catch`.

Например программа, в которой мы дописывали данные в файл, упадёт, если мы передадим не существующий файл. Но мы можем исправить это поведение с помощью функции `catch`. Мы можем перезапускать программу, если произошла ошибка:

```
module FileSafe where

import Control.Applicative
import Control.Monad

main = try `catch` const main

try = msg1 >> getLine >>= read >>= append
  where read    file = readFile file >>= putStrLn >> return file
        append file = msg2 >> getLine >>= appendFile file
        msg1      = putStr "input file: "
        msg2      = putStr "input text: "
```

Часто функции двух аргументов называют так, чтобы при инфиксной форме записи получалась фраза из английского языка. Так если мы запишем `catch` в инфиксной форме получится очень наглядное выражение. Функция обработки ошибок реагирует на любую ошибку перезапуском программы. Попробуем взломать программу:

```
*FileSafe> main
input file: fsldfksld
input file: sd;fls;df1;vll; d;fld;f
input file: dflks;ldkf ldkfldkfld
input file: lsdkfksdlf ksdkf1sdfkls;dfk
input file: bfk
input file: test
Hello!Hi)
input text: HowHow
```

Функция будет запрашивать файл до тех пор, пока мы не введём корректное значение. Мы можем добавить сообщение об ошибке, немного изменив функцию обработки:

```
main = try `catch` const (msg >> main)
      where msg = putStrLn "Wrong filename, try again."
```

А что делать если нам хочется различать ошибки по типу и предпринимать различные действия в зависимости от типа ошибки? Ошибки распознаются с помощью специальных предикатов, которые определены в модуле `System.IO.Error`. Рассмотрим некоторые из них.

Например с помощью с помощью предиката `isDoesNotExistErrorType` мы можем опознать ошибки, которые случились из-за того, что один из аргументов функции не существует. С помощью предиката `isPermissionErrorType` мы можем узнать, что ошибка произошла из-за того, что мы пытались получить доступ к данным, на которые у нас нет прав. Мы можем, немного изменив функцию-обработчик исключений, выводить более информативные сообщения об ошибках перед перезапуском:

```
main = try `catch` handler

handler :: IOError -> IO ()
handler = ( >> main) . putStrLn . msg2 . msg1

msg1 e
| isDoesNotExistErrorType e = "File does not exist. "
| isPermissionErrorType e   = "Access denied. "
| otherwise                 = ""

msg2 = (++ "Try again.")
```

В модуле `System.IO.Error` вы можете найти ещё много разных предикатов.

Потоки текстовых данных

Обмен данными, чтение и запись происходят с помощью потоков. Каждый поток имеет *дескриптор* (`handle`), через него мы можем общаться с потоком, например считывать данные или записывать. Функции для работы с потоками данных определены в модуле `System.IO`.

В любой момент в системе открыты три стандартных потока:

- `stdin` – стандартный ввод
- `stdout` – стандартный вывод
- `stderr` – поток ошибок и отладочных сообщений

Например когда мы выводим строку на экран, на самом деле мы записываем строку в поток `stdout`. А когда мы читаем символ с клавиатуры, мы считываем его из потока `stdin`.

Файлы также являются потоками. При открытии файлу присваивается дескриптор через который, мы можем обмениваться данными. Файл может быть открыт для чтения, записи, дополнения (записи в конец файла) или чтения и записи. Файл открывается функцией:

```
openFile :: FilePath -> IOMode -> IO Handle
```

Функция принимает путь к файлу и режим работы с файлом и возвращает дескриптор. Режим может принимать одно из значений:

- `ReadMode` – чтение
- `WriteMode` – запись
- `AppendMode` – добавление (запись в конец файла)
- `ReadWriteMode` – чтение и запись

Открыв дескриптор, мы можем начать обмениваться данными. Для этого определены функции аналогичные тем, что мы уже рассмотрели. Функции для записи данных:

```
-- запись символа
```

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

```
-- запись строки
```

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

```
-- запись строки с переносом каретки
```

```
hPutStrLn :: Handle -> String -> IO ()
```

```
-- запись значения
```

```
hPrint :: Show a => Handle -> a -> IO ()
```

Все функции принимают первым аргументом дескриптор потока. Дескриптор должен позволять записывать данные. Например для

дескриптора, открытого в режиме `ReadMode`, выполнение этих функций приведёт к ошибке.

Из потоков также можно читать данные. Эти функции похожи на те, что мы уже рассмотрели:

```
-- чтение одного символа
hGetChar :: Handle -> IO Char

-- чтение строки
hGetLine :: Handle -> IO String

-- ленивое чтение строки
hGetContents :: Handle -> IO String
```

Как только, мы закончим работу с файлом, его необходимо закрыть. Нам нужно освободить дескриптор. Сделать это можно функцией `hClose`:

```
hClose :: Handle -> IO ()
```

Стандартные функции ввода/вывода, которые мы рассмотрели ранее определены через функции работы с дескрипторами. Например так мы выводим строку на экран:

```
putStr      :: String -> IO ()
putStr s    = hPutStr stdout s
```

А так читаем строку с клавиатуры:

```
getLine     :: IO String
getLine     = hGetLine stdin
```

В этих функциях используются дескрипторы стандартных потоков данных `stdin` и `stdout`. Отметим функцию `withFile`:

```
withFile :: FilePath -> IOMode -> (Handle -> IO r) -> IO r
```

Она открывает файл в заданном режиме выполняет функцию на его дескрипторе и и закрывает файл. Например через эту функцию определены функции `readFile` и `appendFile`:

```
appendFile :: FilePath -> String -> IO ()
appendFile f txt = withFile f AppendMode (\hdl -> hPutStr hdl txt)

writeFile :: FilePath -> String -> IO ()
writeFile f txt = withFile f WriteMode (\hdl -> hPutStr hdl txt)
```


Форточка в мир побочных эффектов

В самом начале главы я сказал о том, что из мира `IO` нет выхода. Нет функции с типом `IO a -> a`. На самом деле выход есть. Функция с таким типом живёт в модуле `System.IO.Unsafe`:

```
unsafePerformIO :: IO a -> a
```

Длинное имя функции намекает на то, что её необходимо использовать с *крайней* осторожностью. Поскольку последствия могут быть непредсказуемыми.

Эта функция используется при чтении конфигурационных файлов. Если есть уверенность в том, что файл будет только читаться и во время выполнения программы файл не может быть изменён другой программой, то мы можем считать, что его значение окажется неизменным на протяжении работы программы. Это говорит о том, что нам не важно когда читать данные. Поэтому здесь мы вроде бы ничем не рискуем. “Вроде бы” потому что ответственность за постоянство файла лежит на наших плечах.

Эта функция часто используется при вызове функций `C` через `Haskell`. В `Haskell` есть возможность вызывать функции, написанные на `C`. Но по умолчанию такие функции заворачиваются в тип `IO`. Если функция является чистой в `C`, то она будет чистой и при вызове через `Haskell`. Мы можем поручиться за её чистоту и вычислитель нам поверит. Но если мы его обманули, мы пожнём плоды своего обмана.

Отладка программ

Раз уж речь зашла о “грязных” возможностях языка стоит упомянуть функцию `trace` из модуля `Debug.Trace`. Посмотрим на её тип:

```
trace :: String -> a -> a
```

Это служебная функция эхо-печати. Когда дело доходит до вычисления функции `trace` на экран выводится строка, которая была передана в неё первым аргументом, после чего функция возвращает второй аргумент. Это функция `id` с побочным эффектом вывода

сообщения на экран. Ею можно пользоваться для отладки. Например так можно вернуть значение и распечатать его:

```
echo :: Show a => a -> a
echo a = trace (show a) a
```

Композиция монад

Эта глава завершает наше путешествие в мире типов-монад. Мы начали наше знакомство с монадами с композиции, мы определили класс `Monad` через класс `Kleisli`, который упрощал составление специальных функций вида `a -> m b`. Тогда мы познакомились с самыми простыми типами монадами (списки и частично определённые функции), потом мы перешли к типам посложнее, мы научились проводить вычисления с состоянием. В этой главе мы рассмотрели самый важный тип монаду `IO`. Мне бы хотелось замкнуть этот рассказ на теме композиции. Мы поговорим о композиции нескольких монад. Если вы посмотрите в исходный код библиотеки `transformers`, то увидите совсем другое определение для `State`:

```
type State s = StateT s Identity

newtype StateT s m a = StateT { runStateT :: s -> m (a,s) }
newtype Identity a = Identity { runIdentity :: a }
```

Но так ли оно далеко от нашего? Давайте разберёмся. `Identity` это тривиальный тип обёртка. Мы просто заворачиваем значение в конструктор и ничего с ним не делаем. Вы наверняка сможете догадаться как определить экземпляры всех рассмотренных в этой главе классов для этого типа. Тип `StateT` больше похож на наше определение для `State`, единственное отличие – это дополнительный параметр `m` в который завёрнут результат функции обновления состояния. Если мы сотрём `m`, то получим наше определение. Это и сказано в определении для `State`

```
type State s = StateT s Identity
```

Мы передаём дополнительным параметром в `StateT` тип `Identity`, который как раз ничего и не делает с типом. Так мы получим наше исходное определение, но зачем такие премудрости? Такой тип принято называть *монадным трансформером* (monad transformer). Он

определяет композицию из нескольких монад в данном случае одной из монад является `State`. Посмотрим на экземпляр класса `Monad` для `StateT`

```
instance (Monad m) => Monad (StateT s m) where
  return a = StateT $ \s -> return (s, a)

  a >>= f = StateT $ \s0 ->
    runStateT a s0 >>= \(b, s1) -> runStateT (f b) s1
```

В этом определении мы пропускаем состояние через сито методов класса `Monad` для типа `m`. В остальном это определение ничем не отличается от нашего. Также определены и `ReaderT`, `WriterT`, `ListT` и `MaybeT`. Ключевым классом для всех этих типов является класс `MonadTrans`:

```
class MonadTrans t where
  lift :: Monad m => m a -> t m a
```

Этот тип позволяет нам заворачивать специальные значения типа `m` в значения типа `t`. Посмотрим на определение для `StateT`:

```
instance MonadTrans (StateT s) where
  lift m = StateT $ \s -> liftM (,s) m
```

Напомню, что функция `liftM` это тоже самое, что и функция `fmap`, только она определена через методы класса `Monad`. Мы создали функцию обновления состояния, которая ничего не делает с состоянием, а лишь прицепляет его к значению.

Приведём простой пример применения трансформеров. Вернёмся к примеру `FSM` из предыдущей главы. Предположим, что наш конечный автомат не только реагирует на действия, но и ведёт журнал, в который записываются все поступающие на вход события. За переход состояний будет по прежнему отвечать тип `State` только теперь он станет трансформером, для того чтобы включить возможность журналирования. За ведение журнала будет отвечать тип `Writer`. Ведь мы просто накапливаем записи.

Интересно, что для добавления новой возможности нам нужно изменить лишь определение типа `FSM` и функцию `fsm`, теперь они примут вид:

```
module FSMt where
```

```
import Control.Monad.Trans.Class
import Control.Monad.Trans.State
import Control.Monad.Trans.Writer
```

```
import Data.Monoid
```

```
type FSM s = StateT s (Writer [String]) s
```

```
fsm :: Show ev => (ev -> s -> s) -> (ev -> FSM s)
fsm transition e = log e >> run e
  where run e = StateT $ \s -> return (s, transition e s)
        log e = lift $ tell [show e]
```

Все остальные функции останутся прежними. Сначала мы подключили все необходимые модули из библиотеки transformers. В подфункции log мы сохраняем сообщение в журнал, а в подфункции run мы выполняем функцию перехода. Посмотрим, что у нас получилось:

```
*FSMt> let res = mapM speaker session
*FSMt> runWriter $ runStateT res (Sleep, Level 2)
(([(Sleep,Level 2),(Work,Level 2),(Work,Level 3),(Work,Level 2),
  (Sleep,Level 2)],(Sleep,Level 3)),
 ["Button","Louder","Quieter","Button","Louder"])
*FSMt> session
[Button,Louder,Quieter,Button,Louder]
```

Мы видим, что цепочка событий была успешно записана в журнал.

Для трансформеров с типом IO определён специальный класс:

```
class Monad m => MonadIO m where
  liftIO :: IO a -> m a
```

Этот класс живёт в модуле Control.Monad.IO.Class. С его помощью мы можем выполнять IO-действия внутри другой монады. Эта возможность бывает очень полезной. Вам она обязательно понадобится, если вы начнёте писать веб-сайты на Haskell (например в happstack) или будете пользоваться библиотеками, которые надстроены над C (например физический движок Hirtmunk).

Краткое содержание

Наконец-то мы научились писать программы! Программы, которые можно исполнять за пределами интерпретатора. Для этого нам

пришлось познакомиться с типом `IO`. Экземпляр класса `Monad` для этого типа интерпретируется специальным образом. Он вносит упорядоченность в выполнение программы. В нашем статическом мире описаний появляются такие понятия как “сначала”, “затем”, “до” и “после”. Но они не смогут нанести много вреда.

Вычисление операций с побочными эффектами разбивает программу на кадры. Но это не мешает нам писать основные функции в чистом виде, подставляя их по мере необходимости в изменчивый мир побочных эффектов с помощью методов из классов `Functor`, `Applicative`, `Monad`.

Мы узнали как в Haskell обстоят дела с такими типичными задачами мира побочных эффектов как ввод/вывод, чтение/запись файлов, генерация случайных значений, выполнение внешних программ, инициализация программ с помощью флагов. Также мы узнали о том, как обрабатываются специфические для типа `IO` исключения.

Упражнения

Старайтесь свести присутствие функций с побочными эффектами к минимуму. Идеальный случай, когда тип `IO` встречается только в функции `main`. Часто программы устроены более хитрым образом и функции с побочными эффектами пытаются расползтись по всему коду. Но даже в этом случае программу можно разделить на две части: в одной живут подлинные источники побочных эффектов, такие как чтение файлов, генерация случайных значений, а в другой – чистые функции. Старайтесь устроить программу так, чтобы она была максимально чистой. Чистые функции гораздо проще комбинировать, понимать, изменять.

- Это упражнение даёт вам возможность почувствовать преимущества чистого кода. Вспомните функцию поиска корней методом неподвижной точки (этот пример встречался в главе о ленивых вычислениях). Напишите на основе этого примера программу, которая будет распечатывать решение и последовательность приближений. Последовательность приближений состоит из текущего значения корня и расстоянии между корнями.

Напишите два варианта программы, в одном вы измените алгоритм так, чтобы печать происходила одновременно с вычислениями (не пользуясь функцией из модуля `Debug.Trace`). А в другом варианте алгоритм останется прежним. Но теперь вместо решения найдите список первых приближений до решения. А затем передайте его в отдельную функцию печати результатов.

В первом варианте алгоритм смешан с печатью. А во втором программа распадается на две части, часть вычислений и часть вывода результатов на экран. Сравните два подхода.

- Напишите программу для угадывания чисел. Компьютер загадал число в заданном диапазоне и просит вас угадать его. Если вы ошибаетесь он подсказывает: “холодно-горячо” или “больше-меньше”. Программа принимает два аргумента, которые определяют диапазон возможных значений для неизвестного числа.
- С помощью стандартных функций для генерации случайных чисел напишите программу, которая проводит состязание по игре в кости. Программа принимает аргументом суммарное число очков необходимых для победы. Двое игроков бросают по очереди кости побеждает тот, кто первым наберёт заданную сумму.

Сделайте так чтобы результаты выводились постепенно. С каждым нажатием на `Enter` вы подбрасываете кости (два шестигранных кубика). После каждого раунда программа выводит промежуточные результаты.

- Напишите программу, которая принимает два аргумента: набор слов разделённых пробелами и файл. А выводит она строки файла, в которых встречается данное слово.

Воспользуйтесь стандартными функциями из модуля `Data.List`

```
-- разбиение строки на подстроки по переносам каретки
lines :: String -> [String]
```

```
-- разбиение строки на подстроки по пробелам
words :: String -> [String]
```

```
-- возвращает True только в том случае, если
-- первый список полностью содержится во втором
isInfixOf :: Eq a => [a] -> [a] -> Bool
```

- Классы `Functor` и `Applicative` замкнуты относительно композиции. Это свойство говорит о том, что композиция (аппликативных) функторов снова является (аппликативным) функтором. Докажите это! Пусть дан тип, который описывает композицию двух типов:

```
newtype O f g a = O { unO :: f (g a) }
```

Определите экземпляры классов:

```
instance (Functor f, Functor g) => Functor (O f g) where ...
```

```
instance (Applicative f, Applicative g) => Applicative (O f g) where
...
```

Подсказка: если совсем не получается, ответ можно посмотреть в библиотеке `TypeCompose`. Но пока мы не знаем как устанавливать библиотеки и где они живут, всё-таки попробуйте решить это упражнение самостоятельно.

Редукция выражений

В этой главе мы поговорим о том как вычисляются программы. В самом начале мы говорили о том, что процесса вычисления значений нет. В том смысле, что у нас нет новых значений, у нас ничего не меняется, мы лишь расшифровываем синонимы значений.

Вкратце вспомним то, что мы уже знаем о вычислениях. Сначала мы с помощью типов определяем множество всех возможных значений. Значения – это деревья в узлах которых записаны конструкторы, которые мы определяем в типах. Так например мы можем определить тип:

```
data Nat = Zero | Succ Nat
```

Этим типом мы определяем множество допустимых значений. В данном случае это цепочки конструкторов `Succ`, которые заканчиваются конструктором `Zero`:

```
Zero, Succ Zero, Succ (Succ Zero), ...
```

Затем начинаем давать им новые имена, создавая константы (простые имена-синонимы)

```
zero    = Zero
one     = Succ zero
two     = Succ one
```

и функции (составные имена-синонимы):

```
foldNat :: a -> (a -> a) -> Nat -> a
foldNat z s Zero      = z
foldNat z s (Succ n)  = s (foldNat z s n)
```

```
add a = foldNat a Succ
mul a = foldNat one (add a)
```

Затем мы передаём нашу программу на проверку компилятору. Мы просим у него проверить не создаём ли мы случайно какие-нибудь бессмысленные выражения. Бессмысленные потому, что они пытаются создать значение, которое не вписывается в наши типы. Например если мы где-нибудь попробуем составить выражение:


```
add Zero mul
```

Компилятор напомним нам о том, что мы пытаемся подставить функцию `mul` на место обычного значения типа `Nat`. Тогда мы исправим выражение на:

```
add Zero two
```

Компилятор согласится. И передаст выражение вычислителю. И тут мы говорили, что вычислитель начинает проводить расшифровку нашего описания. Он подставляет на место синонимов их определения, правые части из уравнений. Этот процесс мы называли *редукцией*. Вычислитель видит два синонима и одно значение. С какого синонима начать? С `add` или `two`?

Стратегии вычислений

Этот вопрос приводит нас к понятию стратегии вычислений. Поскольку вычисляем мы только константы, то наше выражение также можно представить в виде дерева. Только теперь у нас в узлах записаны не только конструкторы, но и синонимы. Процесс редукции можно представить как процесс очистки такого дерева от синонимов. Посмотрим на дерево нашего значения:

Оказывается у нас есть две возможности очистки синонимов.

Снизу-вверх

начинаем с листьев и убираем все синонимы в листьях дерева выражения. Как только в данном узле и всех дочерних узлах остались одни конструкторы можно переходить на уровень выше. Так мы поднимаемся выше и выше пока не дойдём до корня.

Сверху-вниз

начинаем с корня, самого внешнего синонима и заменяем его на определение (с помощью уравнения на правую часть от знака равно), если на верху снова окажется синоним, мы опять заменим его на определение и так пока на верху не появится конструктор, тогда мы спустимся в дочерние деревья и будем повторять эту процедуру пока не дойдём до листьев дерева.

Посмотрим как каждая из стратегий будет редуцировать наше выражение. Начнём со стратегии от листьев к корню (снизу-вверх):

```
add Zero two
-- видим два синонима add и two
-- раскрываем two, ведь он находится ниже всех синонимов
=> add Zero (Succ one)
-- ниже появился ещё один синоним, раскроем и его
=> add Zero (Succ (Succ zero))
-- появился синоним zero раскроем его
=> add Zero (Succ (Succ Zero))
-- все узлы ниже содержат конструкторы, поднимаемся вверх до синонима
-- заменяем add на его правую часть
=> foldNat Succ Zero (Succ (Succ Zero))
-- самый нижний синоним foldNat, раскроем его
-- сопоставление с образцом проходит во втором уравнении для foldNat
=> Succ (foldNat Succ Zero (Succ Zero))
-- снова раскрываем foldNat
=> Succ (Succ (foldNat Zero Zero))
-- снова раскрываем foldNat, но на этот раз нам подходит
-- первое уравнение из определения foldNat
=> Succ (Succ Zero)
-- синонимов больше нет можно вернуть значение
-- результат:
Succ (Succ Zero)
```

В этой стратегии для каждой функции мы сначала вычисляем до конца все аргументы, потом подставляем расшифрованные значения в определение функции.

Теперь посмотрим на вычисление от корня к листьям (сверху-вниз):

```
add Zero two
-- видим два синонима add и two, начинаем с того, что ближе всех к корню
=> foldNat Succ Zero two
-- теперь выше всех foldNat, раскроем его
```

Но для того чтобы раскрыть foldNat нам нужно узнать какое уравнение выбрать для этого нам нужно понять какой конструктор находится в корне у второго аргумента, если это **Zero**, то мы выберем первое уравнение, а если это **Succ**, то второе:

```
-- в уравнении для foldNat видим декомпозицию по второму
-- аргументу. Узнаем какой конструктор в корне у two
=> foldNat Succ Zero (Succ one)
-- Это Succ нам нужно второе уравнение:
=> Succ (foldNat Succ Zero one)
```

```
-- В корне мы получили конструктор, можем спуститься ниже.
-- Там мы видим foldNat, для того чтобы раскрыть его нам
-- снова нужно понять какой конструктор в корне у второго аргумента:
=> Succ (foldNat Succ Zero (Succ zero))
-- Это опять Succ переходим ко второму уравнению для foldNat
=> Succ (Succ (foldNat Succ Zero zero))
-- Снова раскрываем второй аргумент у foldNat
=> Succ (Succ (foldNat Succ Zero Zero))
-- Ага это Zero, выбираем первое уравнение
=> Succ (Succ Zero)
-- Синонимов больше нет можно вернуть значение
-- результат:
Succ (Succ Zero)
```

В этой стратегии мы всегда раскрываем самый верхний уровень выражения, можно представить как мы вытягиваем конструкторы от корня по цепочке. У этих стратегий есть специальные имена:

- вычисление *по значению* (call by value), когда мы идём от листьев к корню.
- вычисление *по имени* (call by name), когда мы идём от корня к листьям.

Отметим, что стратегию вычисления по значению также принято называть *энергичными вычислениями* (eager evaluation) или *аппликативной* (applicative) стратегией редукции. Вычисление по имени также принято называть *нормальной* (normal) стратегией редукции.

Преимущества и недостатки стратегий

В чём преимущества, той и другой стратегии.

Если выражение вычисляется полностью, первая стратегия более эффективна по расходу памяти.

Вычисляется полностью означает все компоненты выражения участвуют в вычислении. Например то выражении, которое мы рассмотрели так подробно, вычисляется полностью. Приведём пример выражения, при вычислении которого нужна лишь часть аргументов, для этого определим функцию:

```
isZero :: Nat -> Bool
isZero Zero    = True
isZero _       = False
```

Она проверяет является ли нулём данное число, теперь представим как будет вычисляться выражение, в той и другой стратегии:

```
isZero (add Zero two)
```

Первая стратегия сначала вычислит все аргументы у add потом расшифрует add и только в самом конце доберётся до isZero. На это уйдёт восемь шагов (семь на вычисление add Zero two). В то время как вторая стратегия начнёт с isZero. Для вычисления isZero ей потребуется узнать какой конструктор в корне у выражения add Zero two. Она узнает это за два шага. Итого три шага. Налицо экономия усилий.

Почему вторая стратегия экономит память? Поскольку мы всегда вычисляем аргументы функции, мы можем не хранить описания в памяти а сразу при подстановке в функцию начинать редукцию. Эту ситуацию можно понять на таком примере, посчитаем сумму чисел от одного до четырёх с помощью такой функции:

```
sum :: Int -> [Int] -> Int
sum []      res = res
sum (x:xs)  res = sum xs (res + x)
```

Посмотрим на то как вычисляет первая стратегия, с учётом того что мы вычисляем значения при подстановке:

```
=> sum [1,2,3,4] 0
=> sum [2,3,4]   (0 + 1)
=> sum [2,3,4]   1
=> sum [3,4]     (1 + 2)
=> sum [3,4]     3
=> sum [4]       (3+3)
=> sum [4]       6
=> sum []        (6+4)
=> sum []        10
=> 10
```

Теперь посмотрим на вторую стратегию:

```
=> sum [1,2,3,4] 0
=> sum [2,3,4]   0+1
=> sum [3,4]     (0+1)+2
=> sum [4]       ((0+1)+2)+3
=> sum []        (((0+1)+2)+3)+4
=> (((0+1)+2)+3)+4
=> ((1+2)+3)+4
```

```
=> (3+3)+4
=> 6+4
=> 10
```

А теперь представьте, что мы решили посчитать сумму чисел от 1 до миллиона. Сколько вычислений нам придётся накопить! В этом недостаток второй стратегии. Но есть и ещё один недостаток, рассмотрим выражение:

```
(\x -> add (add x x) x) (add Zero two)
```

Первая стратегия сначала редуцирует выражение `add Zero two` в то время как вторая подставит это выражение в функцию и утроит свою работу!

Но у второй стратегии есть одно очень веское преимущество, она может вычислять больше выражений чем вторая. Определим значение бесконечность:

```
infinity    :: Nat
infinity    = Succ infinity
```

Это рекурсивное определение, если мы попытаемся его распечатать мы получим бесконечную последовательность `Succ`. Чем не бесконечность? Теперь посмотрим на выражение:

```
isZero infinity
```

Первая стратегия захлебнётся, вычисляя аргумент функции `isZero`, в то время как вторая найдёт решение за два шага.

Подведём итоги. Плюсы вычисления по значению:

- Эффективный расход памяти в том случае если все составляющие выражения участвуют в вычислении.
- Она не может дублировать вычисления, как стратегия вычисления по имени.

Плюсы вычисления по имени:

- Меньше вычислений в том случае, если при вычислении выражения участвует лишь часть составляющих.

- Большая выразительность. Мы можем вычислить больше значений.

Какую из них выбрать? В Haskell пошли по второму пути. Всё-таки преимущество выразительности языка оказалось самым существенным. Но для того чтобы избежать недостатков стратегии вычисления по имени оно было модифицировано. Давайте посмотрим как.

Вычисление по необходимости

Вернёмся к выражению:

```
(\x -> add (add x x) x) (add Zero two)
```

Нам нужно как-то рассказать функции о том, что имя `x` в её теле указывает на одно и то же значение. И если в одном из `x` значение будет вычислено переиспользовать эти результаты в других `x`.

Вместо значения мы будем передавать в функцию *ссылку* на область памяти, которая содержит рецепт получения этого значения.

Напомню, что мы по-прежнему вычисляем значение сверху вниз, сейчас мы просто хотим избавиться от проблемы дублирования.

Вернитесь к примеру с вычислением по имени и просмотрите его ещё раз. Обратите внимание на то, что значения вычислялись лишь при сопоставлении с образцом. Мы вычисляем верхний конструктор аргумента лишь для того, чтобы понять какое уравнение для `foldNat` выбрать. Теперь мы будем хранить ссылку на `(add Zero two)` в памяти и как только, внешняя функция запросит верхний конструктор мы обновим значение в памяти новым вычисленным до корневого конструктора значением. Если в любом другом месте функции мы вновь обратимся к значению, мы не будем его перевычислять, а сразу вернём конструктор. Посмотрим как это происходит на примере:

выражение	память:
<pre>(\x -> add (add x x) x) M</pre>	<pre>M = (add Zero two)</pre>
<pre>-- подставим ссылку в тело функции</pre>	
<pre>=> add (add M M) M</pre>	
<pre>-- раскроем самый верхний синоним</pre>	
<pre>=> foldNat (add M M) Succ M</pre>	

<pre>-- для foldNat узнаем верхний конструктор -- последнего аргумента (пропуская -- промежуточные шаги, такие же как выше) => -- по M выбираем второе уравнение => Succ (foldNat (add M M) Succ M1) -- запросим следующий верхний конструктор: => -- по M1 выбираем второе уравнение => Succ (Succ (foldNat (add M M) Succ M2)) -- теперь для определения уравнения foldNat -- раскроем M2 => -- выбираем первое уравнение для foldNat: => Succ (Succ (add M M)) -- раскрываем самый верхний синоним: => Succ (Succ (foldNat M Succ M)) -- теперь, поскольку M уже вычислялось, в -- памяти уже записан верхний конструктор, -- мы знаем, что это Succ и выбираем второе -- уравнение: => Succ (Succ (Succ (foldNat M Succ M1))) -- и M1 тоже уже вычислялось, сразу -- выбираем второе уравнение => Succ (Succ (Succ (Succ (foldNat M Succ M2)))) -- M2 вычислено, идём на первое уравнение => Succ (Succ (Succ (Succ (Succ M)))) -- далее остаётся только подставить уже -- вычисленные значения M -- и вернуть значение.</pre>	<pre>M = Succ M1 M1 = foldNat Succ Zero one M = Succ M1 M1 = Succ M2 M2 = foldNat Succ Zero zero M = Succ M1 M1 = Succ M2 M2 = Zero -----+ -----+</pre>
---	---

Итак подставляется не значение а ссылка на него, вычисленная часть значения используется сразу в нескольких местах. Эта стратегия редукции называется вычислением *по необходимости* (call by need) или *ленивой* стратегией вычислений (lazy evaluation).

Теперь немного терминологии. Значение может находиться в четырёх состояниях:

- Нормальная форма (normal form, далее НФ), когда оно полностью вычислено (нет синонимов);
- Слабая заголовочная НФ (weak head NF, далее СЗНФ), когда известен хотя бы один верхний конструктор;

- Отложенное вычисление (thunk), когда известен лишь рецепт вычисления;
- Дно (bottom, часто рисуют как \bot), когда известно, что значение не определено.

Вы могли понаблюдать за значением в первых трёх состояниях на примере выше. Но что такое \bot ? Вспомним определение для функции извлечения головы списка head:

```
head :: [a] -> a
head (a:_) = a
head []    = error "error: empty list"
```

Второе уравнение возвращает \bot . У нас есть две функции, которые возвращают это “значение”:

```
undefined :: a
error      :: String -> a
```

Первая – это \bot в чистом виде, а вторая не только возвращает неопределённое значение, но и приводит к выводу на экран сообщения об ошибке. Обратите внимание на тип этих функций, результат может быть значением любого типа. Это наблюдение приводит нас к ещё одной тонкости. Когда мы определяем тип:

```
data Bool      = False | True
data Maybe a   = Nothing | Just a
```

На самом деле мы пишем:

```
data Bool      = undefined | False | True
data Maybe a   = undefined | Nothing | Just a
```

Компилятор автоматически прибавляет ещё одно значение к любому определённому пользователем типу. Такие типы называют *поднятыми* (lifted type). А значения таких типов принято называть *запакованными* (boxed). Не запакованное (unboxed) значение – это простое примитивное значение. Например целое или действительное число в том виде, в котором оно хранится на компьютере. В Haskell даже числа “запакованы”. Поскольку нам необходимо, чтобы undefined могло возвращать в том числе и значение типа Int:

```
data Int = undefined | I# Int#
```


Тип `Int#` – это низкоуровневое представление ограниченного целого числа. Принято писать не запакованные типы с решёткой на конце. `I#` – это конструктор. Нам приходится запаковывать значения ещё и потому, что значение может принимать несколько состояний (в зависимости от того, насколько оно вычислено), всё это ведёт к тому, что у нас хранится не просто значение, а значение с какой-то дополнительной информацией, которая зависит от конкретной реализации языка Haskell.

Мы решили проблему дублирования вычислений, но наше решение усугубило проблему расхода памяти. Ведь теперь мы храним не просто значения, но ещё и дополнительную информацию, которая отвечает за проведение вычислений. Эта проблема может проявляться в очень простых задачах. Например попробуем вычислить сумму чисел от одного до миллиарда:

```
sum [1 .. 1e9]  
<interactive>: out of memory (requested 2097152 bytes)
```

Интуитивно кажется, что для решения этой задачи нам нужно лишь две ячейки памяти. В одной мы будем постоянно прибавлять к значению единицу, пока не дойдём до миллиарда, так мы последовательно будем получать элементы списка, а в другой мы будем хранить значение суммы. Мы начнём с нуля и будем прибавлять значения первой ячейки. У ленивой стратегии другое мнение на этот счёт. Если вы вернётесь к примеру выше, то заметите, что `sum` копит отложенные выражения до самого последнего момента. Поскольку память ограничена, такой момент не наступает. Как нам быть? В Haskell по умолчанию все вычисления проводятся по необходимости, но предусмотрены и средства для имитации вычисления по значению. Давайте посмотрим на них.

Аннотации строгости

Языки с ленивой стратегией вычислений называют не строгими (non-strict), а языки с энергичной стратегией вычислений соответственно – строгими.

Принуждение к СЗНФ с помощью seq

Мы говорили о том, что при вычислении по имени значения вычисляются только при сопоставлении с образцом или в **case**-выражениях. Есть специальная функция `seq`, которая форсирует приведение к СЗНФ:

```
seq :: a -> b -> b
```

Она принимает два аргумента, при выполнении функции первый аргумент приводится к СЗНФ и *затем* возвращается второй. Вернёмся к примеру с `sum`. Привести к СЗНФ число – означает вычислить его полностью. Определим функцию `sum'`, которая перед рекурсивным вызовом вычисляет промежуточный результат:

```
sum' :: Num a => [a] -> a
sum' = iter 0
  where iter res []      = res
        iter res (a:as) = let res' = res + a
                          in  res' `seq` iter res' as
```

Сохраним результат в отдельном модуле `Strict.hs` и попробуем теперь вычислить значение, придётся подождать:

```
Strict> sum' [1 .. 1e9]
```

И мы ждём, и ждём, и ждём. Но переполнения памяти не происходит. Это хорошо. Но давайте прервём вычисления. Нажмём `ctrl+c`. Функция `sum'` вычисляется, но вычисляется очень медленно. Мы можем существенно ускорить её, если *скомпилируем* модуль `Strict`. Для компиляции модуля переключимся в его текущую директорию и вызовем компилятор `ghc` с флагом `--make`:

```
ghc --make Strict
```

Появились два файла `Strict.hi` и `Strict.o`. Теперь мы можем загрузить модуль `Strict` в интерпретатор и сравнить выполнение двух функций:

```
Strict> sum' [1 .. 1e6]
5.000005e11
(0.00 secs, 89133484 bytes)
Strict> sum [1 .. 1e6]
5.000005e11
(0.57 secs, 142563064 bytes)
```

Обратите внимание на прирост скорости. Умение понимать в каких случаях стоит ограничить лень очень важно. И в программах на Haskell тоже. Также компилировать модули можно из интерпретатора. Для этого воспользуемся командой `:!` , она выполняет системные команды в интерпретаторе `ghci`:

```
Strict> :! ghc --make Strict
[1 of 1] Compiling Strict          ( Strict.hs, Strict.o )
```

Отметим наличие специальной функции применения, которая просит перед применением привести аргумент к СЗНФ, эта функция определена в `Prelude`:

```
($!) :: (a -> b) -> a -> b
f $! a = a `seq` f a
```

С этой функцией мы можем определить функцию `sum` так:

```
sum' :: Num a => [a] -> a
sum' = iter 0
  where iter res []      = res
        iter res (a:as) = flip iter as $! res + a
```

Функции с хвостовой рекурсией

Определим функцию, которая не будет лениться при вычислении произведения чисел, мы назовём её `product'`:

```
product' :: Num a => [a] -> a
product' = iter 1
  where iter res []      = res
        iter res (a:as) = let res' = res * a
                          in res' `seq` iter res' as
```

Смотрите функция `sum` изменилась лишь в двух местах. Это говорит о том, что пора задуматься о том, а нет ли такой общей функции, которая включает в себя и то и другое поведение. Такая функция есть и называется она `foldl'`, вот её определение:

```
foldl' :: (a -> b -> a) -> a -> [b] -> a
foldl' op init = iter init
  where iter res []      = res
        iter res (a:as) = let res' = res `op` a
                          in res' `seq` iter res' as
```

Мы вынесли в аргументы функции бинарную операцию и начальное значение. Всё остальное осталось прежним. Эта функция живёт в модуле `Data.List`. Теперь мы можем определить функции `sum'` и `prod'`:

```
sum'      = foldl' (+) 0
product'  = foldl' (*) 1
```

Также в `Prelude` определена функция `foldl`. Она накапливает значения в аргументе, но без принуждения вычислять промежуточные результаты:

```
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl op init = iter init
  where iter res []      = res
        iter res (a:as) = iter (res `op` a) as
```

Такая функция называется функцией с *хвостовой рекурсией* (tail-recursive function). Рекурсия хвостовая тогда, когда рекурсивный вызов функции является последним действием, которое выполняется в функции. Посмотрите на второе уравнение функции `iter`. Мы вызываем функцию `iter` рекурсивно последним делом. В языках с вычислением по значению часто хвостовая рекурсия имеет преимущество за счёт экономии памяти (тот момент который мы обсуждали в самом начале). Но как видно из этого раздела в ленивых языках это не так. Библиотечная функция `sum` будет накапливать выражения перед вычислением с риском исчерпать всю доступную память, потому что она определена через `foldl`.

Тонкости применения `seq`

Хочу подчеркнуть, что функция `seq` не вычисляет свой первый аргумент полностью. Первый аргумент не приводится к нормальной форме. Мы лишь просим вычислитель узнать какой конструктор находится в корне у данного выражения. Например в выражении `isZero $! infinity` знак `$!` ничем не отличается от простого применения мы и так будем приводить аргумент `infinity` к СЗНФ, когда нам понадобится узнать какое из уравнений для `isZero` выбрать, ведь в аргументе функции есть сопоставление с образцом.

Посмотрим на один типичный пример. Вычисление среднего для списка чисел. Среднее равно сумме всех элементов списка, разделённой на длину списка. Для того чтобы вычислить значение за один проход мы будем одновременно вычислять и сумму элементов и значение длины. Также мы понимаем, что нам не нужно откладывать вычисления, воспользуемся функцией `foldl'`:

```
mean :: [Double] -> Double
mean = division . foldl' count (0, 0)
  where count (sum, leng) a = (sum+a, leng+1)
        division (sum, leng) = sum / fromIntegral leng
```

Проходим по списку, копируем сумму в первом элементе пары и длину во втором. В самом конце делим первый элемент на второй. Обратите внимание на функцию `fromIntegral` она преобразует значения из целых чисел, в какие-нибудь другие из класса `Num`. Сохраним это определение в модуле `Strict` скомпилируем модуль и загрузим в интерпретатор, не забудьте импортировать модуль `Data.List`, он нужен для функции `foldl'`. Посмотрим, что у нас получилось:

```
Prelude Strict> mean [1 .. 1e7]
5000000.5
(49.65 secs, 2476557164 bytes)
```

Получилось очень медленно, странно ведь порядок этой функции должен быть таким же что и у `sum'`. Посмотрим на скорость `sum'`:

```
Prelude Strict> sum' [1 .. 1e7]
5.0000005e13
(0.50 secs, 881855740 bytes)
```

В 100 раз быстрее. Теперь представьте, что у нас 10 таких функций как `mean` они разбросаны по всему коду и делают своё чёрное ленивое дело. Причина такого поведения кроется в том, что мы опять завернули значение в другой тип, на этот раз в пару. Когда вычислитель дойдёт до `seq`, он остановится на выражении `(think, think)` вместо двух чисел. Он вновь будет накапливать отложенные вычисления, а не значения.

Перепишем `mean`, теперь мы будем вычислять значения пары по отдельности и попросим вычислитель привести к СЗНФ каждое из них перед вычислением итогового значения:

```

mean' :: [Double] -> Double
mean' = division . iter (0, 0)
  where iter res [] = res
        iter (sum, leng) (a:as) =
          let s = sum + a
              l = leng + 1
          in s `seq` l `seq` iter (s, l) as

division (sum, leng) = sum / fromIntegral leng

```

Такой вот монстр. Функция `seq` право ассоциативна поэтому скобки будут группироваться в нужном порядке. В этом определении мы просим вычислитель привести к СЗНФ *числа*, а не пары чисел, как в прошлой версии. Для чисел СЗНФ совпадает с НФ, и всё должно пройти гладко, но сохраним это определение и проверим результат:

```

Prelude Strict> :! ghc --make Strict
[1 of 1] Compiling Strict          ( Strict.hs, Strict.o )
Prelude Strict> :load Strict
Ok, modules loaded: Strict.
(0.00 secs, 0 bytes)
Prelude Strict> mean' [1 .. 1e7]
5000000.5
(0.65 secs, 1083157384 bytes)

```

Получилось! Скорость чуть хуже чем у `sum'`, но не в сто раз.

Энергичные образцы

В GHC предусмотрены специальные обозначения для принудительного приведения выражения к СЗНФ. Они не входят в стандарт языка Haskell, поэтому для того, чтобы воспользоваться ими, нам необходимо подключить их. Расширения подключаются с помощью специального комментария в самом начале модуля:

```
{-# LANGUAGE BangPatterns #-}
```

Эта запись активирует расширение языка с именем `BangPatterns`. Ядро языка Haskell фиксировано стандартом, но каждый разработчик компилятора может вносить свои дополнения. Они подключаются через директиву `LANGUAGE`:

```

{-# LANGUAGE
  Расширение1,
  Расширение2,
  Расширение3 #-}

```

Мы заключаем директиву в специальные комментарии с решёткой, говорим `LANGUAGE` а затем через запятую перечисляем имена расширений, которые нам понадобятся. Расширения активны только в рамках данного модуля. Например если мы импортируем функции из модуля, в котором включены расширения, то эти расширения не распространяются дальше на другие модули. Такие комментарии с решёткой называют *прагмами* (pragma).

Нас интересует расширение `BangPatterns` (bang – восклицательный знак, вы сейчас поймёте почему оно так называется). Посмотрим на функцию, которая использует энергичные образцы:

```
iter (!sum, !leng) a = (step + a, leng + 1)
```

В декомпозиции пары перед переменными у нас появились восклицательные знаки. Они говорят вычислителю о том, чтобы он так уж и быть сделал ещё одно усилие и заглянул в корень значений переменных, которые были переданы в эту функцию.

Вычислитель говорит ладно-ладно сделаю. А там числа! И получается, что они не накапливаются. С помощью энергичных образцов мы можем переписать функцию `mean'` через `foldl'`, а не выписывать её целиком:

```
mean'' :: [Double] -> Double
mean'' = division . foldl' iter (0, 0)
  where iter (!sum, !leng) a = (sum + a, leng + 1)
        division (sum, leng) = sum / fromIntegral leng
```

Проверим в интерпретаторе

```
*Strict> :! ghc --make Strict
[1 of 1] Compiling Strict          ( Strict.hs, Strict.o )
*Strict> :l Strict
Ok, modules loaded: Strict.
(0.00 secs, 581304 bytes)
Prelude Strict> mean'' [1 .. 1e7]
5000000.5
(0.78 secs, 1412862488 bytes)
Prelude Strict> mean' [1 .. 1e7]
5000000.5
(0.65 secs, 1082640204 bytes)
```

Функция работает чуть медленнее, чем исходная версия, но не сильно.

Энергичные типы данных

Расширение **BangPatterns** позволяет указывать какие значения привести к СЗНФ не только в образцах, но и в типах данных. Мы можем создать тип:

```
data P a b = P !a !b
```

Этот тип обозначает пару, элементы которой обязаны находиться в СЗНФ. Теперь мы можем написать ещё один вариант функции поиска среднего:

```
mean''' :: [Double] -> Double
mean''' = division . foldl' iter (P 0 0)
  where iter (P sum leng) a = P (sum + a) (leng + 1)
        division (P sum leng) = sum / fromIntegral leng
```

Пример ленивых вычислений

У вас может сложиться ошибочное представление, что ленивые вычисления созданы только для того, чтобы с ними бороться. Пока мы рассматривали лишь недостатки, вскользь упомянув о преимуществе выразительности. Ленивые вычисления могут и экономить память! Мы можем строить огромные промежуточные данные, обрабатывать их разными способами при условии, что в конце программы нам потребуется лишь часть этих данных или конечный алгоритм будет накапливать определённую статистику.

Рассмотрим такое выражение:

```
let longList = produce x
in sum' $ filter p $ map f longList
```

Функция `produce` строит огромный список промежуточных данных. Далее мы преобразуем эти данные функцией `f` и фильтруем их предикатом `p`. Всё это делается для того, чтобы посчитать сумму всех элементов в списке. Посмотрим как повела бы себя в такой ситуации энергичная стратегия вычислений. Сначала был бы вычислен список `longList`, причём полностью. Затем все элементы были бы преобразованы функцией `f`. У нас в памяти уже два огромных списка. Теперь мы фильтруем весь список и в самом конце суммируем. Было

бы очень плохо заставляя энергичный вычислитель редуцировать такое выражение.

А в это время ленивый вычислитель поступит так. Сначала всё выражение будет сохранено в виде описания, затем он скажет разверну сначала `sum'`, эта функция запросит первый элемент списка, что приведёт к вызову `filter`. Фильтр будет запрашивать следующий элемент списка у подчинённых ему функций до тех пор, пока предикат `p` не вернёт `True` на одном из них. Всё это время функция `map` будет вытягивать из `produce` по одному элементу. Причём память, выделенная на промежуточные не нужные значения (на них `p` вернул `False`) будет переиспользована. Как только `sum'` прибавит первый элемент, она запросит следующий, проснётся фильтр и так далее. Вся функция будет работать в постоянном ограниченном объёме памяти, который не зависит от величины списка `longList`!

Примерам ленивых вычислений будет посвящена отдельная глава, а пока приведём один пример. Найдём корень уравнения с помощью метода неподвижной точки. У нас есть функция `f :: a -> a`, и нам нужно найти решение уравнения:

```
f x = x
```

Можно начать с какого-нибудь стартового значения, и подставлять, подставлять, подставлять его в `f` до тех пор, пока значение не перестанет изменяться. Так мы найдём решение.

```
x1 = f x0
x2 = f x1
x3 = f x2
...
до тех пор пока abs (x[N] - x[N-1]) <= eps
```

Первое наблюдение: функция принимает не произвольные значения, а те для которых имеет смысл операции: минус, поиск абсолютного значения и сравнение на больше/меньше. Тип нашей функции:

```
f :: (Ord a, Num a) => a -> a
```

Ленивые вычисления позволяют нам отделить шаг генерации решений, от шага проверки сходимости. Сначала мы сделаем список всех подстановок функции `f`, а затем найдём в этом списке два соседних

элемента расстояние между которыми достаточно мало. Итак первый шаг, генерируем всю последовательность:

```
xNs = iterate f x0
```

Мы воспользовались стандартной функцией `iterate` из `Prelude`.

Теперь ищем два соседних числа:

```
converge :: (Ord a, Num a) => a -> [a] -> a
converge eps (a:b:xs)
  | abs (a - b) <= eps   = a
  | otherwise            = converge eps (b:xs)
```

Поскольку список бесконечный мы можем не проверять случаи для пустого списка. Итоговое решение:

```
roots :: (Ord a, Num a) => a -> a -> (a -> a) -> a
roots eps x0 f = converge eps $ iterate f x0
```

За счёт ленивых вычислений функции `converge` и `iterate` работают синхронно. Функция `converge` запрашивает новое значение и `iterate` передаёт его, но только одно! Найдём решение какого-нибудь уравнения. Запустим интерпретатор. Мы ленимся и не создаём новый модуль для такой “большой” функции. Определяем её сразу в интерпретаторе.

```
Prelude> let converge eps (a:b:xs) = if abs (a-b)<=eps then a else converge
eps (b:xs)
Prelude> let roots eps x0 f = converge eps $ iterate f x0
```

Найдём корень уравнения:

$x(x-2) = 0$

$x^2 - 2x = 0$

$\frac{1}{2} x^2 = x$

```
Prelude> roots 0.001 5 (\x -> x*x/2)
```

Метод завис, остаётся только нажать `ctrl+c` для остановки. На самом деле есть одно условие для сходимости метода. Метод сойдётся, если модуль производной функции `f` меньше единицы. Иначе есть возможность, что мы будем бесконечно генерировать новые подстановки. Вычислим производную нашей функции:

$\frac{d}{dx} \frac{1}{2} x^2 = x$

Нам следует ожидать решения в интервале от минус единицы до единицы:

```
Prelude> roots 0.001 0.5 (\x -> x*x/2)
3.0517578125e-5
```

Мы нашли решение, корень равен нулю. В этой записи `Ne-5` означает $N \cdot 10^{-5}$

Краткое содержание

В этой главе мы узнали о том как происходят вычисления в Haskell. Мы узнали, что они ленивые. Всё вычисляется как можно позже и как можно меньше. Такие вычисления называются вычислениями по необходимости.

Также мы узнали о вычислениях по значению и вычислениях по имени.

- В *вычислениях по значению* редукция проводится от листьев дерева выражения к корню
- В *вычислениях по имени* редукция проводится от корня дерева выражения к листьям.

Вычисление по необходимости является улучшением вычисления по имени. Мы не дублируем выражения во время применения. Мы сохраняем значения в памяти и подставляем в функцию ссылки на значения. После вычисления значения происходит его обновление в памяти. Так если в одном месте выражение уже было вычислено и мы обратимся к нему по ссылке из другого места, то мы не будем перевычислять его, а просто считаем готовое значение.

Мы познакомились с терминологией процесса вычислений. Выражение может находиться в *нормальной форме*. Это значит что оно вычислено. Может находиться в *слабой заголовочной нормальной форме*. Это значит, что мы знаем хотя бы один конструктор в корне выражения. Также возможно выражение ещё не вычислялось, тогда оно является *отложенным* (thunk).

Суть ленивых вычислений заключается в том, что они происходят синхронно. Если у нас есть композиция двух функций:

$$g \circ f \ x$$

Внутренняя функция f не начнёт вычисления до тех пор пока значения не понадобятся внешней функции g . О последствиях этого мы остановимся подробнее в отдельной главе. Значения могут потребоваться только при сопоставлении с образцом. Когда мы хотим узнать какое из уравнений нам выбрать.

Иногда ленивые вычисления не эффективны по расходу памяти. Это происходит когда выражение состоит из большого числа подвыражений, которые будут вычислены в любом случае. В Haskell у нас есть способы борьбы с ленью. Это функция `seq`, энергичные образцы и энергичные типы данных.

Функция `seq`:

```
seq :: a -> b -> b
```

Сначала приводит к слабой заголовочной форме свой первый аргумент, а затем возвращает второй. Взрывные образцы выполняют те же функции, но они используются в декомпозиции аргументов или в объявлении типа.

Упражнения

- Потренируйтесь в понимании того как происходят ленивые вычисления. Вычислите на бумаге следующие выражения (если это возможно):
 - `sum $ take 3 $ filter (odd . fst) $ zip [1 ..] [1, undefined, 2, undefined, 3, undefined, undefined]`
 - `take 2 $ foldr (+) 0 $ map Succ $ repeat Zero`
 - `take 2 $ foldl (+) 0 $ map Succ $ repeat Zero`
- Функция `seq` приводит первый аргумент к СЗНФ, убедитесь в этом на таком эксперименте. Определите тип:

```
data TheDouble = TheDouble { runTheDouble :: Double }
```

Он запаковывает действительные числа в конструктор.

Определите для этого типа экземпляра класса `Num` и посмотрите как быстро будет работать функция `sum` на таких числах. Как изменится скорость если мы заменим в определении типа `data` на `newtype`? как изменится скорость, если мы вернём `data`, но сделаем тип `TheDouble` энергичным? Поэкспериментируйте.

- Посмотрите на приведение к СЗНФ в энергичных типах данных. Определите два типа:

```
data Strict a = Strict !a
data Lazy  a = Lazy  a
```

И повычисляйте в интерпретаторе различные значения с `undefined`, `const`, `($!)` и `seq`:

```
> seq (Lazy undefined) "Hi"
> seq (Strict undefined) "Hi"
> seq (Lazy (Strict undefined)) "Hi"
> seq (Strict (Strict (Strict undefined))) "Hi"
```

- Посмотрите на такую функцию вычисления суммы всех чётных и нечётных чисел в списке.

```
sum2 :: [Int] -> (Int, Int)
sum2 = iter (0, 0)
      where iter c []      = c
            iter c (x:xs) = iter (tick x c) xs

tick :: Int -> (Int, Int) -> (Int, Int)
tick x (c0, c1) | even x    = (c0, c1 + 1)
                | otherwise = (c0 + 1, c1)
```

Эта функция очень медленная. Кто-то слишком много ленится. Узнайте кто, и ускорьте функцию.

Реализация Haskell в GHC

На момент написания этой книги основным компилятором Haskell является GHC. Остальные конкуренты отстают очень сильно. Отметим компилятор Hugs (его хорошо использовать для демонстрации Haskell на чужом компьютере, если вы не хотите устанавливать тяжёлый GHC). В этой главе мы обзорно рассмотрим как язык Haskell реализован в GHC. GHC – как ни парадоксально это звучит, это самая успешная программа написанная на Haskell. GHC уже двадцать лет. Отметим основных разработчиков. Это Саймон Пейтон Джонс (Simon Peyton Jones) и Саймон Марлоу (Simon Marlow).

GHC состоит из трёх частей. Это сам компилятор, основные библиотеки языка (такие как Prelude) и низкоуровневая система вычислений (она отвечает за управление памятью, потоками, вычисление примитивных операций). Весь GHC кроме системы вычислений написан на Haskell. Система вычислений написана на C. Компилятор принимает набор файлов с исходным кодом (а также возможно объектных и интерфейсных файлов) и генерирует код низкого уровня. Система вычислений низкого уровня используется в этом коде как библиотека. Она статически подключается к любому нативному коду, который генерируется GHC. Далее мы сосредоточимся на изучении компилятора.

Но перед этим давайте освежим в памяти (или узнаем) несколько терминов. У нас есть код на Haskell, что значит перевести в код низкого уровня? Код низкого уровня представляет собой набор инструкций, которые изменяют значения в памяти компьютера. Изменение значений происходит с помощью базовых операций, которые выполняются в процессоре компьютера. Память компьютера представляет собой ленту ячеек. У каждой ячейки есть адрес и содержание. По адресу мы можем читать данные из ячейки и записывать их туда. Эти операции также выполняются с помощью инструкций. Мы будем делить память на стек (stack), кучу (heap) и регистры (registers).

Стек – это очередь с принципом работы “последним пришёл, первым ушёл”. Стек можно представить как стопку книг. У нас есть две

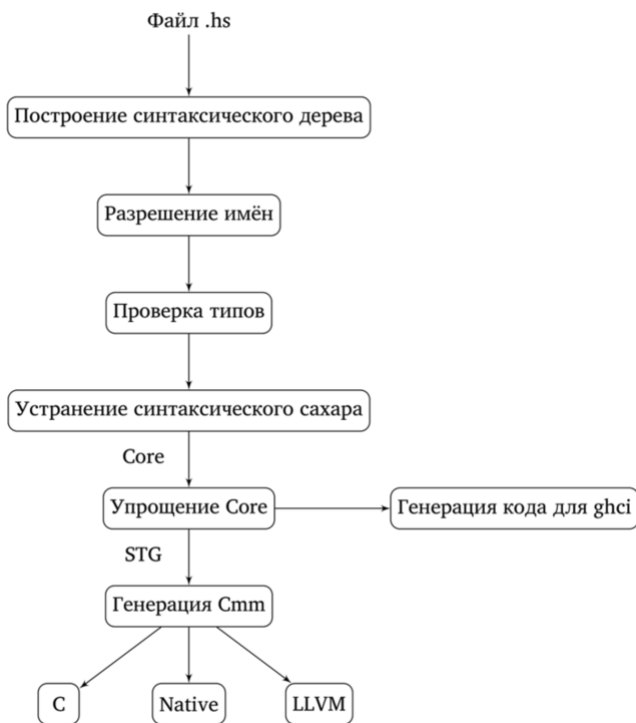
операции: положить книгу наверх, и снять верхнюю книгу. Стек очень удобен для переключения контекстов вычисления. Представьте, что у нас есть функция, которая внутри вызывает другую функцию, а та следующую. Находясь в верхней функции при заходе во вторую мы сохраняем контекст внешней функции в стеке. Контекст – это та информация, которая нужна нам для того, чтобы продолжить вычисления. Как только мы доходим до третьей функции, мы “кладем на стопку сверху” контекст второй функции, как только третья функция вычислена, мы обращаемся к стеку и снимаем с него контекст второй функции продолжаем вычислять и как только вторая функция заканчивается снова обращаемся к стеку. А там сверху уже лежит контекст самой первой функции. Мы можем продолжать вычисления. Так происходит вычисление вложенных функций в императивных языках программирования.

В куче мы храним разные данные. Данные бывают статическими (они нужны нам на протяжении выполнения всей программы) и динамическими (время жизни динамических данных заранее неизвестно, например это могут быть отложенные вычисления, мы не знаем когда ни нам понадобятся). У кучи также две операции: выделить блок памяти, эта операция принимает размер блока и возвращает адрес, по которому удалось выделить память, и освободить память по указанному адресу. Регистры находятся в процессоре. В них можно записывать и читать данные, при этом операции обращения к регистрам будут происходить очень быстро.

Посмотрим как GHC справляется с переводом процесса редукции синонимов на язык понятный нашему компьютеру. Язык обновления стека и кучи. Это большая и трудная глава, читайте не спеша. Если покажется совсем трудно – пропустите, вернётесь потом, когда захочется писать не только красивые, но и эффективные программы.

Этапы компиляции

Рассмотрим этапы компиляции программы.



Этапы компиляции

На первых трёх этапах происходит проверка ошибок. Сначала мы строим синтаксическое дерево программы. Если мы нигде не забыли скобки, не ошиблись в простановке ключевых слов, то этот этап успешно завершится. Далее мы приписываем ко всем функциям их полные имена. Дописываем перед всеми определениями имя модуля, в котором они определены. Обычно на этом этапе нам сообщают о том, что мы забыли определить какую-нибудь функцию, часто это связано с простой опечаткой. Следующий этап – самый важный. Происходит вывод типов для всех значений и проверка программы по типам. Блок кода, отвечающий за проверку типов, является самым большим в GHC. Haskell имеет очень развитую систему типов. Многих возможностей мы ещё не коснулись, часть из них мы рассмотрим в главе 17. Допустим, что мы исправили все ошибки связанные с типами, тогда компилятор начнёт переводить Haskell в Core.

Core – это функциональный язык программирования, который является сильно урезанной версией Haskell. Помните мы говорили, что в Haskell поддерживается несколько стилей (композиционный и декларативный). Что хорошо для программиста, не очень хорошо для

компилятора. Компилятор устраняет весь синтаксический сахар и выражает все определения через простейшие конструкции языка Core. Далее происходит серия оптимизаций языка Core. На дереве описания программы выполняется серия функций типа `Core -> Core`. Например происходит замена вызовов коротких функций на их правые части уравнений (встраивание или `inlining`), выражения, которые проводят декомпозицию в `case`-выражениях по константам, заменяются на соответствующие этим константам выражения. По требованию GHC может провести анализ строгости (`strictness analysis`). Он заключается в том, что GHC ищет аргументы функций, которые могут быть вычислены более эффективно с помощью вычисления по значению и расставляет аннотации строгости. И многие-многие другие оптимизации кода. Все они представлены в виде преобразования синтаксического дерева программы. Также этот этап называют упрощением программы.

После этого Core переводится на STG. Это функциональный язык, повторяющий Core. Он содержит дополнительную информацию, которая необходима низкоуровневым библиотекам на этапе вычисления программы. Затем из STG генерируется код языка `C--`. Это язык низкого уровня, “портируемый ассемблер”. На этом языке не пишут программы, он предназначен для автоматической генерации кода. Далее из него получают другие низкоуровневые коды. Возможна генерация C, LLVM и нативного кода (код, который выполняется операционной системой).

Язык STG

STG расшифровывается как `Spineless Tagless G-machine`. G-machine или Г-машина – это низкоуровневое описание процесса редукции графов (от `Graph`). Пока мы называли этот процесс редукцией синонимов. `Spineless` и `Tagless` – это термины специфичные для Г-машины, которая была придумана разработчиками GHC. `Tagless` относится к особому представлению объектов в куче (объекты представлены единообразно, так что им не нужен специальный тег для обозначения типа объекта), а `Spineless` относится к тому, что в отличие от машин-предшественников, которые описывают процесс

редукции графов виде последовательности инструкций, STG является небольшим функциональным языком. На представлен синтаксис языка STG. Синтаксис упрощён для чтения людьми. Несмотря на упрощения мы сможем посмотреть как происходит вычисление выражений.

Переменные	x, y, f, g	
Конструкторы	C	Объявлены в определениях типов
Литералы	$lit ::= i \mid d$	Незапакованные целые или действительные числа
Атомы	$a, v ::= lit \mid x$	Аргументы функций атомарны
Арность функции	$k ::= \bullet \mid n$	Арность неизвестна Арность известна $n \geq 1$
Выражения	$e ::= a$ $\mid f^k a_1 \dots a_n$ $\mid \oplus a_1 \dots a_n$ $\mid \text{let } x = obj \text{ in } e$ $\mid \text{case } e \text{ of } \{alt_1; \dots; alt_n\}$	Атом Вызов функции ($n \geq 1$) Вызов примитивной функции ($n \geq 1$) Выделение нового объекта obj в куче Приведение выражения e к СЗНФ
Альтернативы	$alt ::= C x_1 \dots x_n \rightarrow e$ $\mid x \rightarrow e$	Сопоставление с образцом ($n \geq 1$) Альтернатива по умолчанию
Объекты в куче	$obj ::= FUN(x_1 \dots x_n \rightarrow e)$ $\mid PAP(f a_1 \dots a_n)$ $\mid CON(C a_1 \dots a_n)$ $\mid THUNK e$ $\mid BLACKHOLE$	Функция арности $n \geq 1$ Частичное применение f может указывать только на FUN Полное применение конструктора ($n \geq 0$) Отложенное вычисление Используется только во время выполнения программы
Программа	$prog ::= f_1=obj_1; \dots; f_n=obj_n$	

Синтаксис STG

По синтаксису STG можно понять, какие выражения языка Haskell являются синтаксическим сахаром. Им просто нет места в языке STG. Например, не видим мы сопоставления с образцом. Оно как и **if**-выражения переписывается через **case**-выражения. Исчезли **where**-выражения. Конструкторы могут применяться только полностью, то есть для применения конструктора мы должны передать ему все аргументы. В STG **let**-выражения разделяют на не рекурсивные (**let**) и рекурсивные (**letrec**). Разделение проводится в целях оптимизации, мы же будем считать, что эти случаи описываются одной конструкцией.

На что стоит обратить внимание? Заметим, что функции могут принимать только атомарные значения (либо примитивные значения, либо переменные). В данном случае переменные указывают на объекты в куче. Так если в Haskell мы пишем:

```
foldr f (g x y) (h x)
```

В STG это выражение примет вид:

```
let gxy = THUNK (g x y)
    hx  = THUNK (h x)
in foldr f gxy hx
```

У функций появились степени. Что это? Степени указывают на арность функции, то есть на количество принимаемых аргументов. Количество принимаемых аргументов определяется по левой части функции. Поскольку в Haskell функции могут возвращать другие функции, очень часто мы не можем знать арность, тогда мы пишем `\bullet`.

Отметим два важных принципа вычисления на STG:

- Новые объекты создаются в куче *только* в **let**-выражениях
- Выражение приводится к СЗНФ *только* в **case**-выражениях

Выражение **let** `a = obj` **in** `e` означает добавь в кучу объект `obj` под именем `a` и затем вычисли `e`. Выражение **case** `e of` `{alt1;...;alt2}` означает узнай конструктор в корне `e` и продолжи вычисления в соответствующей альтернативе. Обратите внимание на то, что сопоставления с образцом в альтернативах имеет только один уровень вложенности. Также аргумент **case**-выражения в отличие от функции не обязан быть атомарным.

Для тренировки перепишем на STG пример из раздела про ленивые вычисления.

```
data Nat = Zero | Succ Nat
```

```
zero    = Zero
one      = Succ zero
two      = Succ one
```

```
foldNat :: a -> (a -> a) -> Nat -> a
foldNat z s Zero      = z
foldNat z s (Succ n)  = s (foldNat z s n)
```

```
add a = foldNat a Succ
mul a = foldNat one (add a)
```

```
exp = (\x -> add (add x x) x) (add Zero two)
```

Теперь в STG:

```
data Nat = Zero | Succ Nat
```

```
zero      = CON(Zero)
```

```
one       = CON(Succ zero)
```

```
two       = CON(Succ one)
```

```
foldNat = FUN( z s arg ->
    case arg of
        Zero    -> z
        Succ n  -> let next = THUNK (foldNat z s n)
                    in s next
    )
```

```
add       = FUN( a ->
    let succ = FUN( x ->
        let r = CON(Succ x)
        in r)
    in foldNat a succ
    )
```

```
mul       = FUN( a ->
    let succ = THUNK (add a)
    in foldNat one succ
    )
```

```
exp       = THUNK(
    let f = FUN( x -> let axx = THUNK (add x x)
                      in add axx x)
        a = THUNK (add Zero two)
    in f a
    )
```

Программа состоит из связок вида имя = объектКучи. Эти связки называют глобальными, они становятся статическими объектами кучи, остальные объекты выделяются динамически в **let**-выражениях. Глобальный объект типа **THUNK** называют постоянной аппликативной формой (constant applicative form или сокращённо CAF).

Вычисление STG

Итак у нас есть упрощённый функциональный язык. Как мы будем вычислять выражения? Присутствие частичного применения усложняет этот процесс. Для многих функций мы не знаем заранее их аргументы. Так например в выражении

$f\ x\ y$

Функция f может иметь один аргумент в определении, но вернуть функцию. Есть два способа вычисления таких функций:

- *вставка-вход* (push-enter). Когда мы видим применение функции, мы сначала *вставляем* все аргументы в стек, затем совершаем *вход* в тело функции. В процессе входа мы вычисляем функцию f и узнаём число аргументов, которое ей нужно, после этого мы извлекаем из стека необходимое число аргументов, и применяем к ним функцию, если мы снова получаем функцию, тогда мы опять добираем необходимое число аргументов из стека. И так пока аргументы в стеке не кончатся.
- *вычисление-применение* (eval-apply). Вместе с функцией мы храним информацию о том, сколько аргументов ей нужно. Если это статически определённая функция (определение выписано пользователем), то число аргументов мы можем понять по левой части определения. В этой стратегии, если число аргументов известно, мы сразу *вычисляем* значение с нужным числом аргументов, сохранив оставшиеся в стеке, а затем извлекаем аргументы из стека и *применяем* к ним вычисленное значение.

Возвращаясь к исходному примеру, предположим, что арность функции f равна единице. Тогда стратегия вставка-вход сначала добавит на стек x и y , а затем будет добирать из стека необходимые аргументы. Стратегия вычисление-применение сначала вычислит $(f\ x)$, сохранив y на стеке, затем попытается применить результат к y . Почему мы говорим попытается? Может так случиться, что арность значения $f\ x$ окажется равным трём, но пока y у нас есть лишь один аргумент, тогда мы создадим объект **PAP**, который соответствует частичному применению.

Эти стратегии применимы как к ленивым, так и к энергичным языкам. Исторически сложилось, что ленивые языки тяготеют к первой стратегии, а энергичные ко второй. До недавнего времени и в GHC применялась первая стратегия. Пока однажды разработчики GHC всё же не решили сравнить две стратегии. Реализовав обе стратегии, и проверив их на большом количестве разных по сложности программ,

они пришли к выводу, что ни одна из стратегий не даёт существенного преимущества на этапе вычислений. Потребление ресурсов оказалось примерно равным. Но вторая стратегия заметно выигрывала в простоте реализации. Подробнее об этом можно почитать в статье Simon Marlow, Simon Peyton Jones: Making a Fast Curry: Push/Enter vs. Eval/Apply. Описание модели вычислений GHC, которое вы сейчас читаете копирует описание приведённое в этой статье.

Куча

Объекты кучи принято называть *замыканиями* (closure). Их называют так, потому что обычно для вычисления выражения нам не достаточно знать его текст, например посмотрим на функцию:

```
mul      = FUN( a ->
              let succ = THUNK (add a)
              in foldNat one succ
            )
```

Для того, чтобы вычислить `THUNK(add a)` нам необходимо знать значение `a`, это значение определено в теле функции. Оно определяется из контекста. По отношению к объекту такую переменную называют *свободной* (free). В куче мы будем хранить не только выражение `(add a)`, но и ссылки на все свободные переменные, которые участвуют в выражении объекта. Эти ссылки называют *довесок* (payload). Объект кучи содержит ссылку на специальную таблицу и довесок. В таблице находятся информация о типе объекта и код, который необходимо вычислить, а также другая служебная информация. При вычислении объекта мы заменяем ссылки настоящими значениями или ссылками на конструкторы.

Объект кучи может быть:

- `FUN` – определением функции;
- `PAP` – частичным применением;
- `CON` – полностью применённым конструктором;
- `THUNK` – отложенным вычислением;
- `BLACKHOLE` – это значение используется во время вычисления `THUNK`. Этот трюк предотвращает появление утечек памяти.

Мы будем считать, что куча – это таблица, которая ставит в соответствие адресам объекты или вычисленные значения.

Стек

Стек служит для быстрого переключения контекста. Мы будем пользоваться стеком при вычислении **case**-выражений и **THUNK**-объектов. При вычислении **case**-выражения мы сохраняем в стеке альтернативы и место возврата значения, а сами начинаем вычислять аргумент **case**-выражения. При вычислении **THUNK**-объекта мы запомним в стеке, адрес с которым необходимо связать полученное значение.

При вычислении в стратегии вставка-вход мы будем сохранять в стеке аргументы функции. А при вычислении в стратегии вычисление-применение мы также будем сохранять аргументы функции в стеке. Какая разница между этими вариантами? В первой стратегии мы можем доставать из стека произвольное число аргументов, после определения арности функции мы подбираем столько, сколько нам нужно, поэтому мы будем хранить аргументы по одному. Во второй же стратегии нам нужно просто сохранить все оставшиеся аргументы. Мы сохраняем и извлекаем их все сразу. Упрощая, объекты стека можно представить так:

$k ::=$	$\text{case } \bullet \text{ of } \{alt_1; \dots alt_n\}$	контекст case-выражения
	$Upd\ t\ \bullet$	Обновить отложенное вычисление
	$(\bullet a_1 \dots a_n)$	Применить функцию к аргументам, только для стратегии вычисление-применение
	$Arg\ a$	Аргумент на потом, только для стратегии вставка-вход

Синтаксис STG

Правила общие для обеих стратегий вычисления

Состояние вычислителя состоит из трёх частей. Это выражение для вычисления $\$e\$,$ стек $\$s\$$ и куча $\$H\$$. Мы рассмотрим правила по которым вычислитель переходит из одного состояния в другое. Все они имеют вид:

$$\$e_1; \quad \$s_1; \quad \$H_1 \quad \rightarrow \quad \$e_2; \quad \$s_2; \quad \$H_2$$

Левая часть переходит в правую, при условии, что левая часть имеет определённый вид. Начнём с правил, которые одинаковы и в той и в другой стратегии вычисления. Для простоты пока мы будем полагать, что объекты только добавляются в кучу и никогда не стираются. Мы будем обозначать добавление в стек как добавление элемента в обычный список: $\$elem\ : \ \$$.

Рассмотрим правило для **let**-выражений:

$$\text{let } x = \text{obj in } e; \ s; \ H \Rightarrow e[x'/x]; \ s; \ H[x' \rightarrow \text{obj}], \ x' - \text{новое имя}$$

Синтаксис STG

В этом правиле мы добавляем в кучу новый объект $\$obj\ \$$ под именем (или по адресу) $\$x'\ \$$. Запись $\$e[x'/x]\ \$$ означает замену $\$x\ \$$ на $\$x'\ \$$ в выражении $\$e\ \$$.

Теперь разберёмся с правилами для **case**-выражений.

$$\begin{aligned} \text{case } v \text{ of } \{ \dots; C \ x_1 \dots x_n \rightarrow e; \dots \}; \ s; \ H &\Rightarrow e[a_1/x_1 \dots a_n/x_n]; \ s; \ H \\ s; \ H[v \rightarrow \text{CON}(C \ a_1 \dots a_n)] & \\ \text{case } v \text{ of } \{ \dots; x \rightarrow e \}; \ s; \ H &\Rightarrow e[v/x]; \ s; \ H \\ \text{Если } v - \text{литерал или } H[v] - \text{значение,} & \\ \text{которое не подходит ни по одной из альтернатив} & \\ \text{case } e \text{ of } \{ \dots \}; \ s; \ H &\Rightarrow e; \ \text{case } \bullet \text{ of } \{ \dots \} : s; \ H \\ v; \ \text{case } \bullet \text{ of } \{ \dots \} : s; \ H &\Rightarrow \text{case } v \text{ of } \{ \dots \}; \ s; \ H \end{aligned}$$

Синтаксис STG

Вычисления начинаются с третьего правила, в котором нам встречается **case**-выражения с произвольным $\$e\ \$$. В этом правиле мы сохраняем в стеке альтернативы и адрес возвращаемого значения и продолжаем вычисление выражения $\$e\ \$$. После вычисления мы перейдём к четвёртому правилу, тогда мы снимем со стека информацию необходимую для продолжения вычисления **case**-выражения. Это приведёт нас к одному из первых двух правил. В первом правиле значение аргумента содержит конструктор, подходящий по одной из альтернатив, а во втором мы выбираем альтернативу по умолчанию.

Теперь посмотрим как вычисляются **THUNK**-объекты.

$$\begin{aligned} x; \ s; \ H[x \rightarrow \text{THUNK } e] &\Rightarrow e; \ \text{Upd } x \bullet : s; \ H[x \rightarrow \text{BLACKHOLE}] \\ y; \ \text{Upd } x \bullet : s; \ H &\Rightarrow y; \ s; \ H[x \rightarrow H[y]] \\ \text{если } H[y] \text{ является значением} & \end{aligned}$$

Синтаксис STG

Если переменная указывает на отложенное вычисление $\$e\$,$ мы сохраняем в стеке адрес по которому необходимо обновить значение и вычисляем значение $\$e\$$. В это время мы записываем в по адресу $\$x\$$ объект $\$BLACKHOLE\$$. У нас нет такого правила, которое реагирует на левую часть, если в ней содержится объект $\$BLACKHOLE\$$. Поэтому во время вычисления $\$THUNK\$$ ни одно из правил сработать не может. Этот трюк необходим для избежания утечек памяти. Как только выражение будет вычислено, мы извлечём из стека адрес $\$x\$$ и обновим значение.

Правила применения функций, если арность совпадает с числом аргументов в тексте выражения:

$$\begin{array}{lcl}
 f^n a_1 \dots a_n; \quad s; \quad H[y \rightarrow FUN(x_1 \dots x_n \rightarrow e)] & \Rightarrow & e[a_1/x_1 \dots a_n/x_n]; s; H \\
 \oplus a_1 \dots a_n; \quad s; \quad H & \Rightarrow & a; \quad s; \quad H \\
 & & a - \text{результат вычисления } (\oplus a_1 \dots a_n)
 \end{array}$$

Синтаксис STG

Мы просто заменяем все вхождения аргументов на значения. Второе правило выполняет применение примитивной функции к значениям.

Правила для стратегии вставка-вход

$$\begin{array}{lcl}
 f^k a_1 \dots a_m; \quad s; \quad H & \Rightarrow & f; \quad Arg a_1 : \dots : Arg a_m : s; \quad H \\
 f; \quad Arg a_1 : \dots : Arg a_n : s; \quad H[f \rightarrow FUN(x_1 \dots x_n \rightarrow e)] & \Rightarrow & e[a_1/x_1 \dots a_n/x_n]; \quad s; \quad H \\
 f; \quad Arg a_1 : \dots : Arg a_m : s; \quad H[f \rightarrow FUN(x_1 \dots x_n \rightarrow e)] & \Rightarrow & p; \quad s; \quad H[p \rightarrow PAP(f a_1 \dots a_m)] \\
 & & \text{при } m \geq 1; \quad m < n; \text{ верхний элемент } s \\
 & & \text{не является } Arg; \quad p - \text{новый адрес} \\
 f; \quad Arg a_{n+1} : s; \quad H[f \rightarrow PAP(g a_1 \dots a_n)] & \Rightarrow & g; \quad Arg a_1 : \dots : Arg a_n : Arg a_{n+1} : s; \quad H
 \end{array}$$

Синтаксис STG

Первое правило выполняет этап “вставка”. Если мы видим применение функции, мы первым делом сохраняем все аргументы в стеке. Во втором правиле мы вычислили значение f , оно оказалось функцией с арностью $\$n\$$. Тогда мы добираем из стека $\$n\$$ аргументов и подставляем их в правую часть функции $\$e\$$. Если в стеке оказалось слишком мало аргументов, то мы переходим к третьему правилу и составляем частичное применение. Последнее правило говорит о том

как расшифровывается частичное применение. Мы вставляем в стек все аргументы и начинаем вычисление функции $\$g\$$ из тела $\$PAP\$$.

Правила для стратегии вычисление-применение

$$\begin{aligned}
 f^\bullet a_1 \dots a_n; \quad s; \quad H[f \rightarrow FUN(x_1 \dots x_n \rightarrow e)] &\Rightarrow e[a_1/x_1 \dots a_n/x_n]; \quad s; \quad H \\
 f^k a_1 \dots a_m; \quad s; \quad H[f \rightarrow FUN(x_1 \dots x_n \rightarrow e)] &\Rightarrow e[a_1/x_1 \dots a_n/x_n]; \quad (\bullet a_{n+1} \dots a_m) : s; \quad H \\
 &\quad \text{при } m \geq n \\
 &\Rightarrow p; \quad s; \quad H[p \rightarrow PAP(f a_1 \dots a_m)] \\
 &\quad \text{при } m < n, \quad p - \text{новый адрес} \\
 f^\bullet a_1 \dots a_m; \quad s; \quad H[f \rightarrow THINK e] &\Rightarrow f; \quad (\bullet a_1 \dots a_m) : s; \quad H \\
 f^k a_{n+1} \dots a_m; \quad s; \quad H[f \rightarrow PAP(g a_1 \dots a_n)] &\Rightarrow g^\bullet a_1 \dots a_n a_{n+1} \dots a_m; \quad s; \quad H \\
 f; \quad (\bullet a_1 \dots a_n) : s; \quad H &\Rightarrow f^\bullet a_1 \dots a_n; \quad s; \quad H \\
 &\quad H[f] \text{ является } FUN \text{ или } PAP
 \end{aligned}$$

Синтаксис STG

Разберёмся с первыми двумя правилами. В первом правиле статическая арность $\$f\$$ неизвестна, но значение $\$f\$$ уже вычислено, и мы можем узнать арность по объекту $\$FUN\$$, далее возможны три случая. Число аргументов переданных в функцию совпадает с арностью $\$FUN\$$, тогда мы применяем аргументы к правой части $\$FUN\$$. Если в функцию передано больше аргументов чем нужно, мы сохраняем лишние на стеке. Если же аргументов меньше, то мы создаём объект $\$PAP\$$. Третье правило говорит о том, что нам делать, если значение $\$f\$$ ещё не вычислено. Оно является $\$THUNK\$$. Тогда мы сохраним аргументы на стеке и вычислим его. В следующем правиле мы раскрываем частичное применение. Мы просто организуем вызов функции со всеми аргументами (и со стека и из частичного применения). Последнее правило срабатывает после третьего. Когда мы вычислим $\$THUNK\$$ и увидим там $\$FUN\$$ или $\$PAP\$$. Тогда мы составляем применение функции.

Сложность применения стратегии вставка-вход связана с плохо предсказуемым изменением стека. Если в стратегии вычисление-выполнение мы добавляем и снимаем все аргументы, то в стратегии вставка-вход мы добавляем их по одному и неизвестно сколько снимем в следующий раз. Кроме того стратегия вычисление-

применение позволяет проводить оптимизацию перемещения аргументов. Вместо стека мы можем хранить аргументы в регистрах. Тогда скорость обращения к аргументам резко возрастёт.

Представление значений в памяти. Оценка занимаемой памяти

Ранее мы говорили, что полностью вычисленное значение – это дерево, в узлах которого находятся одни лишь конструкторы. Процесс вычисления похож на очистку дерева выражения от синонимов. Мы начинаем с самого верха и идём к листьям. Потом мы выяснили, что для предотвращения дублирования вычислений мы подставляем в функции не сами значения, а ссылки на значения. Теперь нам понятно, что ссылки указывают на объекты в куче. Ссылки – это атомарные переменные. Полностью вычисленное значение является сетью (или графом) объектов кучи типа **CON**.

Поговорим о том сколько места в памяти занимает то или иное значение. Как мы говорили память компьютера состоит из ячеек, в которых хранятся значения. У каждой ячейки есть адрес. Ячейки памяти неделимы, их также принято называть словами. Мы будем оценивать размер значения в словах.

Каждый конструктор требует столько слов сколько у него полей плюс ещё одно слово для ссылки на служебную информацию (она нужна вычислителю). Посмотрим на примеры:

```
data Int = I# Int#           -- 2 слова
data Pair a b = Pair a b    -- 3 слова
```

У этого правила есть исключение. Если у конструктора нет полей, то есть он является константой или примитивным конструктором, то в процессе вычисления значение этого конструктора представлено ссылкой. Это означает, что внутри программы все значения ссылаются на одну область памяти. У нас действительно есть лишь один пустой список или одно значение **True** или **False**.

Посчитаем число слов в значении **[Pair 1 2]**. Для этого для начала перепишем его в STG

```

nil = []                                -- глобальный объект (не в счёт)

let x1 = I# 1                          -- 2 слова
    x2 = I# 2                          -- 2 слова
    p  = Pair x1 x2                    -- 3 слова
    val = Cons p nil                  -- 3 слова
in val                                -----
                                      -- 10 слов

```

Поскольку объект кучи **CON** может хранить только ссылки, нам пришлось введением дополнительных переменных “развернуть” значение. Примитивный конструктор не считается, поскольку он сохранён глобально, в итоге получилось 10 слов. Посмотрим на ещё один пример, распишем значение **[Just True, Just True, Nothing]**:

```

nil      = []
true     = True
nothing  = Nothing

let x1 = Just true                    -- 2 слова
    x2 = Just true                    -- 2 слова
    p1 = Cons nothing nil             -- 3 слова
    p2 = Cons x2 p1                   -- 3 слова
    p3 = Cons x1 p2                   -- 3 слова
in p3                                -----
                                      -- 13 слов

```

Обычно одно слово соответствует 16, 32 или 64 битам. Эта цифра зависит от процессора. Мы считали, что любое значение можно поместить в одно слово, но это не так. Возьмём к примеру действительные числа с двойной точностью, они не поместятся в одно слово. Это необходимо учитывать при оценке объёма занимаемой памяти.

Управление памятью. Сборщик мусора

В прошлом разделе для простоты мы считали, что объекты только добавляются в кучу. На самом деле это не так. Допустим во время вычисления функции нам нужно было вычислить какие-то промежуточные данные, например объявленные в локальных переменных, тогда после вычисления результата все эти значения больше не нужны. При этом в куче висит много-много объектов, которые уже не нужны. Нам нужно как-то от них избавиться. Этой задачей занимается отдельный блок вычислителя, который называется

сборщиком мусора (garbage collector), соответственно процесс автоматического освобождения памяти называется сборкой мусора (garbage collection или GC).

На данный момент в GHC используется копирующий последовательный сборщик мусора, который работает по алгоритму Чейни (Cheney). Для начала рассмотрим простой алгоритм сборки мусора. Мы выделяем небольшой объём памяти и начинаем наполнять его объектами. Как только место кончится мы найдём все “живые” объекты, а остальное пространство памяти будем считать свободным. Как только после очередной очистки оказалось, что нам всё же не хватает места. Мы найдём все живые объекты, подсчитаем сколько места они занимают и запросим у системы этот объём памяти. Скопируем все живые объекты на новое место, а старую память будем считать свободной. Так например, если у нас было выделено 30 Мб памяти и оказалось, что живые объекты занимают 10 Мб, мы выделим ещё 10 Мб, скопируем туда все живые объекты и общий объём памяти станет равным 40 Мб.

Мы можем оптимизировать сборку мусора. Есть такая гипотеза, что большинство объектов имеют очень короткую жизнь. Это промежуточные данные, локальные переменные. Нам нужен лишь результат функции, но на подходе к результату мы сгенерируем много разовой информации. Ускорить очистку можно так. Мы выделим совсем небольшой участок памяти внутри нашей кучи, его принято называть *яслями* (nursery area), и будем выделять и собирать новые объекты только в нём, как только этот участок заполнится мы скопируем все живые объекты из яслей в остальную память и снова будем наполнять ясли. Как только вся память закончится мы поступим так же как и в предыдущем сценарии. Когда заканчивается место в яслях, мы проводим поверхностную очистку (minor GC), а когда заканчивается вся память в текущей куче, мы проводим глубокую очистку (major GC). Эта схема соответствует сборке с двумя поколениями.

Статистика выполнения программы

Процесс управления памятью скрыт от программиста. Но при этом в GHC есть развитые средства косвенной диагностики работы

программы. Пока мы пользовались самым простым способом проверки. Мы включали флаг `s` в интерпретаторе. Пришло время познакомиться и с другими.

Статистика вычислителя

Для начала научимся смотреть статистику работы вычислителя. Посмотреть статистику можно с помощью флагов `s[file]` и `S[file]`. Эти флаги предназначены для вычислителя низкого уровня (realtime system или RTS, далее просто вычислитель), они заключаются в окружение `+RTS ... -RTS`, если флаги идут в конце строки и считается, что все последующие флаги предназначены для RTS мы можем просто написать `ghc --make file.hs +RTS ...`. Например скомпилируем такую программу:

```
module Main where
```

```
main = print $ sum [1 .. 1e5]
```

Теперь скомпилируем:

```
$ ghc --make sum.hs -rtsopts -fforce-recomp
```

Флаг `rtsopts` позволяет передавать скомпилированной программе флаги для вычислителя низкого уровня, далее для краткости мы будем называть его просто вычислителем. С флагом `fforce-recomp` программа будет каждый раз заново пересобирааться. Теперь посмотрим на статистику выполнения программы (флаг `s[file]`, в этом примере мы перенаправляем выход в поток `stderr`):

```
$ ./sum +RTS -sstderr
```

```
5.00005e9
```

```
14,145,284 bytes allocated in the heap
```

```
11,110,432 bytes copied during GC
```

```
2,865,704 bytes maximum residency (3 sample(s))
```

```
460,248 bytes maximum slop
```

```
7 MB total memory in use (0 MB lost due to fragmentation)
```

```
Tot time (elapsed)  Avg pause  Max pause
```

```
Gen 0      21 colls,    0 par    0.00s   0.01s   0.0006s   0.0036s
```

```
Gen 1       3 colls,    0 par    0.01s   0.01s   0.0026s   0.0051s
```

```
INIT    time    0.00s  ( 0.00s elapsed)
```

```
MUT     time    0.01s  ( 0.01s elapsed)
```

```
GC      time    0.01s  ( 0.02s elapsed)
```

```
EXIT    time    0.00s ( 0.00s elapsed)
Total   time    0.02s ( 0.03s elapsed)

%GC      time      60.0% (69.5% elapsed)

Alloc rate    1,767,939,507 bytes per MUT second

Productivity  40.0% of total user, 26.0% of total elapsed
```

Был распечатан результат и отчёт о работе программы. Разберёмся с показателями:

```
bytes allocated in the heap  -- число байтов выделенных в куче
                             -- за всё время работы программы
bytes copied during GC       -- число скопированных байтов
                             -- за всё время работы программы
bytes maximum residency      -- в каком объёме памяти работала программа
                             -- в скобках указано число глубоких очисток
bytes maximum slop           -- максимум потерь памяти из-за фрагментации

total memory in use          -- сколько всего памяти было запрошено у ОС
```

Показатель `bytes maximum residency` измеряется только при глубокой очистке, поскольку новая память выделяется именно в этот момент. Размер памяти выделенной в куче гораздо больше общего объёма памяти. Так происходит потому, что этот показатель указывает на общее число памяти в куче за всё время работы программы. Ведь мы переиспользуем не нужную нам память. По этому показателю можно судить о том, сколько замыканий (объектов) было выделено в куче.

Следующие две строчки говорят о числе сборок мусора. Мы видим, что GC выполнил 21 поверхностную очистку (поколение 0) и 3 глубоких (поколение 1). Далее идут показатели скорости. `INIT` и `EXIT` – это инициализация и завершение программы. `MUT` – это полезная нагрузка, время, которая наша программа тратила на изменение (MUTation) значений. `GC` – время сборки мусора. Далее GHC сообщил нам о том, что мы провели 60% времени в сборке мусора. Это очень плохо. Продуктивность программы очень низкая. Затратна глубокая сборка мусора, поверхностная – это дело обычное. Теперь посмотрим на показатели строгой версии этой программы:

```
module Main where

import Data.List(foldl')
```

```
sum' = foldl' (+) 0
main = print $ sum' [1 .. 1e5]
```

Скомпилируем:

```
$ ghc --make sumStrict.hs -rtsopts -fforce-recomp
```

Посмотрим на статистику:

```
$ ./sumStrict +RTS -sstderr
5.00005e9
  10,474,128 bytes allocated in the heap
   24,324 bytes copied during GC
  27,072 bytes maximum residency (1 sample(s))
  27,388 bytes maximum slop
    1 MB total memory in use (0 MB lost due to fragmentation)

                               Tot time (elapsed)  Avg pause  Max pause
Gen  0          19 colls,    0 par    0.00s   0.00s    0.0000s   0.0000s
Gen  1           1 colls,    0 par    0.00s   0.00s    0.0001s   0.0001s

INIT   time    0.00s ( 0.00s elapsed)
MUT    time    0.01s ( 0.01s elapsed)
GC     time    0.00s ( 0.00s elapsed)
EXIT   time    0.00s ( 0.00s elapsed)
Total  time    0.01s ( 0.01s elapsed)

%GC     time    0.0% (3.0% elapsed)

Alloc rate   1,309,266,000 bytes per MUT second

Productivity 100.0% of total user, 116.0% of total elapsed
```

Мы видим, что произошла лишь одна глубокая сборка. И это существенно сказалось на продуктивности. Кроме того мы видим, что программа заняла лишь 27 Кб памяти, вместо 2 Мб как в прошлом случае. Теперь давайте покрутим ручки у GC. В GHC можно устанавливать разные параметры сборки мусора с помощью флагов. Все флаги можно посмотреть в документации GHC. Мы обратим внимание на несколько флагов. Флаг **H** назначает минимальное значение для стартового объёма кучи. Флаг **A** назначает объём памяти для яслей. По умолчанию размер яслей равен 512 Кб (эта цифра может измениться). Изменением этих параметров мы можем отдалить сборку мусора. Чем дольше работает программа между

сборками, тем выше вероятность того, что многие объекты уже не нужны.

Давайте убедимся в том, что поверхностные очистки происходят очень быстро и совсем не тормозят программу. Установим размер яслей на 32 Кб вместо 512 Кб как по умолчанию (размер пишется сразу за флагом, за цифрой идёт k или m):

```
$ ./sumStrict +RTS -A32k -sstderr
```

```
...
          Tot time (elapsed)  Avg pause  Max pause
Gen  0      318 colls,      0 par    0.00s    0.00s    0.0000s    0.0000s
Gen  1       1 colls,      0 par    0.00s    0.00s    0.0001s    0.0001s
...
MUT    time    0.01s ( 0.01s elapsed)
GC     time    0.00s ( 0.00s elapsed)
...
%GC     time      0.0% (11.8% elapsed)
```

Мы видим, что за счёт уменьшения памяти очистки существенно участились, но это не сказалось на общем результате. С помощью флага `H[size]` мы можем устанавливать рекомендуемое минимальное значение для размера кучи. Оно точно не будет меньше. Вернёмся к первому варианту и выделим алгоритму побольше памяти, например 20 Мб:

```
./sum +RTS -A1m -H20m -sstderr
```

```
5.00005e9
```

```
14,145,284 bytes allocated in the heap
319,716 bytes copied during GC
324,136 bytes maximum residency (1 sample(s))
60,888 bytes maximum slop
22 MB total memory in use (1 MB lost due to fragmentation)
```

```
          Tot time (elapsed)  Avg pause  Max pause
Gen  0      2 colls,      0 par    0.00s    0.00s    0.0001s    0.0001s
Gen  1      1 colls,      0 par    0.00s    0.00s    0.0007s    0.0007s

INIT    time    0.00s ( 0.00s elapsed)
MUT     time    0.02s ( 0.02s elapsed)
GC      time    0.00s ( 0.00s elapsed)
EXIT    time    0.00s ( 0.00s elapsed)
Total   time    0.02s ( 0.02s elapsed)

%GC     time      0.0% (4.4% elapsed)
```

```
Alloc rate    884,024,998 bytes per MUT second
```

```
Productivity 100.0% of total user, 78.6% of total elapsed
```

Произошла лишь одна глубокая очистка (похоже, что эта очистка соответствует начальному выделению памяти) и продуктивность программы стала стопроцентной. С помощью флага **S** вместо **s** мы можем посмотреть более детальную картину управления памяти. Будут распечатаны показатели памяти для каждой очистки.

```
./sum +RTS -Sfile
```

В файле `file` мы найдём такую таблицу:

память			GC		время		Total		Тип очистки	
выделено	скопировано	в живых								
Alloc	Copied	Live	GC	GC	TOT	TOT	Page	Flts		
bytes	bytes	bytes	user	elap	user	elap				
545028	150088	174632	0.00	0.00	0.00	0.00	0	0	(Gen: 1)	
523264	298956	324136	0.00	0.00	0.00	0.00	0	0	(Gen: 0)	
...										

Итак у нас появился один существенный показатель качества программ. Это количество глубоких очисток. Во время глубокой очистки вычислитель производит две затратные операции: сканирование всей кучи и запрос у системы возможно большого блока памяти. Чем меньше таких очисток, тем лучше. Сократить их число можно удачной комбинацией показателей **A** и **H**. Но не стоит сразу начинать обновлять параметры по умолчанию, если ваша программа работает слишком медленно. Лучше сначала попробовать изменить алгоритм. Найти функцию, которая слишком много ленится и ограничить её с помощью `seq` или энергичных образцов. В этом примере у нас была всего одна функция, поэтому поиск не составил труда. Но что если их уже очень много? Скорее всего так и будет. Не стоит оптимизировать не рабочую программу. А в рабочей программе обычно много функций. Но это не так страшно, помимо суммарных показателей GHC позволяет собирать более конкретную статистику.

Стоит отметить функцию `performGC` из модуля `System.Mem`, она форсирует поверхностную сборку мусора. Допустим вы читаете какие-то данные из файла и тут же преобразуете их в структуру данных. После того как чтение данных закончится, вы знаете, что

промежуточные данные, связанные с чтением, вам уже не нужны. Выполнив `performGC` вы можете подсказать об этом вычислителю.

Профилирование функций

Время и общий объём памяти

Процесс отслеживания показателей память/скорость называется профилированием программы. Всё вроде бы работает, но работает слишком медленно, необходимо установить причину. Рассмотрим такую программу:

```
module Main where

concatR = foldr (++) []
concatL = foldl (++) []

fun :: Double
fun = test concatL - test concatR
    where test f = last $ f $ map return [1 .. 1e6]

main = print fun
```

У нас есть подозрение, что какая-то из двух функций `concatX` работает слишком медленно. Мы можем посмотреть какая, если добавим к ним специальную прагму `SCC`:

```
concatR = {-# SCC "right" #-} foldr (++) []
concatL = {-# SCC "left"  #-} foldl (++) []
```

Напомню, что прагмой называется специальный блочный комментарий с решёткой. Это специальное сообщение компилятору. Прагмой `SCC` мы устанавливаем так называемый центр затрат (cost center). Она пишется сразу за знаком равно. В кавычках пишется имя, под которым статистика войдёт в итоговый отчёт. После этого вычислитель будет следить за нагрузкой, которая приходится на эту функцию. Теперь нам нужно скомпилировать модуль с флагом `prof`, который активирует подсчёт статистики в центрах затрат:

```
$ ghc --make concat.hs -rtsopts -prof -fforce-recomp
$ ./concat +RTS -p
```

Второй командой мы запускаем программу и передаём вычислителю флаг `p`. После этого будет создан файл `concat.prof`. Откроем этот файл:

```
concat +RTS -p -RTS
```

```
total time   =          1.45 secs  (1454 ticks @ 1000 us, 1 processor)
total alloc  = 1,403,506,324 bytes (excludes profiling overheads)
```

```
COST CENTRE MODULE    %time %alloc
```

```
left           Main      99.8  99.8
```

				individual	
inherited					
COST	CENTRE	MODULE	%time	%alloc	
			no.	entries	%time %alloc
MAIN		MAIN	46	0	0.0 0.0
100.0	100.0				
CAF		GHC.Integer.Logarithms.Internals	91	0	0.0 0.0
0.0	0.0				
CAF		GHC.IO.Encoding.Iconv	71	0	0.0 0.0
0.0	0.0				
CAF		GHC.IO.Encoding	70	0	0.0 0.0
0.0	0.0				
CAF		GHC.IO.Handle.FD	57	0	0.0 0.0
0.0	0.0				
CAF		GHC.Conc.Signal	56	0	0.0 0.0
0.0	0.0				
CAF		Main	53	0	0.2 0.2
100.0	100.0				
right		Main	93	1	0.0 0.0
0.0	0.0				
left		Main	92	1	99.8 99.8
99.8	99.8				

Мы видим, что почти всё время работы программа провела в функции `concatL`. Функция `concatR` была вычислена мгновенно (`time`) и почти не потребовала ресурсов памяти (`alloc`). У нас есть две пары колонок результатов. `individual` указывает на время вычисления функции, а `inherited` – на время вычисления функции и всех дочерних функций. Колонка `entries` указывает число вызовов функции. Если мы хотим проверить все функции мы можем не указывать функции прагмами. Для этого при компиляции указывается флаг `auto-all`. Отметим также, что все константы определённы на

самом верхнем уровне модуля, сливаются в один центр. Они называются в отчёте как CAF. Для того чтобы вычислитель следил за каждой константой по отдельности необходимо указать флаг `caf-all`. Попробуем на таком модуле:

```
module Main where

fun1 = test concatL - test concatR
fun2 = test concatL + test concatR

test f = last $ f $ map return [1 .. 1e4]

concatR = foldr (++) []
concatL = foldl (++) []

main = print fun1 >> print fun2
```

Скомпилируем:

```
$ ghc --make concat2.hs -rtsopts -prof -auto-all -caf-all -fforce-recomp
$ ./concat2 +RTS -p
0.0
20000.0
```

После этого можно открыть файл `concat2.prof` и посмотреть итоговую статистику по всем значениям. Программа с включённым профилированием будет работать гораздо медленней, не исключено, что ей не хватит памяти на стеке, в этом случае вы можете добавить памяти с помощью флага вычислителя `K`, впрочем если это произойдёт GHC подскажет вам что делать.

Динамика изменения объёма кучи

В предыдущем разделе мы смотрели общее время и память затраченные на вычисление функции. В этом мы научимся измерять динамику изменения расхода памяти на куче. По этому показателю можно понять в какой момент в программе возникают утечки памяти. Мы увидим характерные горбы на картинках, когда GC будет активно запрашивать новую память. Для этого сначала нужно скомпилировать программу с флагом `prof` как и в предыдущем разделе, а при выполнении программы добавить один из флагов `hc`, `hm`, `hd`, `hy` или `hr`. Все они начинаются с буквы `h`, от слова `heap` (куча). Вторая буква указывает тип графика, какими показателями мы интересуемся. Все они создают специальный файл `имяПриложения.hr`, который мы

можем преобразовать в график в формате **PostScript** с помощью программы **hp2ps**, она устанавливается автоматически вместе с **GHC**.

Рассмотрим типичную утечку памяти (из упражнения к предыдущей главе):

```
module Main where

import System.Environment(getArgs)

main = print . sum2 . xs . read =<< fmap head getArgs
      where xs n = [1 .. 10 ^ n]

sum2 :: [Int] -> (Int, Int)
sum2 = iter (0, 0)
      where iter c [] = c
            iter c (x:xs) = iter (tick x c) xs

tick :: Int -> (Int, Int) -> (Int, Int)
tick x (c0, c1) | even x = (c0, c1 + 1)
                | otherwise = (c0 + 1, c1)
```

Скомпилируем с флагом профилирования:

```
$ ghc --make leak.hs -rtsospts -prof -auto-all
```

Статистика вычислителя показывает, что эта программа вызывала глубокую очистку 8 раз и выполняла полезную работу лишь 40% времени.

```
$ ./leak 6 +RTS -K30m -sstderr
```

```
...
```

			Tot time (elapsed)		Avg pause	Max pause
Gen 0	493 colls,	0 par	0.26s	0.26s	0.0005s	0.0389s
Gen 1	8 colls,	0 par	0.14s	0.20s	0.0248s	0.0836s

```
...
```

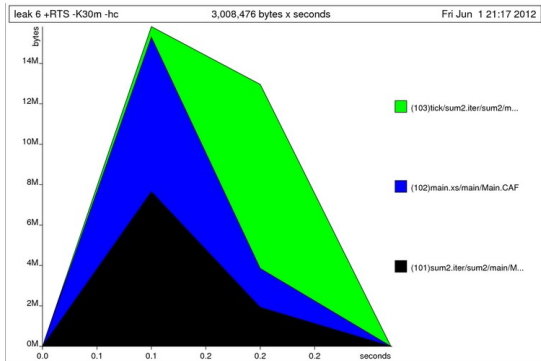
```
Productivity 40.5% of total user, 35.6% of total elapsed
```

Теперь посмотрим на профиль кучи.

```
$ ./leak 6 +RTS -K30m -hc
(500000,500000)
$ hp2ps -e80mm -c leak.hp
```

В первой команде мы добавили флаг **hc** для того, чтобы создать файл с расширением **.hp**. Он содержит таблицу с показателями размера кучи, которые вычислитель замеряет через равные промежутки времени. Мы можем изменять интервал с помощью флага **iN**, где **N** –

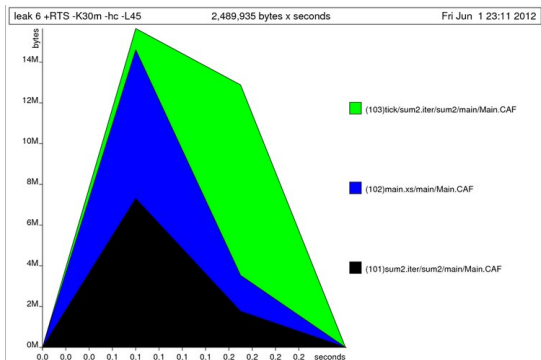
время в секундах. Второй командой мы преобразуем профиль в картинку. Флаг `s`, говорит о том, что мы хотим получить цветную картинку, а флаг `e80mm`, говорит о том, что мы собираемся вставить картинку в текст LaTeX. После `e` указан размер в миллиметрах. Мы видим характерный горб.



Профиль кучи для утечки памяти

В картинку не поместились имена функций мы можем увеличить строку флагом `L`. Теперь все имена поместились.

```
$ ./leak 6 +RTS -K30m -hc -L45  
(500000,500000)  
$ hp2ps -e80mm -c leak.hp
```

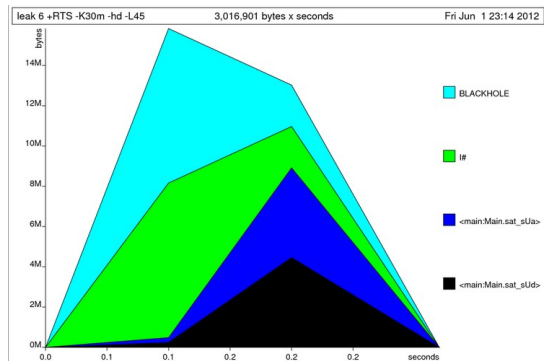


Профиль кучи для утечки памяти

С помощью флага `hd` посмотрим на объекты, которые застряли в куче:

```
$ ./leak 6 +RTS -K30m -hd -L45  
(500000,500000)  
$ hp2ps -e80mm -c leak.hp
```

Теперь куча разбита по типу объектов (замыканий). **BLACKHOLE** это специальный объект, который заменяет **THUNK** во время его вычисления. **I#** – это скрытый конструктор **Int**. **sat_sUa** и **sat_sUd** – это имена застрявших отложенных вычислений. Если бы наша программа была очень большой на этом месте мы бы запустили профилирование по функциям с флагом **p** и из файла **leak.prof** узнали бы в каких функциях программа тратит больше всего ресурсов. После этого мы бы пошли смотреть исходный код подозрительных функций и после внесённых изменений снова посмотрели бы на графики кучи.



Профиль кучи для утечки памяти

Если подумать, что мы делаем? Мы создаём отложенное вычисление, которое обещает построить большой список, вытягиваем из списка по одному элементу и, если элемент оказывается чётным, прибавляем к одному элементу пары, а если не чётным, то к другому. Проблема в том, что внутри пары происходит накопление отложенных вычислений, необходимо сразу вычислять значения перед запаковыванием их в пару. Изменим код:

```
{-# Language BangPatterns #-}
module Main where

import System.Environment(getArgs)

main = print . sum2 . xs . read =<< fmap head getArgs
  where xs n = [1 .. 10 ^ n]

sum2 :: [Int] -> (Int, Int)
sum2 = iter (0, 0)
  where iter c [] = c
        iter c (x:xs) = iter (tick x c) xs

tick :: Int -> (Int, Int) -> (Int, Int)
```



```

tick x (!c0, !c1) | even x    = (c0, c1 + 1)
                  | otherwise = (c0 + 1, c1)

```

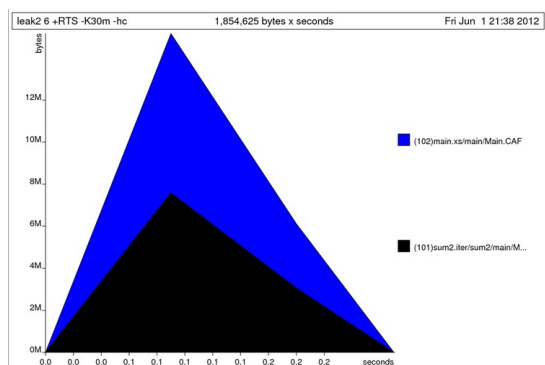
Мы сделали функцию `tick` строгой. Теперь посмотрим на профиль:

```

$ ghc --make leak2.hs -rtsopts -prof -auto-all
$ ./leak2 6 +RTS -K30m -hc
(500000,500000)
$ hp2ps -e80mm -c leak2.hp

```

Не получилось. Как же так. Посмотрим на расход памяти отдельных функций. `tick` стала строгой, но этого не достаточно, потому что в первом аргументе `iter` накапливаются вызовы `tick`. Сделаем `iter` строгой по первому аргументу:



Опять двойка

```

sum2 :: [Int] -> (Int, Int)
sum2 = iter (0, 0)
      where iter !c [] = c
              iter !c (x:xs) = iter (tick x c) xs

```

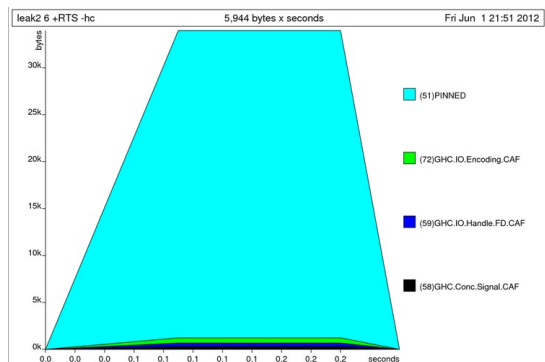
Теперь снова посмотрим на профиль:

```

$ ghc --make leak2.hs -rtsopts -prof -auto-all
$ ./leak2 6 +RTS -K30m -hc
(500000,500000)
$ hp2ps -e80mm -c leak2.hp

```

Мы видим, что память резко подскакивает и остаётся постоянной. Но теперь показатели измеряются не в мегабайтах, а в килобайтах. Мы справились. Остальные флаги `hX` позволяют наблюдать за разными специфическими объектами в куче. Мы можем узнать сколько памяти приходится на разные модули (`hm`), сколько памяти приходится на разные конструкторы (`hd`), на разные типы замыканий (`hy`).



Профиль кучи без утечки памяти

Поиск источников внезапной остановки

case-выражения и декомпозиция в аргументах функции могут стать источником очень неприятных ошибок. Программа прошла проверку типов, завелась и вот уже работает-работает как вдруг мы видим на экране:

```
*** Exception: Prelude.head: empty list
```

или

```
*** Exception: Maybe.fromJust: Nothing
```

И совсем не понятно откуда эта ошибка. В каком модуле сидит эта функция. Может мы её импортировали из чужой библиотеки или написали сами. Как раз для таких случаев в GHC предусмотрен специальный флаг `xs`.

Посмотрим на выполнение такой программы:

```
module Main where
```

```
addEvens :: Int -> Int -> Int
```

```
addEvens a b
```

```
    | even a && even b = a + b
```

```
q = zipWith addEvens [0, 2, 4, 6, 7, 8, 10] (repeat 0)
```

```
main = print q
```

Для того, чтобы воспользоваться флагом `xs` необходимо скомпилировать программу с возможностью профилирования:

```
$ ghc --make break.hs -rtsospts -prof
```

```
$ ./break +RTS -xc
*** Exception (reporting due to +RTS -xc): (THUNK_2_0), stack trace:
  Main.CAF
break: break.hs:(4,1)-(5,30): Non-exhaustive patterns in function addEvens
```

Так мы узнали в каком месте кода проявился злосчастный вызов, это строки (4,1)-(5,30). Что соответствует определению функции `addEvens`. Не очень полезная информация. Мы и так бы это узнали. Нам бы хотелось узнать тот путь, по которому шла программа к этому вызову. Проблема в том, что все вызовы слились в один CAF для модуля. Так разделим их:

```
$ ghc --make break.hs -rtsopts -prof -caf-all -auto-all
$ ./break +RTS -xc
*** Exception (reporting due to +RTS -xc): (THUNK_2_0), stack trace:
  Main.addEvens,
  called from Main.q,
  called from Main.CAF:q
  --> evaluated by: Main.main,
  called from :Main.CAF:main
break: break.hs:(4,1)-(5,30): Non-exhaustive patterns in function addEvens
```

Теперь мы видим путь к этому вызову, мы пришли в него из значения `q`, которое было вызвано из `main`.

Оптимизация программ

В этом разделе мы поговорим о том этапе компиляции, на котором происходят преобразования `Core -> Core`. Мы называли этот этап упрощением программы.

Флаги оптимизации

Мы можем задавать степень оптимизации программы специальными флагами. Самые простые флаги начинаются с большой буквы `O`. Естественно, чем больше мы оптимизируем, тем дольше компилируется код. Поэтому не стоит увлекаться оптимизацией на начальном этапе проектирования. Посмотрим какие возможности у нас есть:

- без `-O` – минимум оптимизаций, код компилируется как можно быстрее.
- `-O0` – выключить оптимизацию полностью
- `-O` – умеренная оптимизация.

- **02** – активная оптимизация, код компилируется дольше, но пока **02** не сильно выигрывает у **0** по продуктивности.

Для оптимизации мы компилируем программу с заданным флагом, например попробуйте скомпилировать самый первый пример с флагом **0**:

```
ghc --make sum.hs -O
```

и утечка памяти исчезнет.

Посмотреть описание конкретных шагов оптимизации можно в документации к GHC. Например при включённой оптимизации GHC применяет анализ строгости. В ходе него GHC может исправить простые утечки памяти за нас. Стоит отметить оптимизацию `-fexcess-precision`, он может существенно ускорить программы, в которых много вычислений с **Double**. Но при этом вычисления могут потерять в точности, округление становится непредсказуемым.

Прагма **INLINE**

Если мы посмотрим в исходный файл для модуля **Prelude**, то мы найдём такое определение для композиции функций:

```
-- | Function composition.
{-# INLINE (.) #-}
-- Make sure it has TWO args only on the left, so that it inlines
-- when applied to two functions, even if there is no final argument
(.) :: (b -> c) -> (a -> b) -> a -> c
(.) f g = \x -> f (g x)
```

Помимо знакомого нам определения и комментариев мы видим новую прагму **INLINE**. Она указывает компилятору на то, что на этапе упрощения программы необходимо заменить вызов функции на её правую часть. Этот процесс называют встраиванием функций. Замена будет произведена только в случае полного применения функции, если синтаксическая арность (количество аргументов слева от знака равно) совпадает с числом переданных в функцию аргументов. Поэтому для GHC есть существенная разница между определениями:

```
(.) f g = \x -> f (g x)
```

```
(.) f g x = f (g x)
```

Встраиванием функций мы экономим на создании лишних объектов в куче, но при этом код может существенно разбухнуть. GHC пользуется эвристическим алгоритмом при определении когда функцию стоит встраивать, а когда – нет. По умолчанию GHC проводит встраивание только внутри модуля. Если мы компилируем с флагом `O`, функции будут встраиваться между модулями. Для этого GHC сохраняет в интерфейсном файле (с расширением `.hi`) не только типы функций, но и правые части достаточно кратких функций. Длина функции определяется числом узлов в синтаксическом дереве кода её правой части. Директивой `INLINE` мы приказываем GHC встроить функцию. Также есть более слабая версия этой прагмы `-INELINABLE`. Этой прагмой мы рекомендуем произвести встраивание функции не смотря на её величину.

Задать порог величины функции для встраивания можно с помощью флага `-funfolding-use-threshold=16`. Отметим, что если функция не является экспортируемой и используется лишь один раз, то GHC встроит её в любом случае, поэтому стоит определять списки экспортируемых определений в шапке модуля, иначе компилятор будет считать, что экспортируются все определения.

Прагма `INLINE` может стоять в любом месте, где можно было бы объявить тип значения. Так например можно указать компилятору встраивать методы класса:

```
instance Monad T where
  {-# INLINE return #-}
  return = ...
  {-# INLINE (>>=) #-}
  (>>=) = ...
```

Встраивание значений может существенно ускорить программу. Но не стоит венчать каждую экспортируемую функцию прагмой `INLINE`, возможно GHC встроит их автоматически. Посмотреть какие функции были встроены можно по определениям, попавшим в файл `.hi`.

Например если мы скомпилируем такой код с флагом `ddump-hi`:

```
module Inline(f, g) where
```

```
g :: Int -> Int
g x = x + 2
```

```
f :: Int -> Int
f x = g $ g x
```

то среди прочих определений увидим:

```
ghc -c -ddump-hi -O Inline.hs
...
f :: GHC.Types.Int -> GHC.Types.Int
{- Arity: 1, HasNoCafRefs, Strictness: U(L)m,
   Unfolding: InlineRule (1, True, False)
   (\ x :: GHC.Types.Int ->
     case x of wild { GHC.Types.I# x1 ->
       GHC.Types.I# (GHC.Prim.+# (GHC.Prim.+# x1 2) 2) }) -}
...
```

В этом виде прочесть функцию не так просто. Ко всем именам добавлены имена модулей. Приведём вывод к более простому виду с помощью флага `dsuppress-all`:

```
ghc -c -ddump-hi -dsuppress-all -O Inline.hs
...
f :: Int -> Int
{- Arity: 1, HasNoCafRefs, Strictness: U(L)m,
   Unfolding: InlineRule (1, True, False)
   (\ x :: Int -> case x of wild { I# x1 -> I# (+# (+# x1 2) 2) }) -}
...
```

Мы видим, что все вызовы функции `g` были заменены. Если вы всё же подозреваете, что GHC не справляется с встраиванием ваших часто используемых функций и это сказывается, попробуйте добавить к ним `INLINE`, но при этом лучше узнать, привело ли это к росту производительности, проверить с помощью профилирования.

Отметим также прагму `NOINLINE` с её помощью мы можем запретить встраивание функции. Эта прагма часто используется при различных трюках с `unsafePerformIO`, встраивание функции, которая содержит неконтролируемые побочные эффекты, может повлиять на её результат.

Прагма RULES

Разработчики GHC хотели, чтобы их компилятор был расширяемым и программист мог бы определять специфические для его приложения правила оптимизации. Для этого была придумана прагма `RULES`. За

счёт чистоты функций мы можем в очень простом виде выразить инварианты программы. Инвариант – это некоторое свойство значения, которое остаётся постоянным при некоторых преобразованиях. Наиболее распространённые инварианты имеют собственные имена. Например, это коммутативность сложения:

```
forall a b. a + b = b + a
```

Здесь мы пишем: для любых a и b изменение порядка следования аргументов у $(+)$ не влияет на результат. С ключевым словом `forall` мы уже когда-то встречались, когда говорили о типе `ST`. Помните тип функции `runST`? Пример свойства функции `map`:

```
forall f g. map f . map g = map (f . g)
```

Это свойство принято называть дистрибутивностью. Мы видим, что функция композиции дистрибутивна относительно функции `map`. Инварианты определяют скрытые закономерности значений. За счёт чистоты функций мы можем безболезненно заменить в любом месте программы левую часть на правую или наоборот. Оптимизация начинается тогда, когда мы понимаем, что одна из частей может быть вычислена гораздо эффективнее другой. Так в примере с `map` выражение справа от знака равно гораздо эффективнее, поскольку в нём мы не строим промежуточный список. Особенно ярко разница проявляется в энергичной стратегии вычислений. Или посмотрим на такое совсем простое свойство:

```
map id = id
```

Если мы заменим левую часть на правую, то число сэкономленных усилий будет пропорционально длине списка. Вряд ли программист станет писать такие выражения, однако они могут появиться после выполнения других оптимизаций, например после многих встраиваний различных функций.

Можно представить, что эти правила являются дополнительными уравнениями в определении функции:

```
map f []           = []
map f (x:xs)       = f x : map f xs
map id a           = a
map f (map g x)    = map (f . g) x
```

Словно теперь мы можем проводить сопоставление с образцом не только по конструкторам, но и по выражениям самого языка и функция `map` стала конструктором. Что интересно, зависимости могут быть какими угодно, они могут выражать закономерности, присущие той области, которую мы описываем. В дополнительных уравнениях мы подставляем аргументы так же как и в обычных, если где-нибудь в коде программы находится соответствие с левой частью уравнения, мы заменяем её на правую. При этом мы пишем правила так, чтобы действительно происходила оптимизация программы, поэтому слева пишется медленная версия.

Такие дополнительные правила пишутся в специальной прагме **RULES**:

```
{-# RULES
  "map/compose" forall f g x. map f (map g x) = map (f . g) x
  "map/id"      map id                = id
-#-}
```

Первым в кавычках идёт имя правила. Оно используется только для подсчёта статистики (например если мы хотим узнать сколько правил сработало в данном прогоне программы). За именем правила пишут уравнение. В одной прагме может быть несколько уравнений. Правила разделяются точкой с запятой или переходом на другую строку. Все свободные переменные правила перечисляются в окружении **forall** (...) .~. Компилятор доверяет нам абсолютно. Производится только проверка типов. Никаких других проверок не проводится.

Выполняется ли на самом деле это свойство, будет ли вычисление правой части действительно проще программы вычисления левой – известно только нам.

Отметим то, что прагма **RULES** применяется до тех пор пока есть возможность её применять, при этом мы можем войти в бесконечный цикл:

```
{-# RULES
  "infinite" forall a b. f a b = f b a
-#-}
```

С помощью прагмы **RULES** можно реализовать очень сложные схемы оптимизации. Так в `Prelude` реализуется слияние (`fusion`) списков. За счёт этой оптимизации многие выражения вида свёртка/развёртка

не будут производить промежуточных списков. Этой схеме будет посвящена отдельная глава. Например если список преобразуется серией функций `map`, `filter` и `foldr` промежуточные списки не строятся.

Посмотрим как работает прагма `RULES`, попробуем скомпилировать такой код:

```
module Main where

data List a = Nil | Cons a (List a)
    deriving (Show)

foldrL :: (a -> b -> b) -> b -> List a -> b
foldrL cons nil x = case x of
    Nil      -> nil
    Cons a as -> cons a (foldrL cons nil as)

mapL :: (a -> b) -> List a -> List b
mapL = undefined

{-# RULES
"mapL" forall f xs.
    mapL f xs = foldrL (Cons . f) Nil xs
#-}

main = print $ mapL (+100) $ Cons 1 $ Cons 2 $ Cons 3 Nil
```

Функция `mapL` не определена, вместо этого мы сделали косвенное определение в прагме `RULES`. Проверим, для того чтобы `RULES` заработали, необходимо компилировать с одним из флагов оптимизаций `O` или `O2`:

```
$ ghc --make -O Rules.hs
$ ./Rules
Rules: Prelude.undefined
```

Что-то не так. Дело в том, что GHC проводит встраивание простых функций. GHC слишком поторопился и заменил `mapL` на её определение. Также обратим внимание на то, что выражение не соответствует левой части правила. У нас:

```
mapL f xs      /=      mapL f $ xs
```

Функция \$ также является простой и GHC встраивает её. Для успешной замены нам необходимо, чтобы \$ встроился раньше mapL и чтобы наше правило сработало раньше встраивания mapL.

Фазы компиляции

Для решения этой проблемы в прагмы **RULES** и **INLINE** были введены ссылки на фазы компиляции. С помощью них мы можем указать GHC в каком порядке реагировать на эти прагмы. Фазы пишутся в квадратных скобках:

```
{-# INLINE [2] someFun #-}  
{-# RULES  
"fun" [0] forall ...  
"fun" [1] forall ...  
"fun" [~1] forall ...  
#-}
```

Компиляция выполняется в несколько фаз. Фазы следуют от некоторого заданного целого числа, например трёх, до нуля. Мы можем сослаться на фазу двумя способами: просто номером и номером с тильдой. Ссылка без тильды говорит: не применяй правило до наступления данной фазы, далее – применяй. Ссылка с тильдой говорит: попытайся применить это правило как можно раньше – до наступления данной фазы, далее – не применяй.

В нашем примере мы задержим встраивание для mapL и foldrL так:

```
{-# INLINE [1] foldrL #-}  
foldrL :: (a -> b -> b) -> b -> List a -> b  
  
{-# INLINE [1] mapL #-}  
mapL :: (a -> b) -> List a -> List b
```

Посмотреть какие правила сработали можно с помощью флага `ddump-rule-firings`. Теперь скомпилируем:

```
$ ghc --make -O Rules.hs -ddump-rule-firings  
...  
Rule fired: SPEC Main.$fShowList [GHC.Integer.Type.Integer]  
Rule fired: mapL  
Rule fired: Class op show  
...  
$ ./Rules  
Cons 101 (Cons 102 (Cons 103 Nil))
```

Среди прочих правил, определённых в стандартных библиотеках, сработало и наше.

Прагма UNPACK

Наш основной враг на этапе оптимизации программы это лишние объекты кучи. Чем меньше объектов мы создаём на пути к результату, тем эффективнее наша программа. С помощью прагмы `INLINE` мы можем избавиться от многих объектов, связанных с вызовом функции, это объекты типа `FUN`. Прагма `UNPACK` позволяет нам бороться с лишними объектами типа `CON`. В прошлой главе мы говорили о том, что значения в Haskell содержат дополнительную служебную информацию, которая необходима на этапе вычисления, например значение сначала было отложенным, потом мы до него добрались и вычислили, возможно оно оказалось не определённым значением (`undefined`). Такие значения называются запакованными (`boxed`). Незапакованное значение, это примитивное значение, как оно представлено в памяти компьютера. Вспомним определение целых чисел:

```
data Int = I# Int#
```

По традиции все незапакованные значения пишутся с решёткой на конце. Запакованные значения позволяют откладывать вычисления, пользоваться `undefined` при определении функции. Но за эту гибкость приходится платить. Вспомним расход памяти в выражении `[Pair 1 2]`

```
nil = []                -- глобальный объект (не в счёт)

let x1 = I# 1           -- 2 слова
    x2 = I# 2           -- 2 слова
    p  = Pair x1 x2     -- 3 слова
    val = Cons p nil    -- 3 слова
in val                 -----
                        -- 10 слов
```

Получилось десять слов для списка из одного элемента, который фактически хранит два значения. Размер списка, который хранит такие пары будет зависеть от числа элементов N как $10N$. Тогда как полезная нагрузка составляет $2N$. С помощью прагмы `UNPACK` мы можем отказаться от ленивой гибкости в пользу меньшего расхода

памяти. Эта прагма позволяет встраивать один конструктор в поле другого. Это поле должно быть строгим (с пометкой `!`) и мономорфным (тип поля должен быть конкретным типом, а не параметром), причём подчинённый тип должен содержать лишь один конструктор (у него нет альтернатив):

```
data PairInt = PairInt
  {-# UNPACK #-} !Int
  {-# UNPACK #-} !Int
```

Мы конкретизировали поля `Pair` и сделали их строгими с помощью восклицательных знаков. После этого значения из конструктора `Int` будут храниться прямо в конструкторе `PairInt`:

```
nil = []                -- глобальный объект (не в счёт)

let p  = PairInt 1 2    -- 3 слова
    val = Cons p nil    -- 3 слова
in val                 -----
                        -- 6 слов
```

Так мы сократим размер до `$6N$`. Но мы можем пойти ещё дальше. Если этот тип является ключевым типом нашей программы и мы рассчитываем на то, что в нём будет храниться много значений мы можем создать специальный список для таких пар и распаковать значение списка:

```
data ListInt = ConsInt {-# UNPACK #-} !PairInt
              | NilInt
```

```
nil = NilInt
```

```
let val = ConsInt 1 2 nil    -- 4 слова
in val                       -----
                              -- 4 слова
```

Значение будет встроено дважды и получится, что у нашего нового конструктора `Cons` уже три поля. Отметим, что эта прагма имеет смысл лишь при включённом флаге оптимизации `-O` или выше. Если мы не включим этот флаг, то компилятор не будет проводить встраивание функций, поэтому при вычислении функций вроде

```
sumPair :: PairInt -> Int
sumPair (Pair a b) = a + b
```

Плюс не будет встроен и вместо того, чтобы сразу сложить два числа с помощью примитивной функции, компилятор сначала запакует их в конструктор `I#` и затем применит функцию `+`, в которой опять распакует их, сложит и затем, снова запаковав, вернёт результат.

Компилятор автоматически запаковывает все такие значения при передаче в ленивую функцию, это может привести к снижению быстродействия даже при включённом флаге оптимизации, при недостаточном встраивании. Это необходимо учитывать. В таких случаях проводите профилирование, убедитесь в том, что оптимизация привела к повышению эффективности.

В стандартных библиотеках предусмотрено много незапакованных типов. Например это специальные кортежи. Они пишутся с решётками:

```
newtype ST s a = ST (STRep s a)
type STRep s a = State# s -> (# State# s, a #)
```

Это определение типа `ST`. Специальные кортежи используются для возврата нескольких значений напрямую, без создания промежуточного кортежа в куче. В этом случае значения будут сохранены в регистрах или на стеке. Для использования специальных значений необходимо активировать расширения `MagicHash` и `UnboxedTuples`

Разработчики различных библиотек могут предоставлять несколько вариантов своих данных: ленивые версии и незапакованные. Например в `ST`-массив незапакованных значений `STUArray s i a` эквивалентен массиву значений в `C`. В таком массиве можно хранить лишь примитивные типы.

Краткое содержание

Эта глава была посвящена компилятору GHC. Мы говорим Haskell подразумеваем GHC, говорим GHC подразумеваем Haskell. К сожалению на данный момент у этого компилятора нет достойных конкурентов. А может и к счастью, ведь если бы не было GHC, у нас была бы бурная конкуренция среди компиляторов поплоше. Мы бы не знали, что они не так хороши. Но у нас не было бы программ, которые способны тягаться по скорости с `C`. И мы бы говорили: ну декларативное

программирование, что поделаешь, за радость абстракций приходится платить. Но есть GHC! Всё-таки это очень трудно: написать компилятор для ленивого языка

Отметим другие компиляторы: Hugs разработан Марком Джонсом (написан на C), nhc98 основанный Николасом Райомо (Niklas Røjemo) этот компилятор задумывался как легковесный и простой в установке, он разрабатывался при поддержке NUTEK, Йоркского университета и Технического университета Чалмерса. От этого компилятора отпочковался UHC, Йоркский компилятор. UHC – компилятор Утрехтского университета, разработан для тестирования интересных идей в теории типов. JHC (Джон Мичэм, John Meacham) и LHC (Дэвид Химмельструп и Остин Сипп, David Himmelstrup, Austin Seipp) компиляторы предназначенные для проведения более сложных оптимизаций программ с помощью преобразований дерева программы.

В этой главе мы узнали как вычисляются программы в GHC. Мы узнали об этапах компиляции. Сначала проводится синтаксический анализ программы и проверка типов, затем код Haskell переводится на язык Core. Это сильно урезанная версия Haskell. После этого проводятся оптимизации, которые преобразуют дерево программы. На последнем этапе Core переводится на ещё более низкоуровневый, но всё ещё функциональный язык STG, который превращается в низкоуровневый код и исполняется вычислителем. Посмотреть на текст вашей программы в Core и STG можно с помощью флагов `ddump-simpl` `ddump-stg` при этом лучше воспользоваться флагом `ddump-suppress-all` для пропуска многочисленных деталей. Хардкорные разработчики Haskell смотрят Core для того чтобы понять насколько строгой оказалась та или иная функция, как аргументы размещаются в памяти. Но это уже высший пилотаж искусства оптимизации на Haskell.

Мы узнали о том как работает сборщик мусора и научились просматривать разные параметры работы программы. У нас появилось несколько критериев оценки производительности программ: минимум глубоких очисток и отсутствие горбов на графике изменения кучи. Мы потренировались в охоте за утечками памяти и посмотрели как разные типы профилирования могут подсказать нам в каком месте затаилась ошибка. Отметим, что не стоит в каждой медленной

программе искать утечку памяти. Так в примере `concat` у нас не было утечек памяти, просто один из алгоритмов работал очень плохо и через профилирование функций мы узнали какой.

Также мы познакомились с новыми прагмами оптимизации программ. Это встраиваемые функции `INLINE`, правила преобразования выражений `RULE` и встраиваемые конструкторы `UNPACK`. Разработчики GHC отмечают, что грамотное использование прагмы `INLINE` может существенно повысить скорость программы. Если мы встраиваем функцию, которая используется очень часто, нам не нужно создавать лишних отложенных вычислений при её вызовах.

Надеюсь, что содержание этой главы упростит понимание программ. Как они вычисляются, куда идёт память, почему она висит в куче. При оптимизации программ предпочитайте изменение алгоритма перед настройкой параметров компилятора под плохой алгоритм. Вспомните самый первый пример, увеличением памяти под сборку мусора нам удалось вытянуть ленивую версию `sum`, но ведь строгая версия требовала в 100 раз меньше памяти, причём её запросы не зависели от величины списка. Если бы мы остановились на ленивой версии, вполне могло бы так стать, что первый год нас бы устраивали результаты, но потом наши аппетиты могли возрасти. И вдруг программа, так тщательно настроенная, взорвалась. За год мы, конечно, многое позабыли о её внутренностях, искать ошибку было бы гораздо труднее. Впрочем не так безнадежно: включаем `auto-all`, `caf-all` с флагом `prof` и смотрим отчёт после флага `p`.

Упражнения

- Попробуйте понять причину утечки памяти в примере с функцией `sum2` на уровне STG. Не забывайте этот пример, вроде, ага, тут у нас копятся отложенные вычисления в аргументе. Переведите на STG и посмотрите в каком месте происходит слишком много вызовов `let`-выражений. Переведите и пример без утечки памяти, а также промежуточный вариант, который не сработал. Для этого вам понадобится выразить энергичный образец через функцию `seq`.

Подсказка: За счёт семантики `case`-выражений нам не нужно специальных конструкций для того чтобы реализовать `seq` в STG:

```
seq = FUN( a b ->
           case a of
             x -> b )
```

При этом вызов функции `seq` будет встроен. Необходимо будет заменить в коде все вызовы `seq` на правую часть определения (без `FUN`). Также обратите внимание на то, что `plus` не является примитивной функцией:

```
plusInt = FUN( ma mb ->
               case ma of
                 I# a -> case mb of
                           I# b -> case (primitivePlus a b) of
                               res -> I# res )
```

В этой функции всплыла на поверхность одна тонкость. Если бы мы писали это выражение в Haskell, то мы бы сразу вернули результат (`I# (primitivePlus a b)`), но мы пишем в STG и конструктор может принять только атомарное выражение. Тогда мы могли бы подумать и сохранить его по старинке в `let`-выражении:

```
-> let v = primitivePlus a b
    in I# v
```

Но это не правильное выражение в STG! Конструкция в правой части `let`-выражения должна быть объектом кучи, а у нас там простое выражение. Но было бы плохо добавить к нему `THUNK`, поскольку это выражение содержит вызов примитивной функции на незапакованных значениях. Эта операция выполняется очень быстро. Было бы плохо создавать для неё специальный объект на куче. Поэтому мы сразу вычисляем это выражение в третьем `case`. Эта функция также будет встроенной, необходимо заменить все вызовы на определение.

- Набейте руку в профилировании, пусть это станет привычкой. Вы долго писали большую программу и теперь вы можете узнать много подробностей из её жизни, что происходит с ней во время вычисления кода. Вернитесь к прошлой главе и попрофилируйте разные примеры. В конце главы мы

рассматривали пример с поиском корней, там мы создавали большой список промежуточных результатов и в нём искали решение. Я говорил, что такие алгоритмы очень эффективны при ленивой стратегии вычислений, но так ли это? Будьте критичны, не верьте на слово, ведь теперь у вас есть инструменты для проверки моих туманных гипотез.

- Откройте документацию к GHC. Прокрутите её. Проникнитесь уважением к разработчикам GHC. Найдите исходники GHC и почитайте их. Посмотрите на Haskell-код, написанный профессионалами. Выберите функцию `naугад` и попытайтесь понять как она строит свой результат.
- Откройте документацию вновь. Нас интересует глава **Profiling**. Найдите в разделе профилирование кучи как выполняется `retainer profiling`. Это специальный тип профилирования направленный на поиск данных, которые удерживают в памяти другие данные (типичный сценарий для утечек памяти). Разберитесь с этим типом профилирования (флаг `hr`).
- Постройте систему правил, которая выполняет слияние для списков **List**, определённых в примере для прагмы **RULES**. Сравните показатели производительности с правилами и без (для этого скомпилируйте дважды с флагом **O** и без) на тестовом выражении:

```
main = print $ sumL $  
    mapL (\x -> x - 1000) $ mapL (+100) $ mapL (*2) $ genL 0 1e6
```

Функция `sumL` находит сумму элементов в списке, функция `genL` генерирует список чисел с единичным шагом от первого аргумента до второго.

Подсказка: вам нужно воспользоваться такими свойствами (не забудьте о фазах компиляции)

```
mapL f (mapL g xs)           = ...  
foldrL cons nil (mapL f xs)  = ...
```

- Откройте исходный код **Prelude** и присмотритесь к различным прагмам. Попытайтесь понять почему они там используются.

Ленивые чудеса

В прошлой главе мы узнали, что такое ленивые вычисления. В этой главе мы посмотрим чем они хороши. С ними можно делать невозможные вещи. Обращаться к ещё не вычисленным значениям, работать с бесконечными данными.

Мы пишем программу, чтобы решить какую-нибудь сложную задачу. Часто так бывает, что сложная задача оказывается сложной до тех пор пока её не удаётся разбить на отдельные независимые подзадачи. Мы решаем задачи по-меньше, потом собираем из них решения, из этих решений собираем другие решения и вот уже готова программа. Но мы решаем задачу не на листочке, нам необходимо объяснить её компьютеру. И тот язык, на котором мы пишем программу, оказывает сильное влияние на то как мы будем решать задачу. Мы не можем разбить программу на независимые подзадачи, если в том языке на котором мы собираемся объяснять задачу компьютеру нет средств для того, чтобы собрать эти решения вместе.

Об этом говорит *Джон Хьюз* (John Hughes) в статье “Why functional programming matters”. Он приводит такую метафору. Если мы делаем стул и у нас нет хорошего клея. Единственное что нам остаётся это вырезать из дерева стул целиком. Это невероятно трудная задача. Гораздо проще сделать отдельные части и потом собрать вместе. Функциональные языки программирования предоставляют два новых вида “клея”. Это функции высшего порядка и ленивые вычисления. В статье можно найти много примеров. Некоторые из них мы рассмотрим в этой главе.

С функциями высших порядков мы уже знакомы, они позволяют склеивать небольшие решения. С их помощью мы можем параметризовать функцию другой функцией (поведением). Они дают нам возможность выделять сложные закономерности и собирать их в функции. Ленивые вычисления же предназначены для склеивания больших программ. Они синхронизируют выполнение подзадач, избавляя нас от необходимости выполнять это вручную.

Эта идея разбиения программы на независимые части приводит нас к понятию модульности. Когда мы решаем задачу мы пытаемся разложить её на простейшие составляющие. При этом часто оказывается, что эти составляющие применимы не только для нашей задачи, но и для многих других. Мы получаем целый букет решений, там где искали одно.

Численные методы

Рассмотрим несколько численных методов. Все эти методы построены на понятии сходимости. У нас есть последовательность решений и она сходится к одному решению, но мы не знаем когда. Мы только знаем, что промежуточные решения будут всё ближе и ближе к итоговому.

Поскольку у нас ленивый язык мы сначала построим все возможные решения, а затем выберем итоговое. Так же как мы делали это в прошлой главе, когда искали корни уравнения методом неподвижной точки. Эти примеры взяты из статьи “Why functional programming matters” Джона Хьюза.

Дифференцирование

Найдём производную функции в точке. Посмотрим на математическое определение производной:

$$f'(x) = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h}$$

Производная это предел последовательности таких отношений, при h стремящемся к нулю. Если предел сходится, то производная определена. Для того чтобы решить эту задачу мы начнём с небольшого значения h и будем постепенно уменьшать его, вычисляя промежуточные значения производной. Как только они перестанут сильно изменяться мы будем считать, что мы нашли предел последовательности

Этот процесс напоминает то, что мы делали при поиске корня уравнения методом неподвижной точки. Мы можем взять из того решения функцию определения сходимости последовательности:

```

converge :: (Ord a, Num a) => a -> [a] -> a
converge eps (a:b:xs)
  | abs (a - b) <= eps   = a
  | otherwise            = converge eps (b:xs)

```

Теперь осталось только создать последовательность значений производных. Напишем функцию, которая вычисляет промежуточные решения:

```

easydiff :: Fractional a => (a -> a) -> a -> a -> a
easydiff f x h = (f (x + h) - f x) / h

```

Мы возьмём начальное значение шага и будем последовательно уменьшать его вдвое:

```
halves = iterate (/2)
```

Соберём все части вместе:

```

diff :: (Ord a, Fractional a) => a -> a -> (a -> a) -> a -> a
diff h0 eps f x = converge eps $ map (easydiff f x) $ iterate (/2) h0
  where easydiff f x h = (f (x + h) - f x) / h

```

Сохраним эти определения в отдельном модуле и найдём производную какой-нибудь функции. Протестируем решение на экспоненте.

Известно, что производная экспоненты равна самой себе:

```

*Numeric> let exp' = diff 1 1e-5 exp
*Numeric> let test x = abs $ exp x - exp' x
*Numeric> test 2
1.4093421286887065e-5
*Numeric> test 5
1.767240203776055e-5

```

Интегрирование

Теперь давайте поинтегрируем функции одного аргумента. Интеграл это площадь кривой под графиком функции. Если бы кривая была прямой, то мы могли бы вычислить интеграл по формуле трапеций:

```

easyintegrate :: Fractional a => (a -> a) -> a -> a -> a
easyintegrate f a b = (f a + f b) * (b - a) / 2

```

Но мы хотим интегрировать не только прямые линии. Мы представим, что функция является ломаной прямой линией. Мы посчитаем интеграл на каждом из участков и сложим ответы. При этом чем ближе точки

друг к другу, тем точнее можно представить функцию в виде ломаной прямой линии, тем точнее будет значение интеграла.

Проблема в том, что мы не знаем заранее насколько близки должны быть точки друг к другу. Это зависит от функции, которую мы хотим проинтегрировать. Но мы можем построить последовательность решений. На каждом шаге мы будем приближать функцию ломаной прямой, и на каждом шаге число изломов будет расти вдвое. Как только решение перестанет меняться мы вернём ответ.

Построим последовательность решений:

```
integrate :: Fractional a => (a -> a) -> a -> a -> [a]
integrate f a b = easyintegrate f a b :
  zipWith (+) (integrate a mid) (integrate mid b)
  where mid = (a + b)/2
```

Первое решение является площадью под прямой, которая соединяет концы отрезка. Потом мы делим отрезок пополам, строим последовательность приближений и складываем частичные суммы с помощью функции zipWith.

Эта версия функции хоть и наглядная, но не эффективная. Функция f вычисляется заново при каждом рекурсивном вызове. Было бы хорошо вычислять её только для новых значений. Для этого мы будем передавать значения с предыдущего шага:

```
integrate :: Fractional a => (a -> a) -> a -> a -> [a]
integrate f a b = integ f a b (f a) (f b)
  where integ f a b fa fb = (fa+fb)*(b-a)/2 :
    zipWith (+) (integ f a m fa fm)
      (integ f m b fm fb)
  where m = (a + b)/2
        fm = f m
```

В этой версии мы вычисляем значения в функции f лишь один раз для каждой точки. Запишем итоговое решение:

```
int :: (Ord a, Fractional a) => a -> (a -> a) -> a -> a -> a
int eps f a b = converge eps $ integrate f a b
```

Мы опять воспользовались функцией converge, нам не нужно было её переписывать. Проверим решение. Для проверки также воспользуемся экспонентой. В прошлой главе мы узнали, что

```
$$e^x = 1 + \int_0^x e^t dt$$
```

Посмотрим, так ли это для нашего алгоритма:

```
*Numeric> let exp' = int 1e-5 exp 0
*Numeric> let test x = abs $ exp x - 1 - exp' x
*Numeric> test 2
8.124102876649886e-6
*Numeric> test 5
4.576306736225888e-6
*Numeric> test 10
1.0683757864171639e-5
```

Алгоритм работает. В статье ещё рассмотрены методы повышения точности этих алгоритмов. Что интересно для улучшения точности не надо менять существующий код. Функция принимает последовательность промежуточных решений и преобразует её.

Степенные ряды

Напишем модуль для вычисления степенных рядов. Этот пример взят из статьи Дугласа МакИлроя (Douglas McIlroy) “Power Series, Power Serious”. Степенной ряд представляет собой функцию, которая определяется списком коэффициентов:

```
$$F(x) = f_0 + f_1 x + f_2 x^2 + f_3 x^3 + f_4 x^4 + ...$$
```

Степенной ряд содержит бесконечное число слагаемых. Для вычисления нам потребуются функции сложения и умножения. Ряд $F(x)$ можно записать и по-другому:

```
$F(x)$      $=$ $F_0 (x)$
             $=$ $f_0 + x F_1 (x)$
             $=$ $f_0 + x (f_1 + x F_2 (x))$
```

Это определение очень похоже на определение списка. Ряд есть коэффициент f_0 и другой ряд $F_1(x)$ умноженный на x . Поэтому для представления рядов мы выберем конструкцию похожую на список:

```
data Ps a = a :+: Ps a
          deriving (Show, Eq)
```

Но в нашем случае списки бесконечны, поэтому у нас лишь один конструктор. Далее мы будем писать просто $f + x F_1$, без скобок для аргумента.

Определим вспомогательные функции для создания рядов:

```
p0 :: Num a => a -> Ps a
p0 x = x :+: p0 0

ps :: Num a => [a] -> Ps a
ps [] = p0 0
ps (a:as) = a :+: ps as
```

Обратите внимание на то, как мы дописываем бесконечный хвост нулей в конец ряда. Теперь давайте определим функцию вычисления ряда. Мы будем вычислять лишь конечное число степеней.

```
eval :: Num a => Int -> Ps a -> a -> a
eval 0 _ = 0
eval n (a :+: p) x = a + x * eval (n-1) p x
```

В первом случае мы хотим вычислить ноль степеней ряда, поэтому мы возвращаем ноль, а во втором случае значение ряда $a + x P$ складывается из числа a и значения ряда P умноженного на заданное значение.

Арифметика рядов

В результате сложения и умножения рядов также получается ряд. Также мы можем создать ряд из числа. Эти операции говорят о том, что мы можем сделать степенной ряд экземпляром класса `Num`.

Сложение

Рекурсивное представление ряда $f + x F$ позволяет нам очень кратко выражать операции, которые мы хотим определить. Теперь у нас нет бесконечного набора коэффициентов, у нас всего лишь одно число и ещё один ряд. Операции существенно упрощаются. Так сложение двух бесконечных рядов имеет вид:

$$f + G = (f + x F_1) + (g + x G_1) = (f+g) + x (F_1 + G_1)$$

Переведём на Haskell:

```
(f :+: fs) + (g :+: gs) = (f + g) :+: (fs + gs)
```

Умножение

Умножим два ряда:

$$F * G = (f + x F_1) * (g + x G_1) = f g + x (f G_1 + F_1 * G)$$

Переведём:

```
(.*) :: Num a => a -> Ps a -> Ps a
k .* (f :+: fs) = (k * f) :+: (k .* fs)

(f :+: fs) * (g :+: gs) = (f * g) :+: (f .* gs + fs * (g :+: gs))
```

Дополнительная операция (.) выполняет умножение всех коэффициентов ряда на число.

Класс Num

Соберём определения для методов класса Num вместе:

```
instance Num a => Num (Ps a) where
  (f :+: fs) + (g :+: gs) = (f + g) :+: (fs + gs)
  (f :+: fs) * (g :+: gs) = (f * g) :+: (f .* gs + fs * (g :+: gs))
  negate (f :+: fs) = negate f :+: negate fs
  fromInteger n = p0 (fromInteger n)
```

```
(.*) :: Num a => a -> Ps a -> Ps a
k .* (f :+: fs) = (k * f) :+: (k .* fs)
```

Методы `abs` и `signum` не определены для рядов. Обратите внимание на то, как рекурсивное определение рядов приводит к рекурсивным определениям функций для рядов. Этот приём очень характерен для Haskell. Поскольку наш ряд это число и ещё один ряд за счёт рекурсии мы можем воспользоваться операцией, которую мы определяем, на “хвостовом” ряде.

Деление

Результат деления Q удовлетворяет соотношению:

$$F = Q * G$$

Переписав F , G и Q в нашем представлении, получим

$$\begin{aligned} f + x F_1 &= (q + x Q_1) * G = qG + x Q_1 * G \\ &= q(g + x G_1) + x Q_1 * G \\ &= qg + x (q G_1 + Q_1 * G) \end{aligned}$$

Следовательно

$$\begin{aligned} q &= f/g \\ Q_1 &= (F_1 - q G_1)/G \end{aligned}$$

Если $g = 0$ деление имеет смысл только в том случае, если и $f = 0$. Переведём на Haskell:

```
instance (Eq a, Fractional a) => Fractional (Ps a) where
  (0 :+: fs) / (0 :+: gs) = fs / gs
  (f :+: fs) / (g :+: gs) = q :+: ((fs - q .* gs)/(g :+: gs))
    where q = f/g

fromRational x = p0 (fromRational x)
```

Производная и интеграл

Производная одного члена ряда вычисляется так:

$$\frac{d}{dx} x^n = n x^{n-1}$$

Из этого выражения по свойствам производной

$$\frac{d}{dx}(f(x) + g(x)) = \frac{d}{dx}f(x) + \frac{d}{dx}g(x)$$

$$\frac{d}{dx}(k * f(x)) = k * \frac{d}{dx} f(x)$$

мы можем получить формулу для всего ряда:

$$\frac{d}{dx} F(x) = f_1 + 2 f_2 x + 3 f_3 x^2 + 4 f_4 x^3 + \dots$$

Для реализации нам понадобится вспомогательная функция, которая будет обновлять значение дополнительного множителя n в выражении $n x^{n-1}$:

```
diff :: Num a => Ps a -> Ps a
diff (f :+: fs) = diff' 1 fs
    where diff' n (g :+: gs) = (n * g) :+: (diff' (n+1) gs)
```

Также мы можем вычислить и интеграл степенного ряда:

```
int :: Fractional a => Ps a -> Ps a
int (f :+: fs) = 0 :+: (int' 1 fs)
    where int' n (g :+: gs) = (g / n) :+: (int' (n+1) gs)
```

Элементарные функции

Мы можем выразить элементарные функции через операции взятия производной и интегрирования. К примеру уравнение для e^x выглядит так:

$$\frac{dy}{dx} = y$$

Проинтегрируем с начальным условием $y(0) = 1$:

$$y(x) = 1 + \int_0^x y(t) dt$$

Теперь переведём на Haskell:

```
expx = 1 + int expx
```

Кажется невероятным, но это и есть определение экспоненты. Так же мы можем определить и функции для синуса и косинуса:

$$\begin{aligned} \frac{d}{dx} \sin{x} &= \cos{x}, & \sin{0} &= 0, \\ \frac{d}{dx} \cos{x} &= -\sin{x}, & \cos{0} &= 1 \end{aligned}$$

Что приводит нас к:

```
sinx = int cosx  
cosx = 1 - int sinx
```

И это работает! Вычисление этих функций возможно за счёт того, что вне зависимости от аргумента функция `int` вернёт ряд, у которого первый коэффициент равен нулю. Это значение подхватывается и используется на следующем шаге рекурсивных вычислений.

Через синус и косинус мы можем определить тангенс:

```
tanx = sinx / cosx
```

Водосборы

В этом примере мы рассмотрим одну интересную технику рекурсивных вычислений, которая называется *мемоизацией* (memoization). Она заключается в том, что мы запоминаем все значения, с которыми вызывалась функция и, если с данным значением функция уже

вычислялась, просто используем значение из памяти, а если значение ещё не вычислялось, вычисляем его и сохраняем.

В ленивых языках программирования для мемоизации функций часто используется такой приём. Мы сохраняем все значения функции в некотором контейнере, а затем обращаемся к элементам. При этом значения сохраняются в контейнере и не перевычисляются. Это происходит за счёт ленивых вычислений. Что интересно вычисляются не все значения, а лишь те, которые нам действительно нужны, те которые мы извлекаем из контейнера хотя бы один раз.

Посмотрим на такой классический пример. Вычисление чисел Фибоначчи. Каждое последующее число ряда Фибоначчи равно сумме двух предыдущих. Наивное определение выглядит так:

```
fib :: Int -> Int
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

В этом определении число вычислений растёт экспоненциально. Для того чтобы вычислить `fib n` нам нужно вычислить `fib (n-1)` и `fib (n-2)`, для того чтобы вычислить каждое из них нам нужно вычислить ещё два числа, и так вычисления удваиваются на каждом шаге. Если мы вызовем в интерпретаторе `fib 40`, то вычислитель зависнет. Что интересно в этой функции вычисления пересекаются, они могут быть переиспользованы. Например для вычисления `fib (n-1)` и `fib (n-2)` нужно вычислить `fib (n-2)` (снова), `fib (n-3)`, `fib (n-3)` (снова) и `fib (n-4)`.

Если мы сохраним все значения функции в списке, каждый вызов функции будет вычислен лишь один раз:

```
fib' :: Int -> Int
fib' n = fibs !! n
  where fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

Попробуем вычислить для 40:

```
*Fib> fib' 40
102334155
*Fib> fib' 4040
700852629
```

Вычисления происходят мгновенно. Если задача состоит из множества подзадач, которые самоподобны и для вычисления последующих подзадач используются решения из предыдущих, стоит задуматься об использовании мемоизации. Такие задачи называются задачами *динамического программирования*. Вычисление чисел Фибоначчи яркий пример задачи динамического программирования.

Рассмотрим такую задачу. Дана прямоугольная “карта местности”, в каждой клетке целым числом указана высота точки. Необходимо разметить местность по следующим правилам:

- Из каждой клетки карты вода стекает не более чем в одном из четырёх возможных направлений (“север”, “юг”, “запад”, “восток”).
- Если у клетки нет ни одного соседа с высотой меньше её собственной высоты, то эта клетка – водосток, и вода из неё никуда дальше не течёт.
- Иначе вода из текущей клетки стекает на соседнюю клетку с минимальной высотой.
- Если таких соседей несколько, то вода стекает по первому из возможных направлений из списка “на север”, “на запад”, “на восток”, “на юг”.

Все клетки из которых вода стекает в один и тот же водосток принадлежат к одному бассейну водосбора. Необходимо отметить на карте все бассейны. Решение этой задачи встретилось мне в статье Дмитрия Астапова “Рекурсия+мемоизация = динамическое программирование”. Здесь оно и приводится с незначительными изменениями.

Карта местности представлена в виде двумерного массива, в каждой клетке которого отмечена высота точки, нам необходимо получить двумерный массив того же размера, который вместо высот содержит метки водостоков. Мы будем отмечать их буквами латинского алфавита в том порядке, в котором они встречаются при обходе карты сверху вниз, слева направо. Например:

1 2 3 4 5 6		a a a b b b
7 8 9 2 4 5		a a b b b b
3 5 3 3 6 7	->	c c d b b e
6 4 5 5 3 1		f g d b e e
2 2 4 5 3 7		f g g h h e

Для представления двумерного массива мы воспользуемся типом `Array` из стандартного модуля `Data.Array`. Тип `Array` имеет два параметра:

```
data Array i a
```

Первый указывает на индекс, а второй на содержание. Массивы уже встречались нам в главе о типе `ST`. Напомню, что подразумевается, что этот тип является экземпляром класса `Ix`, который описывает целочисленные индексы, вспомним его определение:

```
class Ord a => Ix a where
  range      :: (a, a) -> [a]
  index      :: (a, a) -> a -> Int
  inRange    :: (a, a) -> a -> Bool
  rangeSize  :: (a, a) -> Int
```

Первый аргумент у всех этих функций это пара, которая представляет верхнюю и нижнюю грань последовательности. Попробуйте догадаться, что делают методы этого класса по типам и именам.

Для двумерного массива индекс будет задаваться парой целых чисел:

```
import Data.Array

type Coord = (Int, Int)
type HeightMap = Array Coord Int
type SinkMap   = Array Coord Coord
type LabelMap  = Array Coord Char
```

Значение типа `HeightMap` хранит карту высот, значение типа `SinkMap` хранит в каждой координате, ту точку, которая является водостоком для данной точки. Тип `LabelMap` обозначает итоговую разметку водостоков. Начнём с функции:

```
flow :: HeightMap -> SinkMap
```

Мы будем решать эту задачу рекурсивно. Представим, что мы знаем водостоки для всех точек кроме данной. Для каждой точки мы можем узнать в какую сторону из неё стекает вода. При этом водосток для

следующей точки такой же как и для текущей. Если же из данной точки вода никуда не течёт, то она сама является водостоком. Мы определим эту функцию через комбинатор неподвижной точки `fix`.:

```
flow :: HeightMap -> SinkMap
flow arr = fix $ \result -> listArray (bounds arr) $
    map (\x -> maybe x (result !) $ getSink arr x) $
    range $ bounds arr
```

```
getSink :: HeightMap -> Coord -> Maybe Coord
```

Мы ищем решение в виде неподвижной точки функции, которая принимает карту стоков и возвращает карту стоков. Функция `getSink` по данной точке на карте вычисляет соседнюю точку, в которую стекает вода. Эта функция частично определена, поскольку для водостоков нет такой соседней точки, в которую бы утекала вода. Функция `listArray` конструирует значение типа `Array` из списка значений. Первым аргументом она принимает диапазон значений для индексов. Размеры массива совпадают с размерами карты высот, поэтому первым аргументом мы передаём `bounds arr`.

Теперь разберёмся с тем как заполняются значения в список. Сначала мы создаём список координат исходной карты высот с помощью выражения:

```
range $ bounds arr
```

После этого мы по координатам точек находим водостоки, причём сразу для всех точек. Это происходит в лямбда-функции:

```
\x -> maybe x (result !) $ getSink arr x
```

Мы принимаем текущую координату и с помощью функции `getSink` находим соседнюю точку, в которую убегает вода. Если такой точки нет, то в следующем выражении мы вернём исходную точку, поскольку в этом случае она и будет водостоком, а если такая соседняя точка всё-таки есть мы спросим результат из будущего. Мы обратимся к результату `(result !)`, посмотрим каким окажется водосток для соседней точки и вернём это значение. Поскольку за счёт ленивых вычислений значения результирующего массива вычисляются лишь один раз, после того как мы найдём водосток для данной точки этим результатом смогут воспользоваться все соседние точки. При этом

порядок обращения к значениям из будущих вычислений не играет роли.

Осталось только определить функцию поиска ближайшего стока и функцию разметки.

```
getSink :: HeightMap -> Coord -> Maybe Coord
getSink arr (x, y)
  | null sinks = Nothing
  | otherwise  = Just $ snd $ minimum $ map (\i -> (arr!i, i)) sinks
  where sinks = filter p [(x+1, y), (x-1, y), (x, y-1), (x, y+1)]
        p i   = inRange (bounds arr) i && arr ! i < arr ! (x, y)
```

В функции разметки мы воспользуемся ассоциативным массивом из модуля `Data.Map`. Функция `nub` из модуля `Data.List` убирает из списка повторяющиеся элементы. Затем мы составляем список пар из координат водостоков и меток и в самом конце размечаем исходный массив:

```
label :: SinkMap -> LabelMap
label a = fmap (m M.! ) a
  where m = M.fromList $ flip zip ['a' .. ] $ nub $ elems a
```

Ленивее некуда

Мы выяснили, что значение может редуцироваться только при сопоставлении с образцом и в специальной функции `seq`. Функцию `seq` мы можем применять, а можем и не применять. Но кажется, что в декомпозиции мы не можем уйти от необходимости проведения хотя бы одной редукции. Оказывается можем, в Haskell для этого предусмотрены специальные *ленивые образцы* (*lazy patterns*). Они обозначаются знаком тильда:

```
lazyHead :: [a] -> a
lazyHead ~(x:xs) = x
```

Перед скобками сопоставления с образцом пишется символ тильда. Этим мы говорим вычислителю: доверься мне, здесь точно такой образец, можешь даже не проверять дальше. Он и правда дальше не пойдёт. Например если мы напомним такое определение:

```
lazySafeHead :: [a] -> Maybe a
lazySafeHead ~(x:xs) = Just x
lazySafeHead []      = Nothing
```

Если мы подставим в эту функцию пустой список мы получим ошибку времени выполнения, вычислитель доверился нам в первом уравнении, а мы его обманули. Сохраним в модуле `Strict` и проверим:

```
Prelude Strict> :! ghc --make Strict
[1 of 1] Compiling Strict          ( Strict.hs, Strict.o )

Strict.hs:67:0:
  Warning: Pattern match(es) are overlapped
    In the definition of `lazySafeHead': lazySafeHead [] = ...
Prelude Strict> :l Strict
Ok, modules loaded: Strict.
Prelude Strict> lazySafeHead [1,2,3]
Just 1
Prelude Strict> lazySafeHead []
Just *** Exception: Strict.hs:(67,0)-(68,29): Irrefutable
pattern failed for pattern (x : xs)
```

При компиляции нам даже сообщили о том, что образцы в декомпозиции пересекаются. Но мы были упрямы и напоролись на ошибку, если мы поменяем образцы местами, то всё пройдет гладко:

```
Prelude Strict> :! ghc --make Strict
[1 of 1] Compiling Strict          ( Strict.hs, Strict.o )
Prelude Strict> :l Strict
Ok, modules loaded: Strict.
Prelude Strict> lazySafeHead []
Nothing
```

Отметим, что сопоставление с образцом в `let` и `where` выражениях является ленивым. Функцию `lazyHead` мы могли бы написать и так:

```
lazyHead a = x
  where (x:xs) = a

lazyHead a =
  let (x:xs) = a
  in  x
```

Посмотрим как используются ленивые образцы при построении потоков, или бесконечных списков. Мы будем представлять функции одного аргумента потоками значений с одинаковым шагом. Так мы будем представлять непрерывные функции дискретными сигналами. Считаем, что шаг дискретизации (или шаг между соседними точками) нам известен.

$f : \mathbb{R} \rightarrow \mathbb{R} \setminus \rightarrow f_n = f(n \tau), \quad n = 0, 1, 2, \dots$

Где τ – шаг дискретизации, а n пробегает все натуральные числа. Определим функцию решения дифференциальных уравнений вида:

$$\frac{dx}{dt} = f(t)$$

$$x(0) = \hat{x}$$

Символ \hat{x} означает начальное значение функции x .

Перейдём к дискретным сигналам:

$$\frac{x_n - x_{n-1}}{\tau} = f_n, \quad x_0 = \hat{x}$$

Где τ – шаг дискретизации, а x и f – это потоки чисел, индекс n пробегает от нуля до бесконечности по всем точкам функции, превращённой в дискретный сигнал. Такой метод приближения дифференциальных уравнений называют методом Эйлера. Теперь мы можем выразить следующий элемент сигнала через предыдущий.

$$x_n = x_{n-1} + \tau f_n, \quad x_0 = \hat{x}$$

Закодируем это уравнение:

```
-- шаг дискретизации
dt :: Fractional a => a
dt = 1e-3

-- метод Эйлера
int :: Fractional a => a -> [a] -> [a]
int x0 (f:fs) = x0 : int (x0 + dt * f) fs
```

Смотрите в функции `int` мы принимаем начальное значение `x0` и поток всех значений функции правой части уравнения, поток значений функции $f(t)$. Мы помещаем начальное значение в первый элемент результата, а остальные значения получаем рекурсивно.

Определим две вспомогательные функции:

```
time :: (Enum a, Fractional a) => [a]
time = [0, dt ..]

dist :: Fractional a => Int -> [a] -> [a] -> a
dist n a b = ( / fromIntegral n) $
    foldl' (+) 0 $ take n $ map abs $ zipWith (-) a b
```

Функция `time` пробегает все значения отсчётов шага дискретизации по времени. Это тождественная функция представленная в виде потока с шагом `dt`.

Функция проверки результата `dist` принимает два потока и по ним считает расстояние между ними. Эта функция говорит, что расстояние между двумя потоками в `n` первых точках равно сумме модулей разности между значениями потоков. Для того чтобы оценить среднее расхождение, мы делим в конце результат на число точек.

Также импортируем для удобства символьный синоним для `fmap` из модуля `Control.Applicative`.

```
import Control.Applicative((<$>))
...
```

Проверим функцию `int`. Для этого сохраним все новые функции в модуле `Stream.hs`. Загрузим модуль в интерпретатор и вычислим производную какой-нибудь функции. Найдём решение для правой части константы и проверим, что у нас получилась тождественная функция:

```
*Stream> dist 1000 time $ int 0 $ repeat 1
7.37188088351104e-17
```

Функции практически совпадают, порядок ошибки составляет 10^{-16} . Так и должно быть для линейных функций. Посмотрим, что будет если в правой части уравнения стоит тождественная функция:

```
*Stream> dist 1000 ((\t -> t^2/2) <$> time) $ int 0 time
2.497500000001403e-4
```

Решение этого уравнения равно функции $\frac{t^2}{2}$. Здесь мы видим, что результаты уже не такие хорошие.

Есть функции, которые определяются рекурсивно в терминах дифференциальных уравнений, например экспонента будет решением такого уравнения:

$$\frac{dx}{dt} = x$$

$$x(t) = x(0) + \int_0^t x(\tau) d \tau$$

Опишем это уравнение в Haskell:

```
e = int 1 e
```

Наше описание копирует исходное математическое определение. Добавим это уравнение в модуль `Stream` и проверим результаты:

```
*Stream> dist 1000 (map exp time) e
^CInterrupted.
```

К сожалению вычисление зависло. Нажмём `ctrl+c` и разберёмся почему. Для этого распишем вычисление потока чисел `e`:

```
=>      e                                -- раскроем e
      int 1 e                            -- раскроем int, во втором в аргументе
      int 1 e@(f:fs)                    -- int стоит декомпозиция,
                                         -- для того чтобы узнать какое уравнение
                                         -- для int выбрать нам нужно раскрыть
                                         -- второй аргумент, узнать корневой
                                         -- конструктор, раскроем второй аргумент:

=>      int 1 (int 1 e)
=>      int 1 (int 1e@(f:fs))           -- такая же ситуация
=>      int 1 (int 1 (int 1 e))
```

Проблема в том, что первый элемент решения мы знаем, мы передаём его первым аргументом и присоединяем к решению, но *справа* от знака равно. Но для того чтобы перейти в правую часть вычислителя нужно проверить все аргументы, в которых есть декомпозиция. И он начинает проверять, но слишком рано. Нам бы хотелось, чтобы он сначала присоединил к решению первый аргумент, а затем выполнял бы вычисления следующего элемента.

С помощью ленивых образцов мы можем отложить декомпозицию второго аргумента на потом:

```
int :: Fractional a => a -> [a] -> [a]
int x0 ~(f:fs) = x0 : int (x0 + dt * f) fs
```

Теперь мы видим:

```
*Stream> dist 1000 (map exp time) e
4.988984990735441e-4
```

Вычисления происходят. С помощью взаимно-рекурсивных функций мы можем определить функции синус и косинус:

```
sinx = int 0 cosx
cosx = int 1 (negate <$> sinx)
```

Эти функции описывают точку, которая бежит по окружности. Вот математическое определение:

$\frac{dx}{dt}$	$=$	y
$\frac{dy}{dt}$	$=$	$-x$
$x(0)$	$=$	0
$y(0)$	$=$	1

Проверим в интерпретаторе:

```
*Stream> dist 1000 (sin <$> time) sinx
1.5027460329809257e-4
*Stream> dist 1000 (cos <$> time) cosx
1.9088156807382827e-4
```

Так с помощью ленивых образцов нам удалось попасть в правую часть уравнения для функции `int`, не раскрывая до конца аргументы в левой части. С помощью этого мы могли ссылаться в сопоставлении с образцом на значение, которое ещё не было вычислено.

Краткое содержание

Ленивые вычисления повышают модульность программ. Мы можем в одной части программы создать все возможные решения, а в другой выбрать лучшие по какому-либо признаку. Также мы посмотрели на интересную технику написания рекурсивных функций, которая называется мемоизацией. Мемоизация означает, что мы не вычисляем повторно значения некоторой функции, а сохраняем их и используем в дальнейших вычислениях. Мы узнали новую синтаксическую конструкцию. Оказывается мы можем не только бороться с ленью, но и поощрять её. Лень поощряется ленивыми образцами. Они отменяют приведение к слабой заголовочной нормальной форме при декомпозиции аргументов. Они пишутся как обычные образцы, но со знаком тильда:

```
lazyHead ~(x:xs) = x
```

Мы говорим вычислителю: поверь мне, это значение может иметь только такой вид, потом посмотришь так ли это, когда значения тебе понадобятся. Поэтому ленивые образцы проходят сопоставление с образцом в любом случае.

Сопоставление с образцом в `let` и `where` выражениях является ленивым. Функцию `lazyHead` мы могли бы написать и так:

```
lazyHead a = x
  where (x:xs) = a
```

```
lazyHead a =
  let (x:xs) = a
  in x
```

Упражнения

Мы побывали на выставке ленивых программ. Присмотритесь ещё раз к решениям задач этой главы и подумайте какую роль сыграли ленивые вычисления в каждом из случаев, какие мотивы обыгрываются в этих примерах. Также подумайте каким было бы решение, если бы в Haskell использовалась стратегия вычисления по значению. Критически настроенные читатели могут с помощью профилирования проверить эффективность программ из этой главы.

Структурная рекурсия

Структурная рекурсия определяет способ построения и преобразования значений по виду типа (по составу его конструкторов). Функции, которые преобразуют значения мы будем называть *свёрткой* (fold), а функции которые строят значения – *развёрткой* (unfold). Эта рекурсия встречается очень часто, мы уже пользовались ею и не раз, но в этой главе мы остановимся на ней поподробнее.

Свёртка

Свёртку значения можно представить как процесс, который заменяет в дереве значения все конструкторы на подходящие по типу функции.

Логические значения

Вспомним определение логических значений:

```
data Bool = True | False
```

У нас есть два конструктора-константы. Любое значение типа `Bool` может состоять либо из одного конструктора `True`, либо из одного конструктора `False`. Функция свёртки в данном случае принимает две константы одинакового типа `a` и возвращает функцию, которая превращает значение типа `Bool` в значение `a`, заменяя конструкторы на переданные значения:

```
foldBool :: a -> a -> Bool -> a
foldBool true false = \b -> case b of
  True    -> true
  False   -> false
```

Мы написали эту функцию в композиционном стиле для того, чтобы подчеркнуть, что функция преобразует значение типа `Bool`.

Определим несколько знакомых функций через функцию свёртки, начнём с отрицания:

```
not :: Bool -> Bool
not = foldBool False True
```

Мы поменяли конструкторы местами, если на вход поступит `True`, то мы вернём `False` и наоборот. Теперь посмотрим на “и” и “или”:

```
(||), (&&) :: Bool -> Bool -> Bool

(||) = foldBool (const True) id
(&&) = foldBool id (const False)
```

Определение функций “и” и “или” через свёртки подчёркивает, что они являются взаимно обратными. Смотрите, эти функции принимают значение типа `Bool` и возвращают функцию `Bool -> Bool`. Фактически функция свёртки для `Bool` является `if`-выражением, только в этот раз мы пишем условие в конце.

Натуральные числа

У нас был тип для натуральных чисел Пеано:

```
data Nat = Zero | Succ Nat
```

Помните мы когда-то записывали определения типов в стиле классов:

```
data Nat where
  Zero :: Nat
  Succ :: Nat -> Nat
```

Если мы заменим конструктор `Zero` на значение типа `a`, то конструктор `Succ` нам придётся заменять на функцию типа `a -> a`, иначе мы не пройдем проверку типов. Представим, что `Nat` это класс:

```
data Nat a where
  zero :: a
  succ :: a -> a
```

Из этого определения следует функция свёртки:

```
foldNat :: a -> (a -> a) -> (Nat -> a)
foldNat zero succ = \n -> case n of
  Zero    -> zero
  Succ m  -> succ (foldNat zero succ m)
```

Обратите внимание на рекурсивный вызов функции `foldNat` мы обходим всё дерево значения, заменяя каждый конструктор. Определим знакомые функции через свёртку:

```
isZero :: Nat -> Bool
isZero = foldNat True (const False)
```

Посмотрим как вычисляется эта функция:

```
=>      isZero Zero
      True          -- заменили конструктор Zero

      isZero (Succ (Succ (Succ Zero)))
=>      const False (const False (const False True))
      False         -- заменили и Zero и Succ

=>      False
```

Что интересно за счёт ленивых вычислений на самом деле во втором выражении произойдёт лишь одна замена. Мы не обходим всё дерево, нам это и не нужно, а смотрим лишь на первый конструктор, если там **Succ**, то произойдёт замена на постоянную функцию, которая игнорирует свой второй аргумент и рекурсивного вызова функции свёртки не произойдёт, совсем как в исходном определении!

```
even, odd :: Nat -> Bool

even  = foldNat True  not
odd   = foldNat False not
```

Эти функции определяют чётность числа, здесь мы пользуемся тем свойством, что $\text{not} (\text{not } a) == a$.

Определим сложение и умножение:

```
add, mul :: Nat -> Nat -> Nat

add a = foldNat a      Succ
mul a = foldNat Zero (add a)
```

Maybe

Вспомним определение типа для результата частично определённых функций:

```
data Maybe a = Nothing | Just a
```

Перепишем словно это класс:

```
data Maybe a b where
  Nothing :: b
  Just    :: a -> b
```


Этот класс принимает два параметра, поскольку исходный тип `Maybe` принимает один. Теперь несложно догадаться как будет выглядеть функция свёртки, мы просто получим стандартную функцию `maybe`. Дадим определение экземпляра функтора и монады через свёртку:

```
instance Functor Maybe where
    fmap f = maybe Nothing (Just . f)

instance Monad Maybe where
    return      = Just
    ma >>= mf   = maybe Nothing mf ma
```

Списки

Функция свёртки для списков это функция `foldr`. Выведем её из определения типа:

```
data [a] = a : [a] | []
```

Представим, что это класс:

```
class [a] b where
    cons    :: a -> b -> b
    nil     :: b
```

Теперь получить определение для `foldr` совсем просто:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr cons nil = \x -> case x of
    a:as    -> a `cons` foldr cons nil as
    []      -> nil
```

Мы обходим дерево значения, заменяя конструкторы методами нашего воображаемого класса. Определим несколько стандартных функций для списков через свёртку.

Первый элемент списка:

```
head :: [a] -> a
head = foldr const (error "empty list")
```

Объединение списков:

```
(++) :: [a] -> [a] -> [a]
a ++ b = foldr (:) b a
```

В этой функции мы реконструируем заново первый список но в самом конце заменяем пустой список в хвосте а на второй аргумент, так и получается объединение списков. Обратите внимание на эту особенность, скорость выполнения операции (++) зависит от длины первого списка. Поэтому между двумя выражениями

```
((a ++ b) ++ c) ++ d
a ++ (b ++ (c ++ d))
```

Нет разницы в итоговом результате, но есть огромная разница по скорости вычисления! Второй гораздо быстрее. Убедитесь в этом! Реализуем объединение списка списков в один список:

```
concat :: [[a]] -> [a]
concat = foldr (++) []
```

Через свёртку можно реализовать и функцию преобразования списков:

```
map :: (a -> b) -> [a] -> [b]
map f = foldr (:) . f []
```

Если смысл выражения `(:) . f` не совсем понятен, давайте распишем его типы:

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

Напишем функцию фильтрации:

```
filter :: (a -> Bool) -> [a] -> [a]
filter p = foldr (\a as -> foldBool (a:as) as (p a)) []
```

Тут у нас целых две функции свёртки. Если значение предиката `p` истинно, то мы вернём все элементы списка, а если ложно отбросим первый элемент. Через `foldr` можно даже определить функцию с хвостовой рекурсией `foldl`. Но это не так просто. Всё же попробуем. Для этого вспомним определение:

```
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl f s []           = s
foldl f s (a:as)       = foldl f (f s a) as
```

Нам нужно привести это определение к виду `foldr`, нам нужно выделить два метода воображаемого класса списка `cons` и `nil`:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr cons nil = \x -> case x of
  a:as -> a `cons` foldr cons nil as
  []    -> nil
```

Перенесём два последних аргумента определения `foldl` в правую часть, воспользуемся лямбда-функциями и `case`-выражением:

```
foldl :: (a -> b -> a) -> [b] -> a -> a
foldl f = \x -> case x of
  []    -> \s -> s
  a:as  -> \s -> foldl f as (f s a)
```

Мы поменяли местами порядок следования аргументов (второго и третьего). Выделим тождественную функцию в первом уравнении `case`-выражения и функцию композиции во втором.

```
foldl :: (a -> b -> a) -> [b] -> a -> a
foldl f = \x -> case x of
  []    -> id
  a:as  -> foldl f as . (flip f a)
```

Теперь выделим функции `cons` и `nil`:

```
foldl :: (a -> b -> a) -> [b] -> a -> a
foldl f = \x -> case x of
  []    -> nil
  a:as  -> a `cons` foldl f as
  where nil = id
        cons = \a b -> b . flip f a
              = \a -> ( . flip f a)
```

Теперь запишем через `foldr`:

```
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl f s xs = foldr (\a -> ( . flip f a)) id xs s
```

Кажется мы ошиблись в аргументах, ведь `foldr` принимает три аргумента. Дело в том, что в функции `foldr` мы сворачиваем списки в функции, последний аргумент предназначен как раз для результирующей функции. Отметим, что из определения можно исключить два последних аргумента с помощью функции `flip`.

Вычислительные особенности `foldl` и `foldr`

Если посмотреть на выражение, которое получается в результате вычисления `foldr` и `foldl` можно понять почему они так называются.

В левой свёртке `foldl` скобки группируются влево, поэтому на конце `l` (`left`):

```
foldl f s [a1, a2, a3, a4] =
  (((s `f` a1) `f` a2) `f` a3) `f` a4
```

В правой свёртке `foldr` скобки группируются вправо, поэтому на конце `r` (`right`):

```
foldr f s [a1, a2, a3, a4]
  a1 `f` (a2 `f` (a3 `f` (a4 `f` s)))
```

Кажется, что если функция `f` ассоциативна

$$(a \text{ `f` } b) \text{ `f` } c = a \text{ `f` } (b \text{ `f` } c)$$

то нет разницы какую свёртку применять. Разницы нет по смыслу, но может быть существенная разница в скорости вычисления. Рассмотрим функцию `concat`, ниже два определения:

```
concat = foldl (++) []
concat = foldr (++) []
```

Какое выбрать? Результат и в том и в другом случае одинаковый (функция `++` ассоциативна). Стоит выбрать вариант с правой свёрткой. В первом варианте скобки будут группироваться влево, это чудовищно скажется на производительности. Особенно если в конце небольшие списки:

```
Prelude> let concatl = foldl (++) []
Prelude> let concatr = foldr (++) []
Prelude> let x = [1 .. 1000000]
Prelude> let xs = [x,x,x] ++ map return x
```

Последним выражением мы создали список списков, в котором три списка по миллиону элементов, а в конце миллион списков по одному элементу. Теперь попробуйте выполнить `concatl` и `concatr` на списке `xs`. Вы заметите разницу по скорости печати. Также для сравнения можно установить флаг: `:set +s`.

Также интересной особенностью `foldr` является тот факт, что за счёт ленивых вычислений `foldr` не нужно знать весь список, правая свёртка может работать и на бесконечных списках, в то время как

foldl не вернёт результат, пока не составит всё выражение. Например такое выражение будет вычислено:

```
Prelude> foldr (&&) undefined $ True : True : repeat False
False
```

За счёт ленивых вычислений мы отбросили оставшуюся (бесконечную) часть списка. По этим примерам может показаться, что левая свёртка такая не нужна совсем, но не все операции ассоциативны. Иногда полезно собирать результат в обратном порядке, например так в `Prelude` определена функция `reverse`, которая переворачивает список:

```
reverse :: [a] -> [a]
reverse = foldl (flip (:)) []
```

Деревья

Мы можем определить свёртку и для деревьев. Вспомним тип:

```
data Tree a = Node a [Tree a]
```

Запишем в виде класса:

```
data Tree a b where
  node :: a -> [b] -> b
```

В этом случае есть одна тонкость. У нас два рекурсивных типа: само дерево и внутри него – список. Для преобразования списка мы воспользуемся функцией `map`:

```
foldTree :: (a -> [b] -> b) -> Tree a -> b
foldTree node = \x -> case x of
  Node a as -> node a (map (foldTree node) as)
```

Найдём список всех меток:

```
labels :: Tree a -> [a]
labels = foldTree $ \a bs -> a : concat bs
```

Мы объединяем все метки из поддеревьев в один список и присоединяем к нему метку из текущего узла.

Сделаем дерево экземпляром класса `Functor`:

```
instance Functor Tree where
  fmap f = foldTree (Node . f)
```

Очень похоже на `map` для списков. Вычислим глубину дерева:

```
depth :: Tree a -> Int
depth = foldTree $ \a bs -> 1 + foldr max 0 bs
```

В этой функции за каждый узел мы прибавляем к результату единицу, а в списке находим максимум среди всех поддеревьев.

Развёртка

С помощью развёртки мы постепенно извлекаем значение рекурсивного типа из значения какого-нибудь другого типа. Этот процесс очень похож на процесс вычисления по имени. Сначала у нас есть отложенное вычисление или `thunk`. Затем мы применяем к нему функцию редукции и у нас появляется корневой конструктор. А в аргументах конструктора снова сидят `thunk`'и. Мы применяем редукцию к ним. И так пока не “развернём” всё значение.

Списки

Для разворачивания списков в `Data.List` есть специальная функция `unfoldr`. Присмотримся сначала к её типу:

```
unfoldr :: (b -> Maybe (a, b)) -> b -> [a]
```

Функция развёртки принимает стартовый элемент, а возвращает значение типа пары от `Maybe`. Типом `Maybe` мы кодируем конструкторы списка:

```
data [a] b where
  (:) :: a -> b -> b      -- Maybe (a, b)
  []  :: b               -- Nothing
```

Конструктор пустого списка не нуждается в аргументах, поэтому его мы кодируем константой `Nothing`. Объединение принимает два аргумента голову и хвост, поэтому `Maybe` содержит пару из головы и следующего элемента для разворачивания. Закодируем это определение:

```
unfoldr :: (b -> Maybe (a, b)) -> b -> [a]
unfoldr f = \b -> case (f b) of
  Just (a, b') -> a : unfoldr f b'
  Nothing      -> []
```

Или мы можем записать это более кратко с помощью свёртки `maybe`:

```
unfoldr :: (b -> Maybe (a, b)) -> b -> [a]
unfoldr f = maybe [] (\(a, b) -> a : unfoldr f b) . f
```

Смотрите, перед нами коробочка (типа `b`) с подарком (типа `a`), мы разворачиваем, а там пара: подарок (типа `a`) и ещё одна коробочка. Тогда мы начинаем разворачивать следующую коробочку и так далее по цепочке, пока мы не развернём не обнаружим `Nothing`, это означает что подарки кончились.

Типичный пример развёртки это функция `iterate`. У нас есть стартовое значение типа `a` и функция получения следующего элемента `a -> a`

```
iterate :: (a -> a) -> a -> [a]
iterate f = unfoldr $ \s -> Just (s, f s)
```

Поскольку `Nothing` не используется цепочка подарков никогда не оборвётся. Если только нам не будет лень их разворачивать. Ещё один характерный пример это функция `zip`:

```
zip :: [a] -> [b] -> [(a, b)]
zip = curry $ unfoldr $ \x -> case x of
  ([], _)      -> Nothing
  (_, [])      -> Nothing
  (a:as, b:bs) -> Just ((a, b), (as, bs))
```

Если один из списков обрывается, то прекращаем разворачивать. А если оба содержат голову и хвост, то мы помещаем в голову списка пару голов, а в следующий элемент для разворачивания пару хвостов.

Потоки

Для развёртки хорошо подходят типы у которых, всего один конструктор. Тогда нам не нужно кодировать альтернативы. Например рассмотрим потоки:

```
data Stream a = a :& Stream a
```

Они такие же как и списки, только без конструктора пустого списка. Функция развёртки для потоков имеет вид:

```

unfoldStream :: (b -> (a, b)) -> b -> Stream a
unfoldStream f = \b -> case f b of
    (a, b') -> a :& unfoldStream f b'

```

И нам не нужно пользоваться **Maybe**. Напишем функции генерации потоков:

```

iterate :: (a -> a) -> a -> Stream a
iterate f = unfoldStream $ \a -> (a, f a)

repeat :: a -> Stream a
repeat = unfoldStream $ \a -> (a, a)

zip :: Stream a -> Stream b -> Stream (a, b)
zip = curry $ unfoldStream $ \a :& as, b :& bs -> ((a, b), (as, bs))

```

Натуральные числа

Если присмотреться к натуральным числам, то можно заметить, что они очень похожи на списки. Списки без элементов. Это отражается на функции развёртки. Для натуральных чисел мы будем возвращать не пару а просто следующий элемент для развёртки:

```

unfoldNat :: (a -> Maybe a) -> a -> Nat
unfoldNat f = maybe Zero (Succ . unfoldNat f) . f

```

Напишем функцию преобразования из целых чисел в натуральные:

```

fromInt :: Int -> Nat
fromInt = unfoldNat f
  where f n
      | n == 0    = Nothing
      | n > 0     = Just (n-1)
      | otherwise = error "negative number"

```

Обратите внимание на то, что в этом определении не участвуют конструкторы для **Nat**, хотя мы и строим значение типа **Nat**. Конструкторы для **Nat** как и в случае списков кодируются типом **Maybe**. Развёртка используется гораздо реже свёртки. Возможно это объясняется необходимостью кодирования типа результата некоторым промежуточным типом. Определения теряют в наглядности. Смотрим на функцию, а там **Maybe** и не сразу понятно *что* мы строим: натуральные числа, списки или ещё что-то.

Краткое содержание

В этой главе мы познакомились с особым видом рекурсии. Мы познакомились со структурной рекурсией. Типы определяют не только значения, но и способы их обработки. Структурная рекурсия может быть выведена из определения типа. Есть языки программирования, в которых мы определяем тип и получаем функции структурной рекурсии в подарок. Есть языки, в которых структурная рекурсия является единственным возможным способом составления рекурсивных функций.

Обратите внимание на то, что в этой главе мы определяли рекурсивные функции, но рекурсия встречалась лишь в определении для функции свёртки и развёртки. Все остальные функции не содержали рекурсии, более того почти все они определялись в бесточечном стиле. Структурная рекурсия это своего рода комбинатор неподвижной точки, но не общий, а специфический для данного рекурсивного типа.

Структурная рекурсия бывает свёрткой и развёрткой.

Свёрткой (fold)

мы получаем значение некоторого произвольного типа из данного рекурсивного типа. При этом все конструкторы заменяются на функции, которые возвращают новый тип.

Развёрткой (unfold)

мы получаем из произвольного типа значение данного рекурсивного типа. Мы словно разворачиваем его из значения, этот процесс очень похож на ленивые вычисления.

Мы узнали некоторые стандартные функции структурной рекурсии: `cond` или `if`-выражения, `maybe`, `foldr`, `unfoldr`.

Упражнения

- Определите развёртку для деревьев из модуля `Data.Tree`.
- Определите с помощью свёртки следующие функции:

```
sum, prod  :: Num a => [a] -> a
or, and    :: [Bool] -> Bool
length     :: [a] -> Int
cycle      :: [a] -> [a]

unzip      :: [(a,b)] -> ([a],[b])
unzip3     :: [(a,b,c)] -> ([a],[b],[c])
```

- Определите с помощью развёртки следующие функции:

```
infinity   :: Nat
map         :: (a -> b) -> [a] -> [b]
iterateTree :: (a -> [a]) -> a -> Tree a
zipTree     :: Tree a -> Tree b -> Tree (a, b)
```

- Поэкспериментируйте в интерпретаторе с только что определёнными функциями и теми функциями, что мы определяли в этой главе.
- Рассмотрим ещё один стандартный тип. Он определён в `Prelude`. Это тип `Either` (дословно – один из двух). Этот тип принимает два параметра:

```
data Either a b = Left a | Right b
```

Значение может быть либо значением типа `a`, либо значением типа `b`. Часто этот тип используют как `Maybe` с информацией об ошибке. Конструктор `Left` хранит сообщение об ошибке, а конструктор `Right` значение, если его удалось вычислить.

Например мы можем сделать такие определения:

```
headSafe :: [a] -> Either String a
headSafe [] = Left "Empty list"
headSafe (x:_) = Right x

divSafe :: Fractional a => a -> a -> Either String a
divSafe a 0 = Left "division by zero"
divSafe a b = Right (a/b)
```

Для этого типа также определена функция свёртки она называется `either`. Не подглядывая в `Prelude`, определите её.

- Список является частным случаем дерева. Список это дерево, в каждом узле которого, лишь один дочерний узел. Деревья из модуля `Data.Tree` похожи на списки, но есть в них одно существенное отличие. Они всегда содержат хотя бы один элемент. Пустой список не может быть представлен в виде такого дерева. Например это различие сказывается, если вы захотите определить функцию-аналог `takeWhile` для деревьев.

Определите деревья, которые не страдают от этого недостатка. Определите для них функции свёртки/развёртки, а также функции, которые мы определили для стандартных деревьев. Определите функцию `takeWhile` (в рекурсивном виде и в виде развёртки) и сделайте их экземпляром класса `Monad`, похожий на экземпляр для списков.

Поиграем


Вот и закончилась первая часть книги. Мы узнали основные конструкции языка Haskell. В этой главе мы напишем законченную программу для игры в пятнашки. Ну или почти законченную, глава венчается упражнениями.

Стратегия написания программ

Описание задачи

Решение задачи начинается с описания проблемы и наброска решения. Мы хотим создать программу, в которой можно будет играть в пятнашки. Если вам не знакома эта игра, то взгляните на рисунок. Игра начинается с позиции, в которой все фишки перемешаны. Необходимо, переставляя фишки, вернуться в исходное положение. Каждым ходом мы двигаем одну фишку на пустое поле. В исходном положении фишки идут по порядку.

9	1	4	8
13		11	5
2	10	7	3
15	14	12	6



1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

Случайное и конечное состояние игры пятнашки

Программа будет перемешивать фишки и отображать поле для игры. Она будет спрашивать следующий ход и обновлять поле после хода. Если мы расставим все фишки по порядку, программа сообщит нам об этом и предложит начать новую игру. В каждый момент мы можем не только сделать ход, но и покинуть игру или начать всё заново. Известно, что не из любого положения можно расставить фишки по порядку. Поэтому наш алгоритм перемешивания должен генерировать только такие позиции, для которых решение возможно.

Набросок решения

Программа, которую мы хотим написать, будет вести диалог с пользователем. Она показывает поле для игры и спрашивает следующий ход. Потом она распознаёт ход, и показывает обновлённое поле. И так далее. Нам нужно как-то организовать этот диалог.

При этом в программе можно выделить две независимые части. Одна отвечает за сам диалог. Она принимает реплики пользователя и отображает поле для игры. А другая часть отвечает за правила игры пятнашки: как ходы влияют на поле, какое положение является победным, как перемешивать фишки.

У нас будет два отдельных модуля: один для описания игры, назовём его `Game`, а другой для описания диалога с пользователем. Мы назовём его `Loop` (петля или цикл), поскольку диалог это зацикленная процедура получения реплики и реакции на реплику.

Такой вот набросок-ориентир. После этого можно приступить к реализации. Но с чего начать?

Каркас. Типы и классы

В Haskell программы обычно начинают строить с каркаса – с типов и классов. Нам нужно выделить основные сущности и подумать какие типы подходят для их описания лучше всего.

В нашей задаче есть поле с фишками и ходы. Мы делаем ходы и фишки двигаются. Поле – это матрица или двумерный массив. У нас есть два индекса, которые пробегают значения от нуля до трёх. В каждой ячейке массива хранятся фишки. Фишки обозначаются целыми числами:

```
type Pos    = (Int, Int)
type Label  = Int
```

```
type Board  = Array Pos Label
```

Пустую фишку мы будем также обозначать числом. Физически когда мы ходим, мы меняем положение одной фишки. Но в нашем описании мы меняем местами две фишки, поскольку пустая фишка также обозначается номером. Когда мы ходим, мы меняем положение пустой

фишки, одним ходом мы можем сместить её вверх, вниз, влево или вправо. Введём специальный тип для обозначения ходов:

```
data Move = Up | Down | Left | Right
```

Для того чтобы при каждом ходе не искать пустую клетку, давайте сохраним её текущее положение. Тип `Game` будет содержать текущее положение пустой клетки и положение фишек:

```
data Game = Game {  
    emptyField :: Pos,  
    gameBoard  :: Board }
```

Вот и все типы для описания игры. Сохраним их в модуле `Game`. Теперь подумаем о типах для диалога с пользователем. В этом модуле наверняка будет много функций с типом `IO`, потому что в нём происходит взаимодействие с игроком. Но, что является каркасом для диалога?

Если мы хотим с кем-нибудь общаться, необходимо чтобы у нас был с собеседником общий язык, он и будет каркасом для диалога.

Вспомним, что мы ожидаем от пользователя. Пользователь может:

- Сделать ход
- Начать новую игру
- Выйти из игры

Если пользователь делает ход мы показываем новое положение поля, если он начинает новую игру мы показываем ему новую перемешанную позицию, давайте у нас будет разная степень перемешанности фигур. При перемешивании мы стартуем из победного положения и начинаем случайным образом делать ходы. Чем больше ходов мы сделаем тем сложнее будет собрать игру. Поэтому пользователь будет указывать число шагов для перемешивания при запросе новой игры. Если пользователь попросит закончить игру мы попрощаемся и выйдем из игры.

На основе этих рассуждений вырисовывается следующий тип для сообщений:

```
data Query = Quit | NewGame Int | Play Move
```

Значение типа `Query` (запрос) может быть константа `Quit` (выход), запрос новой игры `NewGame` с числом, которое указывает на сложность новой игры, также игрок может просто сделать ход `Play Move`.

А каков формат наших ответов? Все наши ответы на самом деле будут вызовами функции `putStrLn` мы будем отвечать пользователю изменениями экрана. Поэтому у нас нет специального типа для ответов. Итак у нас есть каркас, который можно начинать покрывать значениями. На этом этапе у нас есть два модуля. Это модуль `Loop`:

```
module Loop where

import Game

data Query = Quit | NewGame Int | Play Move
```

И модуль `Game`:

```
module Game where

import Data.Array

data Move = Up | Down | Left | Right
    deriving (Enum)

type Label = Int

type Pos = (Int, Int)

type Board = Array Pos Label

data Game = Game {
    emptyField  :: Pos,
    gameBoard   :: Board }
```

Ленивое программирование

Мы уже знаем как происходят ленивые вычисления. Мы принимаем выражение и начинаем очищать его от синонимов от корня к листьям или сверху вниз. Оказывается таким способом можно писать программы. Более того в функциональном программировании это очень распространённый подход. Мы начинаем со спецификации задачи (неформального описания) и потихоньку вытягиваем из него выражения языка `Haskell`. Начинаем мы с корня, с самой верхней

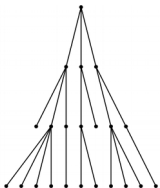
функции. Эта функция будет состоять из подвыражений. Когда мы напишем верхнюю функцию, мы перейдём к подвыражениям. И так мы будем спускаться пока не напишем всю программу.

Кажется, что такой подход очень не надёжен. Ведь мы сможем запустить программу только когда напишем её целиком. На каждом промежуточном шаге у нас есть неопределённые подвыражения. Получается, что очень долгое время мы будем писать программу, не зная работает она или нет.

Оказывается, что в Haskell есть решение этой проблемы. Нам поможет значение `undefined`. Мы будем писать только тип функции (и мысленно будем говорить, пусть она делает то-то), а вместо определения будем писать `undefined`. При этом конечно мы не сможем выполнять программу, вычислитель подорвётся на первом же значении, но мы сможем узнать осмысленна ли наша программа с точки зрения компилятора, проходит ли она проверку типов. В Haskell это большой плюс. Если программа прошла проверку типов, то скорее всего она будет работать.

Такой подход написания программ называется написанием сверху вниз. Мы начинаем с самой верхней функции и потихоньку вычищаем все `undefined`. Если вспомнить ленивые вычисления, то там роль `undefined` выполняли отложенные вычисления.

В чём преимущества такого подхода? Посмотрим на дерево. Если мы идём сверху вниз, то в самом начале у нас лишь одна задача, потом их становится всё больше и больше. Они дробятся, но источник у них один. Мы всегда знаем, что нам нужно чтобы закончить нашу задачу. Написать это, это и это подвыражение. Беда только в том, что это подвыражение содержит ещё больше подвыражений. Но сложные подвыражения мы можем оставить на потом и заняться другими. А потом, когда мы их доделаем может вдруг оказаться, что это сложное выражение нам и не нужно.



Если же мы начинаем идти из листьев, то у нас много отправных точек, которые должны сойтись в одной цели. При этом они могут и не сойтись, мы можем застрять в одной точке и потратить слишком много времени. И на остальные задачи у нас не хватит сил или мы можем потратить много времени на решение задачи, которая совсем не нужна для итогового решения. Также как и в вычислениях по значению, мы можем застрять на вычислении бесконечного значения, даже если в итоговом ответе нам понадобится лишь его малая часть.

Ещё один плюс решения сверху вниз состоит в экономии усилий. Мы можем написать всю программу в виде функций, которые состоят лишь из определений типов. И утрясти общую схему программы на типах. Также при реализации отдельных частей программы, мы можем воспользоваться упрощёнными алгоритмами, достаточными для тестирования приложения, оставив отрисовку деталей на потом. Мы не тратим время на реализацию, а смотрим как программа выглядит “в целом”. Если общий набросок нас устраивает мы можем начать заполнять дыры и детализировать отдельные выражения. Так мы будем детализировать-детализировать пока не придём к первоначальному решению. Далее если у нас останется время мы можем сменить реализацию некоторых частей. Но общая схема останется прежней, она уже устоялась на уровне типов. Часто такую стратегию разработки называют разработкой через прототипы (developing by prototyping). При этом процесс написания приложения можно представить как процесс сходимости, приближения к пределу. У нас есть серия промежуточных решений или прототипов, которые с каждым шагом всё точнее и точнее описывают итоговую программу. Также если мы работаем в команде, то дробление задачи на подзадачи происходит естественно, в ходе детализации, мы можем распределить нагрузку, распределив разные undefined между участниками проекта.

Слово undefined будет встречаться очень часто, буквально в каждом значении. Оно очень длинное, и часто писать его будет слишком утомительно. Определим удобный синоним. Я обычно использую un или lol (что-нибудь краткое и удобное для автоматического поиска):

```
un :: a
un = undefined
```

Но давайте приступим к реализации нашей игры. Самая верхняя функция, будет запускать программу. Назовём её play. Это функция взаимодействия с пользователем она ведёт диалог, поэтому её тип будет `IO ()`:

```
play :: IO ()
play = un
```

Итак у нас появилась корневая функция. Что мы будем в ней делать? Для начала мы поприветствуем игрока (функция greetings). Затем предложим ему начать игру (функция setup), после чего запустим цикл игры (функция gameLoop). Приветствие это просто надпись на экране, поэтому тип у него будет `IO ()`. Предложение игры вернёт стартовую позицию для игры, поэтому тип будет `IO Game`. Цикл игры принимает состояние и продолжает диалог. В типах это выражается так:

```
play :: IO ()
play = greetings >> setup >= gameLoop
```

```
greetings :: IO ()
greetings = un
```

```
setup :: IO Game
setup = un
```

```
gameLoop :: Game -> IO ()
gameLoop = un
```

Сохраним эти определения в модуле `Loop` и загрузим модуль с программой в интерпретатор:

```
Prelude> :l Loop
[1 of 2] Compiling Game           ( Game.hs, interpreted )
[2 of 2] Compiling Loop           ( Loop.hs, interpreted )
Ok, modules loaded: Game, Loop.
*Loop>
```

Модуль загрузился. Он потянул за собой модуль `Game`, потому что мы воспользовались типом `Move` из этого модуля. Программа прошла проверку типов, значит она осмысленна и мы можем двигаться дальше.

У нас три варианта дальнейшей детализации это функции greetings, setup и gameLoop. Мы пока пропустим greetings там мы напомним

какое-нибудь приветствие и сообщим игроку куда он попал и как ходить.

В функции `setup` нам нужно начать первую игру. Для начала игры нам нужно узнать её сложность, на сколько ходов перемешивать позицию. Это значит, что нам нужно спросить у игрока целое число. Мы спросим число функцией `getLine`, а затем попробуем его распознать. Если пользователь ввёл не число, то мы попросим его повторить ввод. Функция `readInt :: String -> Maybe Int` распознаёт число. Она возвращает целое число завёрнутое в `Maybe`, потому что строка может оказаться не числом. Затем это число мы используем в функции `shuffle` (перемешать), которая будет возвращать позицию, которая перемешана с заданной глубиной.

-- в модуль `Loop`

```
setup :: IO Game
setup = putStrLn "Начнём новую игру?" >>
      putStrLn "Укажите сложность (положительное целое число): " >>
      getLine >=> maybe setup shuffle . readInt
```

```
readInt :: String -> Maybe Int
readInt = un
```

-- в модуль `Game`:

```
shuffle :: Int -> IO Game
shuffle = un
```

Функция `shuffle` возвращает состояние игры `Game`, которое завёрнуто в `IO`. Оно завёрнуто в `IO`, потому что перемешивать позицию мы будем случайным образом, это значит, что мы воспользуемся функциями из модуля `Random`. Мы хотим чтобы каждая новая игра начиналась с новой позиции, поэтому скорее всего где-то в недрах функции `shuffle` мы воспользуемся `newStdGen`, которая и потянет за собой тип `IO`.

Игра перемешивается согласно правилам, поэтому функцию `shuffle` мы поселим в модуле `Game`. А функция `readInt` это скорее элемент взаимодействия с пользователем, ведь в ней мы распознаём число в строчном ответе, она останется в модуле `Loop`.

Проверим работает ли наша программа:

```
*Loop> :r
[1 of 2] Compiling Game           ( Game.hs, interpreted )
[2 of 2] Compiling Loop          ( Loop.hs, interpreted )
Ok, modules loaded: Game, Loop.
*Loop>
```

Работает! Можно спускаться по дереву выражения ниже. Сейчас нам предстоит написать одну из самых сложных функций, это функция `gameLoop`.

Пятнашки

Цикл игры

Функция цикла игры принимает текущую позицию. При этом у нас два варианта. Возможно игра пришла в конечное положение (`isGameOver`) и мы можем сообщить игроку о том, что он победил (`showResults`), если это не так, то мы покажем текущее положение (`showGame`), спросим ход (`askForMove`) и среагируем на ход (`reactOnMove`).

-- в модуль Loop

```
gameLoop :: Game -> IO ()
gameLoop game
  | isGameOver game    = showResults game >> setup >>= gameLoop
  | otherwise          = showGame game >> askForMove >>= reactOnMove game
```

```
showResults :: Game -> IO ()
showResults = un
```

```
showGame :: Game -> IO ()
showGame = un
```

```
askForMove :: IO Query
askForMove = un
```

```
reactOnMove :: Game -> Query -> IO ()
reactOnMove = un
```

-- в модуль Game

```
isGameOver :: Game -> Bool
isGameOver = un
```

Как определить закончилась игра или нет это скорее дело модуля `Game`. Все остальные функции принадлежат модулю `Loop`. Функция

`askForMove` возвращает реплику пользователя и тут же направляет её в функцию `reactOnMove`. Функции `showGame` и `showResults` ничего не возвращают, они только меняют состояния экрана. После того как игра закончится мы предложим игроку начать новую.

Обратите внимание на то, как даже не дав определение функции, мы всё же очерчиваем её смысл в объявлении типа. Так посмотрев на функцию `askForMove` и сопоставив тип с именем, мы можем понять, что эта функция предназначена для запроса значения типа `Query`, для запроса реплики пользователя. А по типу функции `showGame` мы можем понять, что она проводит какой-то побочный эффект, судя по имени что-то показывает, из типа видно что показывает значение типа `Game` или текущую позицию.

Отображение позиции

Определим функции отображения результата и позиции. Когда игра закончится мы покажем итоговое положение и объявим результат.

```
showResults :: Game -> IO ()
showResults g = showGame g >> putStrLn "Игра окончена."
```

Теперь определим функцию `showGame`. Если тип `Game` является экземпляром класса `Show`, то определение окажется совсем простым:

-- в модуль Loop

```
showGame :: Game -> IO ()
showGame = putStrLn . show
```

-- в модуль Game

```
instance Show Game where
    show = un
```

Реакция на реплики пользователя

Теперь нужно определить функции `askForMove` и `reactOnMove`. Первая функция требует установить протокол реплик пользователя, в каком виде он будет набирать значения типа `Query`. Нам пока лень об этом думать и мы перейдём к функции `reactOnMove`. Вспомним её тип:

```
reactOnMove :: Game -> Query -> IO ()
```

Функция принимает текущее положение и запрос пользователя. И ничего не возвращает, она продолжает игру. В любом случае в этой функции будет сопоставление с образцом по запросам пользователя так что можно написать:

```
reactOnMove :: Game -> Query -> IO ()
reactOnMove game query = case query of
  Quit      ->
  NewGame n ->
  Play      m ->
```

Рассмотрим каждый из случаев. В первом случае пользователь говорит, что ему надоело и он уже наигрался. Что ж попрощаемся и вернём значение единичного типа.

```
...
  Quit      -> quit
...

quit :: IO ()
quit = putStrLn "До встречи." >> return ()
```

В следующем варианте пользователь хочет начать всё заново. Так начнём!

```
  NewGame n -> gameLoop =<< shuffle n
```

Мы вызвали функцию перемешивания `shuffle` с заданным уровнем сложности. И рекурсивно вызвали цикл игры с новой позицией. Всё началось по новой. В третьей альтернативе пользователь делает ход, на это мы должны обновить позицию запустить цикл игры с новым значением:

```
-- в модуль Loop
  Play      m -> gameLoop $ move m game
-- в модуль Game
move :: Move -> Game -> Game
move = un
```

Функция `move` обновляет согласно правилам текущую позицию. Соберём все определения вместе:

```
reactOnMove :: Game -> Query -> IO ()
reactOnMove game query = case query of
  Quit      -> quit
  NewGame n -> gameLoop =<< shuffle n
  Play      m -> gameLoop $ move m game
```

Слушаем игрока

Теперь всё же вернёмся к функции `askForMove`, научимся слушать пользователя. Сначала мы скажем какую-нибудь вводную фразу, предложение ходить (`showAsk`) затем запросим строку стандартной функцией `getLine`, потом нам нужно будет распознать (`parseQuery`) в строке значение типа `Query`. Если распознать его нам не удастся, мы напомним пользователю как с нами общаться (`remindMoves`) и попросим сходить вновь:

```
askForMove :: IO Query
askForMove = showAsk >>
  getLine >>= maybe askAgain return . parseQuery
  where askAgain = wrongMove >> askForMove

parseQuery :: String -> Maybe Query
parseQuery = un

wrongMove :: IO ()
wrongMove = putStrLn "Не могу распознать ход." >> remindMoves

showAsk :: IO ()
showAsk = un

remindMoves :: IO ()
remindMoves = un
```

Механизм распознавания похож на случай с распознаванием числа. Значение завёрнуто в тип `Maybe`. И в самом деле функция определена лишь частично, ведь не все строки кодируют то, что нам нужно.

Функции `parseQuery` и `remindMoves` тесно связаны. В первой мы распознаём ввод пользователя, а во второй напоминаем пользователю как мы закодировали его запросы. Тут стоит остановиться и серьёзно подумать. Как закодировать значения типа `Query`, чтобы пользователю было удобно набирать их? Но давайте отвлечёмся от этой задачи, она слишком серьёзная. Оставим её на потом, а пока проверим не ушли ли мы слишком далеко, возможно наша программа потеряла смысл. Проверим типы!

```
*Loop> :r
[1 of 2] Compiling Game           ( Game.hs, interpreted )
[2 of 2] Compiling Loop           ( Loop.hs, interpreted )
Ok, modules loaded: Game, Loop.
```

Приведём код в порядок

Нам осталось дописать функции распознавания запросов и несколько маленьких функций с фразами и модуль `Loop` будет готов. Но перед тем как сделать это давайте упорядочим функции. Видно, что у нас выделилось несколько задач по типу общения с пользователем. У нас есть задачи, в которых мы что-то показываем пользователю, меняем состояние экрана и есть задачи, в которых мы просим от пользователя какие-то данные, ожидаем запросы функцией `getLine`. Также в самом верху выражения программы у нас расположены функции, которые координируют действия остальных, это третья группа. Сгруппируем функции по этому принципу.

Основные функции

```
play :: IO ()
play = greetings >> setup >>= gameLoop

gameLoop :: Game -> IO ()
gameLoop game
  | isGameOver game    = showResults game >> setup >>= gameLoop
  | otherwise          = showGame game >> askForMove >>= reactOnMove game

setup :: IO Game
setup = putStrLn "Начнём новую игру?" >>
  putStrLn "Укажите сложность (положительное целое число): " >>
  getLine >>= maybe setup shuffle . readInt
```

Запросы от пользователя (getLine)

```
reactOnMove :: Game -> Query -> IO ()
reactOnMove game query = case query of
  Quit      -> quit
  NewGame n -> gameLoop =<< shuffle n
  Play m    -> gameLoop $ move m game

askForMove :: IO Query
askForMove = showAsk >>
  getLine >>= maybe askAgain return . parseQuery
  where askAgain = wrongMove >> askForMove

parseQuery :: String -> Maybe Query
parseQuery = un

readInt :: String -> Maybe Int
readInt = un
```


Ответы пользователю (putStrLn)

```
greetings :: IO ()
greetings = un

showResults :: Game -> IO ()
showResults g = showGame g >> putStrLn "Игра окончена."

showGame :: Game -> IO ()
showGame = putStrLn . show

showAsk :: IO ()
showAsk = un

quit :: IO ()
quit = putStrLn "До встречи." >> return ()
```

По этим функциям видно, что нам немного осталось. Теперь вернёмся к запросам пользователя.

Формат запросов

Можно вывести с помощью **deriving** экземпляр класса **Read** для типа **Query** и читать их функцией **read**. Но это плохая идея, потому что пользователь нашей программы может и не знать Haskell. Лучше введём сокращённые имена для всех значений. Например такие:

```
left      -- Play Left
right     -- Play Righth
up        -- Play Up
down      -- Play Down

quit      -- Quit
new n     -- NewGame n
```

Можно обратить внимание на то, что все команды начинаются с разных букв. Воспользуемся этим и дадим пользователю возможность набирать команды одной буквой. Это приводит нас к таким определениям для функций разбора значения и напоминания ходов:

```
parseQuery :: String -> Maybe Query
parseQuery x = case x of
    "up"    -> Just $ Play Up
    "u"     -> Just $ Play Up
    "down"  -> Just $ Play Down
    "d"     -> Just $ Play Down
    "left"  -> Just $ Play Left
    "l"     -> Just $ Play Left
    "right" -> Just $ Play Right
```

```

"r"      -> Just $ Play Right
"quit"   -> Just $ Quit
"q"      -> Just $ Quit

'n':'e':'w':' ':n -> Just . NewGame =<< readInt n
'n':' ':n         -> Just . NewGame =<< readInt n
_             -> Nothing

```

```

remindMoves :: IO ()
remindMoves = mapM_ putStrLn talk
  where talk = [
    "Возможные ходы пустой клетки:",
    "  left      или l      -- налево",
    "  right     или r      -- направо",
    "  up        или u      -- вверх",
    "  down      или d      -- вниз",
    "Другие действия:",
    "  new int   или n int -- начать новую игру, int - целое число,",
    "                               "указывающее на сложность",
    "  quit      или q      -- выход из игры"]

```

Проверим работоспособность:

```

Prelude> :l Loop
[1 of 2] Compiling Game           ( Game.hs, interpreted )
[2 of 2] Compiling Loop           ( Loop.hs, interpreted )

```

```

Loop.hs:46:28:
  Ambiguous occurrence `Left'
  It could refer to either `Prelude.Left',
    imported from `Prelude' at Loop.hs:1:8-11
    (and originally defined in `Data.Either')
  or `Game.Left',
    imported from `Game' at Loop.hs:5:1-11
    (and originally defined at Game.hs:10:25-28)

```

```

Loop.hs:47:28:
  Ambiguous occurrence `Left'
  ...
  ...
Failed, modules loaded: Game.
*Game>

```

По ошибкам видно, что произошёл конфликт имён. Конструкторы `Left` и `Right` уже определены в `Prelude`. Это конструкторы типа `Either`. Давайте скроем их, добавим в модуль такую строчку:

```
import Prelude hiding (Either(..))
```

Теперь проверим:

```
*Game> :r
[2 of 2] Compiling Loop                ( Loop.hs, interpreted )
Ok, modules loaded: Game, Loop.
*Loop>
```

Всё работает, можно двигаться дальше.

Последние штрихи

В модуле `Loop` нам осталось определить несколько маленьких функций. Поиск по слову `in` говорит нам о том, что осталось определить функции ``

```
greetings    :: IO ()
readInt      :: String -> Maybe Int
showAsk      :: IO ()
```

Самая простая это функция `showAsk`, она приглашает игрока сделать ход:

```
showAsk :: IO ()
showAsk = putStrLn "Ваш ход: "
```

Теперь функция распознавания целого числа:

```
import Data.Char (isDigit)
...

readInt :: String -> Maybe Int
readInt n
  | all isDigit n = Just $ read n
  | otherwise     = Nothing
```

В первой альтернативе мы с помощью стандартной функции `isDigit :: Char -> Bool` проверяем, что строка состоит из одних только чисел. Если все символы числа, то мы пользуемся функцией из модуля `Read` и читаем целое число, иначе возвращаем `Nothing`.

Последняя функция, это функция приветствия. Когда игрок входит в игру он сталкивается с её результатами. Определим её так:

-- в модуль `Loop`

```
greetings :: IO ()
greetings = putStrLn "Привет! Это игра пятнашки" >>
  showGame initGame >>
  remindMoves
```

```
-- в модуль Game
```

```
initGame :: Game  
initGame = un
```

Сначала мы приветствуем игрока, затем показываем состояние (`initGame`), к которому ему нужно стремиться, и напоминаем как делаются ходы. На этом определении мы раскрыли все выражения в модуле `Loop`, нам остался лишь модуль `Game`.

Правила игры

Определим модуль `Game`, но мы будем определять его не с чистого листа. Те функции, которые нам нужны уже определились в ходе описания диалога с пользователем. Нам нужно уметь составлять начальное состояние `initGame`, уметь составлять перемешанное состояние игры `shuffle`, нам нужно уметь реагировать на ходы `move`, определять какая позиция является выигрышной `isGameOver` и уметь показывать фишки в красивом виде. Приступим!

```
initGame    :: Game  
shuffle     :: Int -> IO Game  
isGameOver  :: Game -> Bool  
move        :: Move -> Game -> Game
```

```
instance Show Game where  
    show = un
```

Таков наш план.

Начальное состояние

Начнём с самой простой функции, составим начальное состояние:

```
initGame :: Game  
initGame = Game (3, 3) $ listArray ((0, 0), (3, 3)) $ [0 .. 15]
```

Мы будем кодировать фишки цифрами от нуля до 14, а пустая клетка будет равна 15. Это просто соглашения о внутреннем представлении фишек, показывать мы их будем совсем по-другому.

С этим значением мы можем легко определить функцию определения конца игры. Нам нужно только добавить `deriving (Eq)` к типу `Game`. Тогда функция `isGameOver` примет вид:

```
isGameOver :: Game -> Bool
isGameOver = (== initGame)
```

Делаем ход

Напишем функцию:

```
move :: Move -> Game -> Game
```

Она обновляет позицию после хода. В пятнашках не во всех позициях доступны все ходы. Если пустышка находится на краю, мы не можем вывести её за пределы доски. Это необходимо как-то учесть. Каждый ход задаёт направление обмена фишками. Если у нас есть текущее положение пустышки и ход, то по ходу мы можем узнать направление, а по направлению ту фишку, которая займёт место пустышки после хода. При этом нам необходимо проверять находится ли та фишка, которую мы хотим поместить на пустое место в пределах доски. Например если пустышка расположена в самом верху и мы хотим сделать ход Up (передвинуть её ещё выше), то положение игры не должно измениться.

```
import Prelude hiding (Either(..))
newtype Vec = Vec (Int, Int)

move :: Move -> Game -> Game
move m (Game id board)
    | within id' = Game id' $ board // updates
    | otherwise  = Game id board
  where id' = shift (orient m) id
        updates = [(id, board ! id'), (id', emptyLabel)]
-- определение того, что индексы внутри доски
within :: Pos -> Bool
within (a, b) = p a && p b
  where p x = x >= 0 && x <= 3
-- смещение положение по направлению
shift :: Vec -> Pos -> Pos
shift (Vec (va, vb)) (pa, pb) = (va + pa, vb + pb)
-- направление хода
orient :: Move -> Vec
orient m = Vec $ case m of
    Up       -> (-1, 0)
    Down     -> (1, 0)
    Left     -> (0, -1)
    Right    -> (0, 1)
-- метка для пустой фишки
emptyLabel :: Label
emptyLabel = 15
```

Маленькие функции `within`, `shift`, `orient`, `emptyLabel` делают как раз то, что подписано в комментариях. Думаю, что их определение не сложно понять. Но есть одна тонкость, поскольку в функции `orient` мы пользуемся конструкторами `Left` и `Right` необходимо спрятать тип `Either` из `Prelude`. Мы ввели дополнительный тип `Vec` для обозначения смещения, чтобы случайно не подставить вместо него индексы.

Разберёмся с функцией `move`. Сначала мы вычисляем положение фишки, которая пойдёт на пустое место `id'`. Мы делаем это, сместив (`shift`) положение пустышки (`id`) по направлению хода (`orient a`).

Мы обновляем массив, который описывает доску с помощью специальной функции `//`. Посмотрим на её тип:

```
(//) :: Idx i => Array i a -> [(i, a)] -> Array i a
```

Она принимает массив и список обновлений в этом массиве.

Обновления представлены в виде пары индекс-значение. В охранном выражении мы проверяем, если индекс перемещаемой фишки в пределах доски, то мы возвращаем новое положение, в котором пустышка уже находится в положении `id'` и массив обновлён. Мы составляем список обновлений `updates` из двух элементов, это перемещения фишки и пустышки. Если же фишка за пределами доски, то мы возвращаем исходное положение.

Перемешиваем фишки

Игра начинается с такого положения, в котором все фишки перемешаны. Но перемешивать фишки произвольным образом было бы не честно, поскольку известно, что в пятнашках половина расстановок не приводит к выигрышу. Поэтому мы будем перемешивать так: мы стартуем из начального положения и делаем несколько ходов произвольным образом. Количество ходов определяет сложность игры:

```
shuffle :: Int -> IO Game  
shuffle n = (iterate (shuffle1 =<<) $ pure initGame) !! n  
shuffle1 :: Game -> IO Game  
shuffle1 = un
```

Функция `shuffle1` перемешивает фишки один раз. С помощью функции `iterate` мы строим список расстановок, которые мы получаем на

каждом шаге перемешивания. В самом конце мы выбираем из списка n-тую позицию. Обратите внимание на то, что мы не можем просто написать:

```
iterate shuffle1 initGame
```

Так у нас не совпадут типы. Для функции `iterate` нужно чтобы вход и выход функции имели одинаковые типы. Поэтому мы пользуемся в функции `iterate` методами классов `Monad` и `Applicative` (глава 6).

Теперь определим функцию `shuffle1`. Мы делаем ход в текущей позиции, который мы выбрали случайным образом из списка доступных ходов. Выбором случайного элемента из списка, будет заниматься функция `randomElem`, а функция `nextMoves` будет возвращать список доступных ходов для данного положения:

```
shuffle1 :: Game -> IO Game
shuffle1 g = flip move g <$> (randomElem $ nextMoves g)
```

```
randomElem :: [a] -> IO a
randomElem = un
```

```
nextMoves :: Game -> [Move]
nextMoves = un
```

Нам осталось определить всего две функции, и всё готово для игры. Определим выбор случайного элемента из списка:

```
import System.Random
...
```

```
randomElem :: [a] -> IO a
randomElem xs = (xs !! ) <$> randomRIO (0, length xs - 1)
```

Мы генерируем случайное число в диапазоне индексов списка и затем извлекаем элемент. Теперь функция определения ходов в текущем положении:

```
nextMoves g = filter (within . moveEmptyTo . orient) allMoves
  where moveEmptyTo v = shift v (emptyField g)
        allMoves = [Up, Down, Left, Right]
```

Мы выполняем схожие операции с теми, что были в функции `move`. Мы фильтруем из списка всех ходов те, что выводят пустую фишку за пределы доски.

Отображение положения

Я немного поторопился, нам осталась ещё одна функция. Это отображение позиции. Я не буду подробно останавливаться на теле функции, скажу лишь то, что она составляет строку так как это показано в комментарии к функции.

```
-- +---+---+---+---+
-- | 1 | 2 | 3 | 4 |
-- +---+---+---+---+
-- | 5 | 6 | 7 | 8 |
-- +---+---+---+---+
-- | 9 | 10 | 11 | 12 |
-- +---+---+---+---+
-- | 13 | 14 | 15 |   |
-- +---+---+---+---+
```

```
instance Show Game where
  show (Game _ board) = "\n" ++ space ++ line ++
    (foldr (\a b -> a ++ space ++ line ++ b) "\n" $ map column [0 .. 3])
  where post id = showLabel $ board ! id
        showLabel n = cell $ show $ case n of
          15 -> 0
          n   -> n+1
        cell "0" = "   "
        cell [x] = ' ':':':x':':':[]
        cell [a,b] = ' ':':':a':':':b':':':[]
        line = "+---+---+---+---+\n"
        nums = ((space ++ "|") ++ ) . foldr (\a b -> a ++ "|" ++ b) "\n".
              map post
        column i = nums $ map (\x -> (i, x)) [0 .. 3]
        space = "\t"
```

Теперь мы можем загрузить модуль `Loop` в интерпретатор и набрать `play`. Немного отвлечёмся и поиграем.

```
Prelude> :l Loop
[1 of 2] Compiling Game           ( Game.hs, interpreted )
[2 of 2] Compiling Loop           ( Loop.hs, interpreted )
Ok, modules loaded: Loop, Game.
*Loop> play
Привет! Это игра пятнашки
```

```
+---+---+---+---+
| 1 | 2 | 3 | 4 |
+---+---+---+---+
| 5 | 6 | 7 | 8 |
+---+---+---+---+
| 9 | 10 | 11 | 12 |
+---+---+---+---+
| 13 | 14 | 15 |   |
+---+---+---+---+
```


Возможные ходы пустой клетки:

left	или l	-- налево
right	или r	-- направо
up	или u	-- вверх
down	или d	-- вниз

Другие действия:

new int или n int -- начать новую игру, int - целое число,
указывающее на сложность

quit или q -- выход из игры

Начнём новую игру?

Укажите сложность (положительное целое число):

5

+	---	+	---	+	---	+	---	+
	1		2		3		4	
+	---	+	---	+	---	+	---	+
	5		6		7		8	
+	---	+	---	+	---	+	---	+
	9				10		11	
+	---	+	---	+	---	+	---	+
	13		14		15		12	
+	---	+	---	+	---	+	---	+

Ваш ход:

r

+	---	+	---	+	---	+	---	+
	1		2		3		4	
+	---	+	---	+	---	+	---	+
	5		6		7		8	
+	---	+	---	+	---	+	---	+
	9		10				11	
+	---	+	---	+	---	+	---	+
	13		14		15		12	
+	---	+	---	+	---	+	---	+

Ваш ход:

r

+	---	+	---	+	---	+	---	+
	1		2		3		4	
+	---	+	---	+	---	+	---	+
	5		6		7		8	
+	---	+	---	+	---	+	---	+
	9		10		11			
+	---	+	---	+	---	+	---	+
	13		14		15		12	
+	---	+	---	+	---	+	---	+

Ваш ход:

d

1	2	3	4	
5	6	7	8	
9	10	11	12	
13	14	15		

Игра окончена.

Ураа, получилось. Мы так долго писали программу, проверяя лишь типы, и в самом конце, когда мы закончили определение, всё работает. Конечно не всё работает так гладко, я уже написал эту программу и объясняю готовое решение, но когда общая схема программы утряслась, возможные ошибки определяются на раз. Мы могли вызвать отображение позиции не в том порядке или забыть проверку конца игры, всё это несколько строчек изменений.

Самые неприятные ошибки происходят, когда в середине выясняется, что мы ошиблись с типами. Типы, которые мы выбрали не могут описать явление, возможно мы не можем делать какие-то операции, которые нам, как неожиданно выяснилось, очень нужны. Это значит, что нужно менять каркас. Менять каркас, это значит сносить весь дом и строить новый. Возможно разрушения окажутся локальными, мы строим не дом, а город. И сносить придётся не всё, а несколько кварталов. Но это тоже большие перемены. Поэтому шаг определения типов очень важен. Впрочем сносить кварталы в Haskell одно удовольствие, поскольку стоит нам изменить какой-нибудь тип, например убрать какой-нибудь тип или изменить имя, компилятор тут же подскажет нам какие функции стали бессмысленными. Более коварные изменения связаны с добавлением конструктора-альтернативы. Например нам вдруг не понравился тип `Bool` и мы решили сделать его более человечным. Мы решили добавить ещё одно значение:

```
data Bool = True | False | IDonTKnow
```

Это может привести к неполному рассмотрению альтернатив в `case`-выражениях и сопоставлениях с образцом в аргументах функции. Такие ошибки крайне неприятны, поскольку они происходят на этапе выполнения программы, когда новое значение `IDonTKnow` дойдёт до `case`. В этом случае нам на выручку может прийти функция свёртки, если мы вместе с типом изменим и функцию свёртки, это скажется на всех функциях, которые были определены через неё. Чем больше таких функций, тем больше ошибок мы поймаем.

Упражнения

- Измените диалог с пользователем. Сделайте так чтобы у игры было главное меню, в котором игрок выбирает разные побочные функции, вроде выхода, начать новую игру, подсказка и игровое меню, в котором игрок только передвигает фишки. Когда игрок собирает игру он попадает в главное меню.
- Добавьте в игру подсчёт статистики. Если игрок дошёл до победной позиции он узнаёт за сколько ходов ему удалось решить задачу. Также ведётся история предыдущих попыток, по которой пользователь может следить как изменяются его результаты.
- Подумайте можно ли выделить интерфейс игры в отдельный класс так, чтобы модуль `Loop` не зависел от конкретной реализации игры. Чтобы можно было, опираясь на абстрактные методы, вроде `show` для `Game`, или реакции на ход, вести диалог с пользователем. Попробуйте переписать игру пятнашки с помощью такого класса.
- Попробуйте написать другую игру, например игру раскладывания пасьянса, крестики-нолики или шашки, не меняя модуля `Loop`. Так чтобы вы сделали необходимые экземпляры для классов из предыдущего упражнения, а всё остальное поведение следовало из них.

Лямбда-исчисление

В этой главе мы узнаем о лямбда-исчислении. Лямбда-исчисление описывает понятие алгоритма. Ещё до появления компьютеров в 30-е годы двадцатого века математиков интересовал вопрос о возможности создания алгоритма, который мог бы на основе заданных аксиом дать ответ о том верно или нет некоторое логическое высказывание. Например у нас есть базовые утверждения и логические связки такие как “и”, “или”, “для любого из”, “существует один из”, с помощью которых мы можем строить из базовых высказываний составные. Некоторые из них окажутся ложными, а другие истинными. Нам интересно узнать какие. Но для решения этой задачи прежде всего необходимо было понять а что же такое алгоритм?

Ответ на этот вопрос дали Алонсо Чёрч (Alonso Church) и Алан Тьюринг (Alan Turing). Чёрч разработал лямбда-исчисление, а Тьюринг теорию машин Тьюринга. Оказалось, что задача автоматического определения истинности формул в общем случае не имеет решения.

В основе лямбда-исчисления лежит понятие функции. Мы можем составлять сложные функции из простейших, а также подставлять в функции аргументы, которые могут быть как константами так и другими функциями. Как только мы составили выражение мы можем передать его вычислителю. Он подставляет аргументы в функции и возвращает такое выражение, в котором невозможно далее проводить подстановки аргументов. Этот процесс проведения подстановок считается вычислением алгоритма.

В рамках теории машин Тьюринга алгоритм описывается по-другому. Машина Тьюринга имеет внутреннее состояние, Состояние содержит некоторое значение, которое изменяется по ходу работы машины. Машина живёт не сама по себе, она читает ленту символов. Лента символов – это большая цепочка букв. На каждую букву машина реагирует серией действий. Она может изменить значение состояния, обновить букву в ленте или перейти к следующему или предыдущему символу. Есть состояния, которые обозначают конец работы, они называются терминальными. Как только машина дойдёт до

терминального состояния мы считаем, что вычисление алгоритма закончилось. После этого мы можем считать результат из состояний машины.

Функциональные языки программирования основаны на лямбда-исчислении. Поэтому мы будем говорить именно об этом описании алгоритма.

Лямбда исчисление без типов

Составление термов

Можно считать, что лямбда исчисление это такой маленький язык программирования. В нём есть множество символов, которые считаются переменными, они что-то обозначают и неделимы. В лямбда-исчислении программный код называется термом. Для написания программного кода у нас есть всего три правила:

- Переменные x , y , z ... являются термами.
- Если M и N – термы, то (MN) – терм.
- Если x – переменная, а M – терм, то $(\lambda x. M)$ – терм

В формальном описании добавляют ещё одно правило, оно говорит о том, что других термов нет. Первое правило, говорит о том, что у нас есть алфавит символов, который что-то обозначает, эти символы являются базовыми строительными блоками программы. Второе и третье правила говорят о том как из базовых элементов получаются составные. Второе правило – это правило применения функции к аргументу. В нём M обозначает функцию, а N обозначает аргумент. Все функции являются функциями одного аргумента, но они могут принимать и возвращать функции. Поэтому применение трёх аргументов к функции Fun будет выглядеть так:

$((Fun\ Arg1)\ Arg2)\ Arg3$

Третье правило говорит о том как создавать функции. Специальный символ лямбда (λ) в выражении $(\lambda x. M)$ говорит о том, что мы собираемся определить функцию с аргументом x и

телом функции $\lambda x. x$. С такими функциями мы уже сталкивались. Это безымянные функции. Приведём несколько примеров функций. Начнём с самого простого, определим тождественную функцию:

$\lambda x. x$

Функция принимает аргумент x и тут же возвращает его в теле. Теперь посмотрим на константную функцию:

$\lambda x. (\lambda y. x)$

Константная функция является функцией двух аргументов, поэтому наш терм принимает переменную x и возвращает другой терм функцию $\lambda y. x$. Эта функция принимает y , а возвращает x . В Haskell мы бы написали это так:

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

Точка сменилась на стрелку, а лямбда потеряла одну ножку. Теперь определим композицию. Композиция принимает две функции одного аргумента и направляет выход второй функции на вход первой:

$\lambda f. (\lambda g. (\lambda x. (f(gx))))$

Переменные f и g – это функции, которые участвуют в композиции, а x это вход результирующей функции. Уже в таком простом выражении у нас пять скобок на конце. Давайте введём несколько соглашений, которые облегчат написание термов:

	Пишем	Подразумеваем
Опустим внешние скобки:	$\lambda x. x$	$\lambda x. x$

В применении группируем скобки влево:	$fghx$	$((fg)h)x$
---------------------------------------	--------	------------

В функциях группируем скобки	$\lambda x. \lambda y. x$	$\lambda x. (\lambda y. x)$
------------------------------	---------------------------	-----------------------------

Пишем

Подразумеваем

вправо:

Пишем

функции

нескольких
аргументов

с одной

лямбдой:

$\lambda xy. x$

$(\lambda x. (\lambda y. x))$

С этими соглашениями мы можем переписать терм для композиции так:

$\lambda f g x. f(gx)$

Сравните с выражением на языке Haskell:

```
\f g x -> f (g x)
```

Выражения очень похожи. Haskell иногда называют засахаренной версией лямбда исчисления. В лямбда-исчислении мы не будем ставить пробелы для применения аргументов к функции. Мы будем считать, что все имена однобуквенные. При этом переменные мы будем писать с маленькой буквы, а составные термы с большой.

Определим ещё несколько функций. Например так выглядит функция flip:

$\lambda fxy. fyx$

Или можно записать в более явном виде, выделим функцию двух аргументов:

$\lambda f. \lambda xy. fyx$

Определим функцию on, она принимает функцию двух аргументов ff и функцию одного аргумента f , а возвращает функцию двух аргументов, в которой к аргументам сначала применяется функция f , а затем они передаются в функцию ff :

$\lambda * f. \lambda x. *(fx)(fx)$

В лямбда-исчислении есть только префиксное применение поэтому мы написали $*(fx)(fx)$ вместо привычного $(fx)*(fx)$. Здесь операция $*$ это не только умножение, а любая бинарная функция.

Абстракция

Функции в лямбда-исчислении называют абстракциями. Мы берём терм M и параметризуем его по переменной x в выражении $\lambda x. M$. При этом если в терме M встречается переменная x , то она становится связанной. Например в терме $\lambda x. \lambda y. x$ Переменная x является *связанной*, но в терме $\lambda y. x$, она уже не связана. Такие переменные называют *свободными*. Множество связанных переменных терма M мы будем обозначать $BV(M)$ от англ. bound variables, а множество свободных переменных мы будем обозначать $FV(M)$ от англ. free variables.

На интуитивном уровне процесс абстракции заключается в том, что мы смотрим на несколько частных случаев и видим в них что-то общее. Это общее мы выделяем в функцию, которая параметризована частностями. Например мы видим выражения:

$$\lambda x. x + x, \quad \lambda x. x * x$$

И в том и в другом у нас есть функция двух аргументов $+$ или $*$ и мы делаем из неё функцию одного аргумента. Мы можем абстрагировать (параметризовать) это поведение в такую функцию:

$$\lambda b. \lambda x. b \ x \ x$$

На Haskell мы бы записали это так:

```
\b -> \x -> b x x
```

Редукция. Вычисление термов

Процесс вычисления термов заключается в подстановке аргументов во все функции. Выражения вида:

$$(\lambda x. M) \ N$$

Заменяются на

$$M[x = N]$$

Эта запись означает, что в терме M все вхождения x заменяются на терм N . Этот процесс называется *редукцией* терма. А выражения вида $(\lambda x. M) \ N$ называются *редексами*. Проведём к примеру редукцию терма:

$\lambda b. \lambda x. b \ xx$ * λ

Для этого нам нужно в терме $\lambda x. b \ xx$ заменить все вхождения переменной b на переменную $*$. После этого мы получим терм:

$\lambda x. * \ xx$

В этом терме нет редексов. Это означает, что он вычислен или находится в *нормальной форме*.

α -преобразование

При подстановке необходимо следить за тем, чтобы у нас не появлялись лишние связывания переменных. Например рассмотрим такой редекс:

$\lambda x \ y. x$ \ y

После подстановки за счёт совпадения имён переменных мы получим тождественную функцию:

$\lambda y. y$

Переменная y была свободной, но после подстановки стала связанной. Необходимо исключить такие случаи. Поскольку с ними получается, что имена связанных переменных в определении функции влияют на её смысл. Например смысл такого выражения

$\lambda x \ z. x$ \ y

После подстановки будет совсем другим. Но мы всего лишь изменили обозначение локальной переменной y на z . И смысл изменился, для того чтобы исключить такие случаи пользуются переименованием переменных или *α -преобразованием*. Для корректной работы функций необходимо следить за тем, чтобы все переменные, которые были свободными в аргументе, остались свободными и после подстановки.

β -редукция

Процесс подстановки аргументов в функции называется *β -редукцией*. В редексе $\lambda x. M$ N вместо свободных

вхождений x в M мы подставляем N . Посмотрим на правила подстановки:

$x[x=N]$	$\rightarrow N$
$y[x=N]$	y
$(PQ)[x=N]$	$(P[x=N] \setminus Q[x=N])$
$(\lambda y. P)[x=N]$	$(\lambda y. P[x=N]), \quad y \notin FV(N)$
$(\lambda x. P)[x=N]$	$(\lambda x. P)$

Первые два правила определяют подстановку вместо переменных. Если переменная совпадает с той, на место которой мы подставляем терм N , то мы возвращаем терм N , иначе мы возвращаем переменную:

$x[x=N]$	$\rightarrow N$
$y[x=N]$	y

Подстановка применения термов равна применению термов, в которых произведена подстановка:

$$(PQ)[x=N] \rightarrow (P[x=N] \setminus Q[x=N])$$

При подстановке в лямбда-функции необходимо учитывать связность переменных. Если переменная аргумента отличается от той переменной на место которой происходит подстановка, то мы заменяем в теле функции все вхождения этой переменной на N :

$$(\lambda y. P)[x = N] \rightarrow (\lambda y. P[x=N]), \quad y \notin FV(N)$$

Условие $y \notin FV(N)$ означает, что необходимо следить за тем, чтобы в N не оказалось свободной переменной с именем y , иначе после подстановки она окажется связанной. Если такая переменная в N всё-таки окажется мы проведём α -преобразование в терме $\lambda y. M$ и заменим y на какую-нибудь другую переменную.

В последнем правиле мы ничего не меняем, поскольку переменная x оказывается связанной. А мы проводим подстановку только вместо свободных переменных:

$$(\lambda x. P)[x = N] \rightarrow (\lambda x. P)$$

Отметим, что не любой терм можно вычислить, например у такого терма нет нормальной формы:

$$(\lambda x. x x)(\lambda x. x x)$$

На каждом шаге редукции мы будем вновь и вновь возвращаться к исходному терму.

Стратегии редукции

В главе о ленивых вычислениях нам встретились две стратегии вычисления выражений. Это вычисление по имени и вычисление по значению. Также там мы узнали о том, что ленивые вычисления это улучшенная версия вычисления по имени, в которой аргументы функций вычисляются не более одного раза.

Эти стратегии вычисления пришли из лямбда-исчисления. Если нам нужно избавиться от всех редексов в выражении, то с какого редекса лучше начать? В вычислении по значению (*аппликативная стратегия*) мы начинаем с самого левого редекса, который не содержит других редексов, то есть с самого маленького подвыражения. А в вычислении по имени (*нормальная стратегия*) мы начинаем с самого левого внешнего редекса. Левый редекс означает, что в записи выражения он находится ближе всех к началу выражения.

Теорема (Карри)

Если у терма есть нормальная форма, то последовательное сокращение самого левого внешнего редекса приводит к ней.

Эта теорема говорит о том, что стратегия вычисления по имени может вычислить все термы, которые имеют нормальную форму. В том, что вычисление по значению может не справиться с некоторыми такими термами мы можем на следующем примере:

$$(\lambda x y. x) \lambda z. ((\lambda x. x x) (\lambda x. x x))$$

Этот терм имеет нормальную форму $\lambda z. z$ несмотря на то, что мы передаём вторым аргументом в константную функцию терм, у которого нет нормальной формы. Алгоритм вычисления по значению зависнет при вычислении второго аргумента. В то время как алгоритм вычисления по имени начнёт с самого внешнего терма и там определит, что второй аргумент не нужен.

Ещё один важный результат в лямбда-исчислении был сформулирован в следующей теореме:

Теорема (Чёрча-Россера)

Если терм M редуцируется к термам N_1 и N_2 , то существует терм L , к которому редуцируются и терм N_1 и терм N_2 .

Эта теорема говорит о том, что у терма может быть только одна нормальная форма. Поскольку если бы их было две, то существовал третий терм, к которому можно было бы редуцировать эти нормальные формы. Но по определению нормальной формы, мы не можем её редуцировать. Из этого следует, что нормальные формы должны совпадать.

Теорема Чёрча-Россера указывает на способ сравнения термов. Для того чтобы понять равны термы или нет, необходимо привести их к нормальной форме и сравнить. Если термы совпадают в нормальной форме, значит они равны.

Рекурсия. Комбинатор неподвижной точки

В лямбда-исчислении все функции являются безымянными. Это означает, что мы не можем в теле функции вызвать саму функцию, ведь мы не можем на неё сослаться, кажется, что у нас нет возможности строить рекурсивные функции. Однако это не так. Нам на помощь придёт комбинатор неподвижной точки. По определению комбинатор неподвижной точки решает задачу: для терма F найти терм X такой, что

$$F X = X$$

Существует много комбинаторов неподвижной точки. Рассмотрим Y -комбинатор:

$\lambda f. (\lambda x. f(xx)) (\lambda x. f(xx))$

Убедимся в том, что для любого терма F , выполнено тождество:
 $F(YF) = YF$:

$YF = (\lambda x. F(xx)) (\lambda x. F(xx)) = F (\lambda x. F(xx)) (\lambda x. F(xx)) = F(YF)$

Так с помощью Y -комбинатора можно составлять рекурсивные функции.

Кодирование структур данных

Вы наверное заметили, что пока мы составляли лишь обобщённые функции. Эти функции комбинируют другие функции, они не выполняют никаких действий над элементами. Что если нам захочется вычислять логические значения или воспользоваться числами?

Оказывается, что логические значения, числа, пары, списки и другие конструкции могут быть закодированы с помощью термов лямбда-исчисления. Тезис Чёрча утверждает, что с помощью лямбда-терма можно представить любую вычислимую числовую функцию. В 1936 году Чёрч с помощью лямбда-исчисления доказал существование неразрешимых проблем в теории чисел. Из этого следовала неразрешимость арифметики и неразрешимость исчисления логики предикатов первого порядка. Система аксиом называется разрешимой в том случае, если существует такой алгоритм, который позволяет по виду формулы определить следует ли она из заданных аксиом или нет.

Посмотрим как с помощью термов кодируются структуры данных. Далее для сокращения записи мы будем считать, что в лямбда исчислении можно определять синонимы с помощью знака равно. Запись $N = M$ говорит о том, что мы дали обозначение N терму M . Этой операции нет в лямбда-исчислении, но мы будем пользоваться ею для удобства.

Логические значения

Суть логических значений заключается в операторе If , с помощью которого мы можем организовывать ветвление алгоритма. Есть два

терма `$True$` и `$False$`, которые для любых термов `a` и `b`, обладают свойствами:

$$\text{\$If\ True\ a\ b\$} \quad \text{\$=\$} \quad \text{\$a\$}$$
$$\text{\$If\ False\ a\ b\$} \quad \text{\$=\$} \quad \text{\$b\$}$$

Термы `$True$`, `$False$` и `If`, удовлетворяющие таким свойствам выглядят так:

$$\text{\$True\$} \quad \text{\$=\$} \quad \text{\$\lambda t\ f .t\$}$$
$$\text{\$False\$} \quad \text{\$=\$} \quad \text{\$\lambda t\ f .f\$}$$
$$\text{\$If\$} \quad \text{\$=\$} \quad \text{\$\lambda b\ x\ y .b x y\$}$$

Проверим выполнение свойств:

$$\text{\$If\ True\ a\ b \rightarrow (\lambda b\ x\ y .b x y) (\lambda t\ f .t)\ a\ b \rightarrow (\lambda t\ f .t)\ a\ b \rightarrow a\$}$$
$$\text{\$If\ False\ a\ b \rightarrow (\lambda b\ x\ y .b x y) (\lambda t\ f .f)\ a\ b \rightarrow (\lambda t\ f .f)\ a\ b \rightarrow b\$}$$

Свойства выполнены. Логические константы кодируются постоянными функциями двух аргументов. Функция `True` возвращает первый аргумент, игнорируя второй. А функция `False` делает то же самое, но наоборот. В такой интерпретации логическое отрицание можно закодировать с помощью функции `flip`. Также мы можем выразить и другие логические операции:

$$\text{\$And\$} \quad \text{\$=\$} \quad \text{\$\lambda a\ b .a\ b\ False\$}$$
$$\text{\$Or\$} \quad \text{\$=\$} \quad \text{\$\lambda a\ b .a\ True\ b\$}$$

Мы определили логические значения не конкретными значениями, а свойствами функций. Мы построили функции, которые ведут себя как логические значения. Этот способ определения напоминает, определение класса типов. Мы объявили три метода `$True$`, `$False$` и `If` и сказали, что экземпляр класса должен удовлетворять определённым свойствам, которые накладывают взаимные ограничения на методы класса. Ни один из методов не имеет смысла по отдельности, важно то как они взаимодействуют.

Натуральные числа

Оказывается, что с помощью термов лямбда исчисления можно закодировать и натуральные числа с арифметическими операциями. Мы будем кодировать числа Пеано. Для этого нам понадобится нулевой элемент и функция определения следующего элемента. Их можно закодировать так:

```
$Zero$      $=$   $\lambda sz .z$  
$Succ$      $=$   $\lambda nsz .s(ns z)$
```

Как и в случае логических значений числа кодируются функциями двух аргументов. Число определяется по терму, подсчётом цепочки первых аргументов ss . Например так выглядит число два:

```
$$Succ\ (Succ\ Zero) \rightarrow (\lambda nsz .s(ns z)) (Succ\  
Zero) \rightarrow \lambda sz .s((Succ\ Zero) sz) \rightarrow
```

```
$$\lambda sz .s(((\lambda ns'z' .s'(ns'z'))\ Zero)sz)  
\rightarrow \lambda sz .s((\lambda s'z' .s'(Zero\ s'z'))\ sz)  
\rightarrow
```

```
$$\lambda sz .s((\lambda s'z' .s'z')\ sz) \rightarrow \lambda  
sz .s(sz)$$
```

И мы получили два вхождения первого аргумента в теле функции. Определим сложение и умножение. Сложение принимает две функции двух аргументов и возвращает функцию двух аргументов.

```
$$Add = \lambda m\ n\ s\ z .m\ s\ (n\ s\ z)$$
```

В этой функции мы применяем m раз аргумент ss к значению, в котором аргумент ss применён n раз, так мы и получаем $m+n$ применений аргумента ss . Сложим 3 и 2:

```
$$Add\ 3\ 2 \rightarrow \lambda s\ z .3\ s\ (2\ s\ z)  
\rightarrow \lambda s\ z .3\ s\ (s\ (s\ z)) \rightarrow \lambda s\ z .s\ (  
\ s\ (s\ (s\ z))) \rightarrow 5
```

В умножении чисел m и n мы будем m раз складывать число n :

```
$$Mul = \lambda m\ n\ s\ z .m\ (Add\ n)\ Zero$$
```

Конструктивная математика

В конструктивной математике существование объекта может быть доказано только описанием алгоритма, с помощью которого можно построить объект. Например доказательство методом “от противного” отвергается.

Лямбда исчисление строит конструктивное описание функции. По лямбда-терму мы можем не только вычислять значения функции, но и понять как она была построена. В классической теории, функция это множество пар $(x, f(x))$ аргумент-значение, которое обладает свойством:

$$x = y \implies f(x) = f(y)$$

По этому определению мы ничего не можем сказать о внутренней структуре функции. Мы можем собирать из одних функций другие с помощью подстановки значений, но мы никак не сможем понять, что находится внутри функции. Лямбда исчисление решает эту проблему.

Расширение лямбда исчисления

Предположим, что мы решили написать язык программирования на основе лямбда-исчисления. Было бы очень неэффективно представлять числа с помощью чисел Пеано. Ведь у нас есть процессор и мы можем спросить у него чему равно значение и получить ответ очень быстро.

В этом случае пользуются расширенным лямбда исчислением. В нём два типа примитивов это переменные и константы. Для констант мы можем определять специальные правила редукции. Например мы можем дополнить исчисление константами:

$$+, *, 0, 1, 2, \dots$$

И ввести для них правила редукции, которые запрашивают ответ у процессора:

$$\begin{array}{lll} a+b & \rightarrow & \text{AddWithCPU}(a, b) \\ a*b & \rightarrow & \text{MulWithCPU}(a, b) \end{array}$$

Так же мы можем определить и константы для логических значений:

$\$ \$ \text{True}, \backslash \text{False}, \backslash \text{If}, \backslash \text{Not}, \backslash \text{And}, \backslash \text{Or} \$ \$$

И определить правила редукции:

$\$ \text{If} \backslash \text{True} \backslash a \backslash b \$$	$\$ = \$$	$\$ a \$$
$\$ \text{If} \backslash \text{False} \backslash a \backslash b \$$	$\$ = \$$	$\$ b \$$
$\$ \text{Not} \backslash \text{True} \$$	$\$ = \$$	$\$ \text{False} \$$
$\$ \text{Not} \backslash \text{False} \$$	$\$ = \$$	$\$ \text{True} \$$
$\$ \text{Add} \backslash \text{False} \backslash a \$$	$\$ = \$$	$\$ \text{False} \$$
$\$ \text{Add} \backslash \text{True} \backslash b \$$	$\$ = \$$	$\$ b \$$

...

Такие правила называют $\$ \backslash \text{delta} \$$ -редукцией (дельта-редукция).

Комбинаторная логика

Одновременно с лямбда-исчислением развивалась комбинаторная логика. Она отличается более компактным представлением. Есть всего лишь одно правило, это применение функции к аргументу. А функции строятся не из произвольных термов, а из набора основных функций. Набор основных функций называют *базисом*.

Рассмотрим лямбда-термы:

$\$ \$ \backslash \text{lambda } x . x, \backslash \text{quad } \backslash \text{lambda } y . y, \backslash \text{quad } \backslash \text{lambda } z . z \$ \$$

Все эти термы несут один и тот же смысл. Они представляют тождественную функцию. Они равны, но с точностью до обозначений. Эта навязчивая проблема с переобозначением аргументов была решена в комбинаторной логике. Посмотрим как строятся термы:

- Есть набор переменных $\$ x \$$, $\$ y \$$, $\$ z \$$, Переменная – это терм.
- Есть две константы $\$ K \$$ и $\$ S \$$, они являются термами.
- Если $\$ M \$$ и $\$ N \$$ – термы, то $\$ (MN) \$$ – терм.
- Других термов нет.

Определены правила редукции для базисных термов:

$\$ K x y \$$	$\$ = \$$	$\$ x \$$
$\$ S x y z \$$	$\$ = \$$	$\$ x z (y z) \$$

В этих правилах мы пользуемся соглашением о расстановки скобок. Также как и в лямбда исчислении в применении скобки группируются влево. Когда мы пишем $\$Kxu\$$, мы подразумеваем $\$((Kx)y)\$$. Термы в комбинаторной логике принято называть комбинаторами. Редукция происходит до тех пор пока мы можем заменять вхождения базисных комбинаторов. Так если мы видим связку $\$KXY\$$ или $\$SXYZ\$$, где $\$X\$$, $\$Y\$$, $\$Z\$$ произвольные термы, то мы можем их заменить согласно правилам редукции. Такие связки называют редексами. Если в терме нет ни одного редекса, то он находится в нормальной форме. Замену редекса принято называть *свёрткой*

Интересно, что комбинаторы $\$K\$$ и $\$S\$$ совпадают с определением класса **Applicative** для функций:

```
instance Applicative (r->) where
  pure a r = a
  (<*>) a b r = a r (b r)
```

В этом определении у функций есть общее окружение $\$r\$$, из которого они могут читать значения, так же как и в случае типа **Reader**. В методе `pure` (комбинатор $\$K\$$) мы игнорируем окружение (это константная функция), а в методе `<*>` (комбинатор $\$S\$$) передаём окружение в функцию и аргумент и составляем применение функции в контексте окружения r к значению, которое было получено в контексте того же окружения.

Вернёмся к проблеме различного представления тождественной функции в лямбда-исчислении. В комбинаторной логике тождественная функция выражается так:

$$\$\$I = SKK\$\$$$

Проверим, определяет ли этот комбинатор тождественную функцию:

$$\$\$Ix = SKKx = Kx(Kx) = x\$\$$$

Сначала мы заменили $\$I\$$ на его определение, затем свернули по комбинатору $\$S\$$, затем по левому комбинатору $\$K\$$. В итоге получилось, что

$$\$\$Ix = x\$\$$$

Связь с лямбда-исчислением

Комбинаторная логика и лямбда-исчисление тесно связаны между собой. Можно определить функцию ϕ , которая переводит термы комбинаторной логики в термы лямбда-исчисления:

$$\begin{aligned}\phi(x) &= x \\ \phi(K) &= \lambda xy. x \\ \phi(S) &= \lambda xyz. xz(yz) \\ \phi(XY) &= \phi(X)\phi(Y)\end{aligned}$$

В первом уравнении x – переменная. Также можно определить функцию ψ , которая переводит термы лямбда-исчисления в термы комбинаторной логики.

$$\begin{aligned}\psi(x) &= x \\ \psi(XY) &= \psi(X)\psi(Y) \\ \psi(\lambda x. Y) &= S[x].\psi(Y)\end{aligned}$$

Запись $S[x].T$, где x – переменная, T – терм, обозначает такой терм D , из которого можно получить терм T подстановкой переменной x , выполнено свойство:

$$S([x].T) \setminus x = T$$

Эта запись означает параметризацию термина T по переменной x . Терм $S[x].T$ можно получить с помощью следующего алгоритма:

$$\begin{aligned}S[x].x &= SKK \\ S[\text{left}[x\text{right}].X] &= KX, \quad x \notin V(X) \\ S[\text{left}[x\text{right}].XY] &= S([x].X)([x].Y)\end{aligned}$$

В первом уравнении мы заменяем переменную на тождественную функцию, поскольку переменные совпадают. Запись $V(X)$ во втором уравнении обозначает множество всех переменных в терме X .

Поскольку переменная по которой мы хотим параметризовать терм (или абстрагировать) не участвует в самом терме, мы можем проигнорировать её с помощью постоянной функции K . В последнем уравнении мы параметризуем применение.

С помощью этого алгоритма можно для любого термина T , все переменные которого содержатся в $\{x_1, \dots, x_n\}$ составить такой комбинатор D , что $Dx_1 \dots x_n = T$. Для этого мы последовательно параметризуем терм T по всем переменным:

$$D[x_1, \dots, x_n].T = [x_1].([x_2, \dots, x_n].T)$$

Так постепенно мы придём к выражению, считаем что скобки группируются вправо:

$$D[x_1].[x_2] \dots [x_n].T$$

Немного истории

Комбинаторную логику открыл Моисей Шейнфинкель. В 1920 году на докладе в Гёттингене он рассказал основные положения этой теории. Комбинаторная логика направлена на выделение простейших строительных блоков математической логики. В этом докладе появилось понятие частичного применения. Шейнфинкель показал как функции многих переменных могут быть сведены к функциям одного переменного. Далее в докладе описываются пять основных функций, называемых комбинаторами:

I	$=$	x	– функция тождества
C	$=$	x	– константная функция
T	$=$	xzy	– функция перестановки
Z	$=$	$x(yz)$	– функция группировки
S	$=$	$xz(yz)$	– функция слияния

С помощью этих функций можно избавиться в формулах от переменных, так например свойство коммутативности функции A можно представить так: $TA = A$. Эти комбинаторы зависят друг от друга. Можно убедиться в том, что:

$$\begin{aligned} I &= SCC \\ Z &= S(CS)S \\ T &= S(ZZS)(CC) \end{aligned}$$

Все комбинаторы выражаются через комбинаторы $\$C\$$ и $\$S\$$. Ранее мы пользовались другими обозначениями для этих комбинаторов. Обозначения $\$K\$$ и $\$S\$$ ввёл Хаскель Карри (Haskell Curry). Независимо от Шейнфинкеля он переоткрыл комбинаторную логику и существенно развил её. В современной комбинаторной логике для обозначения комбинаторов $\$I\$$, $\$C\$$, $\$T\$$, $\$Z\$$ и $\$S\$$ (по Шейнфинкелю) принято использовать имена $\$I\$$, $\$K\$$, $\$C\$$, $\$B\$$, $\$S\$$ (по Карри).

Лямбда-исчисление с типами

Мы можем добавить в лямбда-исчисление типы. Предположим, что у нас есть множество $\$V\$$ базовых типов. Тогда тип это:

$$\$T = V \mid ? \mid T \rightarrow T\$$$

Тип может быть либо одним элементом из множества базовых типов. Либо стрелочным (функциональным) типом. Выражение “терм $\$M\$$ имеет тип α ” принято писать так: M^{α} . Стрелочный тип $\alpha \rightarrow \beta$ как и в Haskell говорит о том, что если у нас есть значение типа α , то с помощью операции применения мы можем из терма с этим стрелочным типом получить терм типа β .

Опишем правила построения термов в лямбда-исчислении с типами:

- Переменные x^{α} , y^{β} , z^{γ} , ... являются термами.
- Если $M^{\alpha \rightarrow \beta}$ и N^{α} – термы, то $(M^{\alpha \rightarrow \beta} N^{\alpha})^{\beta}$ – терм.
- Если x^{α} – переменная и M^{β} – терм, то $(\lambda x^{\alpha} . M^{\beta})^{\alpha \rightarrow \beta}$ – терм
- Других термов нет.

Типизация накладывает ограничение на то, какие выражения мы можем комбинировать. В этом есть плюсы и минусы. Теперь наша система является *строго нормализуемой*, это означает, что любой терм имеет нормальную форму. Но теперь мы не можем выразить все функции на

числах. Например мы не можем составить $\$Y\$$ -комбинатор, поскольку теперь самоприменение $\$(e\ e)\$$ невозможно.

Мы ввели типы, но лишились рекурсии. Как нам быть? Эта проблема решается с помощью введения специальной константы $\$Y_{\tau}^{\{(\tau \rightarrow \tau) \rightarrow \tau\}}\$$, которая обозначает комбинатор неподвижной точки. Правило редукции для $\$Y\$$:

$$\$ (Y_{\tau} f^{\{\tau \rightarrow \tau\}})^{\tau} = (f^{\{\tau \rightarrow \tau\}} (Y_{\tau} f^{\{\tau \rightarrow \tau\}}))^{\tau} \$$$

Можно убедиться в том, что это правило проходит проверку типов. Типизированное лямбда-исчисление дополненное комбинатором неподвижной точки способно выразить все числовые функции.

Краткое содержание

В этой главе мы познакомились с лямбда-исчислением и комбинаторной логикой, двумя конструктивными теориями функций. Конструктивными в том смысле, что определение функции содержит не набор значений, а рецепт получения этих значений. В лямбда-исчислении мы видим как функция была построена, из каких простейших частей она состоит. Редукция термов позволяет вычислять функции.

Мы узнали, что функциями можно кодировать логические значения и числа. Узнали, что все численные функции могут быть закодированы лямбда-термами.

Упражнения

- С помощью редукции убедитесь в том, что верны формулы (в терминах Карри) :

$$\$B\$ \quad \$=\$ \quad \$S(KS)S\$$$

$$\$C\$ \quad \$=\$ \quad \$S(BBS)(KK)\$$$

$$\$Bxyz\$ \quad \$=\$ \quad \$xzy\$$$

$$\$Cxyz\$ \quad \$=\$ \quad \$x(yz)\$$$

- Попробуйте закодировать пары с помощью лямбда термов. Вам необходимо построить три функции: `$Pair$`, `Fst`, `Snd`, которые обладают свойствами:

```
$Fst\ (Pair\ a\ b)$    $=$    $a$
$Snd\ (Pair\ a\ b)$    $=$    $b$
```

- в комбинаторной логике тоже есть комбинатор неподвижной точки, найдите его с помощью алгоритма приведения термов лямбда исчисления к термам комбинаторной логики. Для краткости лучше вместо `SKK` писать просто `I`.
- Напишите типы `Lam` и `App`, которые описывают лямбда-термы и термы комбинаторной логики в Haskell. Напишите функции перевода из значений `Lam` в `App` и обратно.

Теория категорий

Многие понятия в Haskell позаимствованы из теории категорий, например это функторы, монады. Теория категорий – это скорее язык, математический жаргон, она настолько общая, что кажется ей нет никакого применения. Возможно это и так, но в этом языке многие сущности, которые лишь казались родственными и было смутное интуитивное ощущение их близости, становятся тождественными.

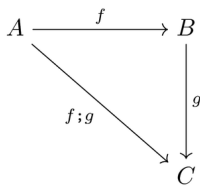
Теория категорий занимается описанием функций. В лямбда-исчислении основной операцией была подстановка значения в функцию, а в теории категорий мы сосредоточимся на операции композиции. Мы будем соединять различные объекты так, чтобы структура объектов сохранялась. Структура объекта будет определяться свойствами, которые продолжают выполняться после преобразования объекта.

Категория

Мы будем говорить об объектах и связях между ними. Связи принято называть “стрелками” или “морфизмами”. Далее мы будем пользоваться термином стрелка. У стрелки есть начальный объект, его называют *доменом* (domain) и конечный объект, его называют *кодоменом* (codomain).

$$A \xrightarrow{f} B$$

В этой записи стрелка f соединяет объекты A и B , в тексте мы будем писать это так $f:A \rightarrow B$, словно стрелка это функция, а объекты это типы. Мы будем обозначать объекты большими буквами A , B , C , ..., а стрелки – маленькими буквами f , g , h , ... Для того чтобы связи было интереснее изучать мы введём такое правило:



Если конец стрелки (f) указывает на начало стрелки (g) , то должна быть такая стрелка $(f \circ g)$, которая обозначает *составную* стрелку. Вводится специальная операция “точка с запятой”, которая называется композицией стрелок: Это правило говорит о том, что связи распространяются по объектам. Теперь у нас есть не просто объекты и стрелки, а целая сеть объектов, связанных между собой. Тот факт, что связи действительно распространяются отражается свойством:

$$(f \circ (g \circ h)) = (f \circ g) \circ h$$

Это свойство называют ассоциативностью. Оно говорит о том, что стрелки, которые образуют составную стрелку являются цепочкой и нам не важен порядок их группировки, важно лишь кто за кем идёт. Подразумевается, что стрелки (f) , (g) и (h) имеют подходящие типы для композиции, что их можно соединять. Это свойство похоже на интуитивное понятие пути, как цепочки отрезков.

Связи между объектами можно трактовать как преобразования объектов. Стрелка $(f : A \rightarrow B)$ – это способ, с помощью которого мы можем перевести объект (A) в объект (B) . Композиция в этой аналогии приобретает естественную интерпретацию. Если у нас есть способ $(f : A \rightarrow B)$ преобразования объекта (A) в объект (B) , и способ $(g : B \rightarrow C)$ преобразования объекта (B) в объект (C) , то мы конечно можем, применив сначала (f) , а затем (g) , получить из объекта (A) объект (C) .

Когда мы думаем о стрелках как о преобразовании, то естественно предположить, что у нас есть преобразование, которое ничего не делает, как тождественная функция. В будем говорить, что для

каждого объекта (A) есть стрелка (id_A) , которая начинается из этого объекта и заканчивается в нём же.

$[id_A : A \rightarrow A]$

Тот факт, что стрелка (id_A) ничего не делает отражается свойствами, которые должны выполняться для всех стрелок:

$$\begin{array}{lll} (id_A \circ f) = f & (f \circ id_B) = f \\ (f \circ id_A) = f & (id_B \circ f) = f \end{array}$$

Если мы добавим к любой стрелке тождественную стрелку, то от этого ничего не изменится.

Всё готово для того чтобы дать формальное определение понятия *категории* (category). Категория это:

- Набор *объектов* (object).
- Набор *стрелок* (arrow) или *морфизмов* (morphism).
- Каждая стрелка соединяет два объекта, но объекты могут совпадать. Так обозначают, что стрелка (f) начинается в объекте (A) и заканчивается в объекте (B) :

$[f : A \rightarrow B]$

При этом стрелка соединяет только два объекта:

$$[f : A \rightarrow B, g : A' \rightarrow B' \iff A=A', B=B']$$

- Определена операция композиции или соединения стрелок. Если конец одной стрелки совпадает с началом другой, то их можно соединить вместе:

$$[f : A \rightarrow B, g : B \rightarrow C \iff f \circ g : A \rightarrow C]$$

- Для каждого объекта есть стрелка, которая начинается и заканчивается в этом объекте. Эту стрелку называют *тождественной* (identity):

$[id_A : A \rightarrow A]$

Должны выполняться аксиомы:

- Тожество $\backslash(id\backslash)$

$$\backslash[id \backslash : \mathbf{\{;\}} \backslash : f = f \backslash]$$

$$\backslash[f \backslash : \mathbf{\{;\}} \backslash : id = f \backslash]$$

- Ассоциативность $\backslash(\backslash : \mathbf{\{;\}} \backslash : \backslash)$

$$\backslash[f \backslash : \mathbf{\{;\}} \backslash : (g \backslash : \mathbf{\{;\}} \backslash : h) = (f \backslash : \mathbf{\{;\}} \backslash : g) \backslash : \mathbf{\{;\}} \backslash : h \backslash]$$

Приведём примеры категорий.

- Одна точка с одной тождественной стрелкой образуют категорию.
- В категории **Set** объектами являются все множества, а стрелками – функции. Стрелки соединяются с помощью композиции функций, тождественная стрелка, это тождественная функция.
- В категории **Hask** объектами являются типы Haskell, а стрелками – функции, стрелки соединяются с помощью композиции функций, тождественная стрелка, это тождественная функция.
- Ориентированный граф может определять категорию. Объекты – это вершины, а стрелки это связанные пути в графе. Соединение стрелок – это соединение путей, а тождественная стрелка, это путь в котором нет ни одного ребра.
- Упорядоченное множество, в котором есть операция сравнения на больше либо равно задаёт категорию. Объекты – это объекты множества. А стрелки это пары объектов таких, что первый объект меньше второго. Первый объект в паре считается начальным, а второй конечным.

$$\backslash[(a, b) : a \rightarrow b \quad \text{если } a \leq b]$$

Стрелки соединяются так:

$$\backslash[(a, b) \backslash : \mathbf{\{;\}} \backslash : (b, c) = (a, c) \backslash]$$

Тождественная стрелка состоит из двух одинаковых объектов:

```
\[id_a = (a, a)\]
```

Можно убедиться в том, что это действительно категория. Для этого необходимо проверить аксиомы ассоциативности и тождества. Важно проверить, что те стрелки, которые получаются в результате композиции, не нарушали бы основного свойства данной структуры, то есть тот факт, что второй элемент пары всегда больше либо равен первого элемента пары.

Отметим, что бывают такие области, в которых стрелки или преобразования с одинаковыми именами могут соединять несколько разных объектов. Например в Haskell есть классы, поэтому функции с одними и теми же именами могут соединять разные объекты. Если все условия категории для объектов и стрелок выполнены, кроме этого, то такую систему называют *прекатегорией* (pre-category). Из любой прекатегории не сложно сделать категорию, если включить имена объектов в имя стрелки. Тогда у каждой стрелки будут только одна пара объектов, которые она соединяет.

Функтор

Вспомним определение класса `Functor`:

```
class Functor f where
  fmap :: (a -> b) -> (f a -> f b)
```

В этом определении участвуют тип `f` и метод `fmap`. Можно сказать, что тип `f` переводит произвольные типы `a` в специальные типы `f a`. В этом смысле тип `f` является функцией, которая определена на типах. Метод `fmap` переводит функции общего типа `a -> b` в специальные функции `f a -> f b`.

При этом должны выполняться свойства:

```
fmap id    = id
fmap (f . g) = fmap f . fmap g
```

Теперь вспомним о категории `Hask`. В этой категории объектами являются типы, а стрелками функции. Функтор `f` отображает объекты

и стрелки категории `Hask` в объекты и стрелки `f Hask`. При этом оказывается, что за счёт свойств функтора `f Hask` образует категорию.

- Объекты – это типы `f a`.
- Стрелки – это функции `fmap f`.
- Композиция стрелок это просто композиция функций.
- Тожественная стрелка это `fmap id`.

Проверим аксиомы:

```
fmap f . fmap id = fmap f . id = fmap f
fmap id . fmap f = id . fmap f = fmap f
```

```
fmap f . (fmap g . fmap h)
= fmap f . fmap (g . h)
= fmap (f . (g . h))
= fmap ((f . g) . h)
= fmap (f . g) . fmap h
= (fmap f . fmap g) . fmap h
```

Видно, что аксиомы выполнены, так функтор `f` порождает категорию `f Hask`. Интересно, что поскольку `Hask` содержит все типы, то она содержит и типы `f Hask`. Получается, что мы построили категорию внутри категории. Это можно пояснить на примере списков. Тип `[]` погружает любой тип в список, а функцию для любого типа можно превратить в функцию, которая работает на списках с помощью метода `fmap`. При этом с помощью класса `Functor` мы проецируем все типы и все функции в мир списков `[a]`. Но сам этот мир списков содержится в категории `Hask`.

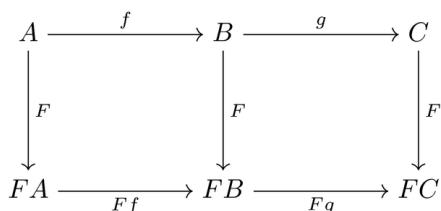
С помощью функторов мы строим внутри одной категории другую категорию, при этом внутренняя категория обладает некоторой структурой. Так если раньше у нас были только произвольные типы `a` и произвольные функции `a -> b`, то теперь все объекты имеют тип `[a]` и все функции имеют тип `[a] -> [b]`. Также и функтор `Maybe` переводит произвольное значение, в значение, которое обладает определённой структурой. В нём выделен дополнительный элемент `Nothing`, который обозначает отсутствие значения. Если по типу `val :: a` мы ничего не можем сказать о содержании значения `val`, то по

типу `val :: Maybe a`, мы знаем один уровень конструкторов. Например мы уже можем проводить сопоставление с образцом.

Теперь давайте вернёмся к теории категорий и дадим формальное определение понятия. Пусть \mathcal{A} и \mathcal{B} – категории, тогда *функтором* из \mathcal{A} в \mathcal{B} называют отображение F , которое переводит объекты \mathcal{A} в объекты \mathcal{B} и стрелки \mathcal{A} в стрелки \mathcal{B} , так что выполнены следующие свойства:

$F f$	$(:)$	$(FA \rightarrow_{\mathcal{B}} FB)$	если $(f: A \rightarrow_{\mathcal{A}} B)$
$F \text{id}_A$	$(=)$	(id_{FA})	для любого объекта A из \mathcal{A}
$F(fg)$	$(=)$	(Ffg)	если $(f: A \rightarrow_{\mathcal{A}} B, g: B \rightarrow_{\mathcal{A}} C)$

Здесь запись \mathcal{A} и \mathcal{B} означает, что эти стрелки в разных категориях. После отображения стрелки f из категории \mathcal{A} мы получаем стрелку в категории \mathcal{B} , это и отражено в типе $(Ff : FA \rightarrow_{\mathcal{B}} FB)$. Первое свойство говорит о том, что после отображения стрелки соединяют те же объекты, что и до отображения. Второе свойства говорит о сохранении тождественных стрелок. А последнее свойство, говорит о том, что “пути” между объектами также сохраняются. Если мы находимся в категории \mathcal{A} в объекте A и перед нами есть путь состоящий из нескольких стрелок в объект B , то неважно как мы пойдём в B либо мы пройдем этот путь в категории \mathcal{A} и в самом конце переместимся в B или мы сначала переместимся в FA и затем пройдем по образу пути в категории \mathcal{B} . Так и так мы попадём в одно и то же место. Схематически это можно изобразить так:



Стрелки сверху находятся в категории \mathcal{A} , а стрелки снизу находятся в категории \mathcal{B} . Функтор $(F : \mathcal{A} \rightarrow \mathcal{A})$, который переводит категорию \mathcal{A} в себя называют *эндофунктором* (endofunctor). Функторы отображают одни категории в другие сохраняя структуру первой категории. Мы словно говорим, что внутри второй категории есть структура подобная первой. Интересно, что последовательное применение функторов, также является функтором. Мы будем писать последовательное применение функторов (F) и (G) слитно, как (FG) . Также можно определить и тождественный функтор, который ничего не делает с категорией, мы будем обозначать его как $(I_{\mathcal{A}})$ или просто (I) , если категория на которой он определён понятна из контекста. Это говорит о том, что мы можем построить категорию, в которой объектами будут другие категории, а стрелками будут функторы.

Естественное преобразование

В программировании часто приходится переводить данные из одной структуры в другую. Каждая из структур хранит какие-то конкретные значения, но мы ничего с ними не делаем мы просто перекладываем содержимое из одного ящика в другой. Например в нашем ящике только один отсек, но вдруг нам пришло бесконечно много подарков, что поделаться нам приходится сохранить первый попавшийся, отбросив остальные. Главное в этой аналогии это то, что мы ничего не меняем, а лишь перекладываем содержимое из одной структуры в другую.

В Haskell это можно описать так:

```
onlyOne :: [a] -> Maybe a
onlyOne []    = Nothing
onlyOne (a:as) = Just a
```

В этой функции мы перекладываем элементы из списка `[a]` в частично определённое значение `Maybe`. Тоже самое происходит и в функции `concat`:

```
concat :: [[a]] -> [a]
```

Элементы переключаются из списка списков в один список. В теории категорий этот процесс называется естественным преобразованием. Структуры определяются функторами. Поэтому в определении будет участвовать два функтора. В функции `onlyOne` это были функторы `[]` и `Maybe`. При переключении элементов мы можем просто выбросить все элементы:

```
burnThemAll :: [a] -> ()
burnThemAll = const ()
```

Можно сказать, что единичный тип также определяет функтор. Это константный функтор, он переводит любой тип в единственное значение `()`, а функцию в `id`:

```
data Empty a = Empty

instance Functor Empty where
    fmap = const id
```

Тогда тип функции `burnThemAll` будет параметризован и слева и справа от стрелки:

```
burnThemAll :: [a] -> Empty a
burnThemAll = const Empty
```

Пусть даны две категории \mathcal{A} и \mathcal{B} и два функтора $(F, G : \mathcal{A} \rightarrow \mathcal{B})$. Преобразованием (transformation) в \mathcal{B} из (F) в (G) называют семейство стрелок (ϵ) :

$\epsilon_A : FA \rightarrow \mathcal{B}GA \quad \text{для любого } A \text{ из } \mathcal{A}$

Рассмотрим преобразование `onlyOne :: [a] -> Maybe a`. Категории \mathcal{A} и \mathcal{B} в данном случае совпадают~ это категория `Hask`. Функтор (F) – это список, а функтор (G) это `Maybe`. Преобразование `onlyOne` для каждого объекта `a` из `Hask` определяет стрелку

```
onlyOne :: [a] -> Maybe a
```

Так мы получаем семейство стрелок, параметризованное объектом из

Hask:

```
onlyOne :: [Int] -> Maybe Int
onlyOne :: [Char] -> Maybe Char
onlyOne :: [Int -> Int] -> Maybe (Int -> Int)
...
...
```

Теперь давайте определим, что значит перекладывать из одной структуры в другую, не меняя содержания. Представим, что функтор – это контейнер. Мы можем менять его содержание с помощью метода `fmap`. Например мы можем прибавить единицу ко всем элементам списка `xs` с помощью выражения `fmap (+1) xs`. Точно так же мы можем прибавить единицу к частично определённым значениям. С точки зрения теории категорий суть понятия “останется неизменным при перекладывании” заключается в том, что если мы возьмём любую функцию к примеру прибавление единицы, то нам неважно когда её применять до функции `onlyOne` или после. И в том и в другом случае мы получим одинаковый ответ. Давайте убедимся в этом:

```
onlyOne $ fmap (+1) [1,2,3,4,5]
=> onlyOne [2,3,4,5,6]
=> Just 2

fmap (+1) $ onlyOne [1,2,3,4,5]
=> fmap (+1) $ Just 1
=> Just 2
```

Результаты сошлись, обратите внимание на то, что функции `fmap (+1)` в двух вариантах являются разными функциями. Первая работает на списках, а вторая на частично определённых значениях. Суть в том, что если при перекладывании значение не изменилось, то нам не важно когда выполнять преобразование внутри функтора `[]` или внутри функтора `Maybe`. Теперь давайте выразим это на языке теории категорий.

Преобразование $\backslash(\backslash\mathrm{varepsilon}\backslash)$ в категории $\backslash(\backslash\mathrm{mathcal}\{B\}\backslash)$ из функтора $\backslash(F\backslash)$ в функтор $\backslash(G\backslash)$ называют *естественным* (natural), если

$$\backslash[Ff \backslash : \backslash\mathrm{mathbf}\{;\}; \backslash : \backslash\mathrm{varepsilon}\backslash_B \backslash = \backslash \backslash\mathrm{varepsilon}\backslash_A \backslash : \backslash\mathrm{mathbf}\{;\}; \backslash : Gf \backslash \quad \backslash\mathrm{text}\{\text{для любого } f : A \rightarrow \backslash\mathrm{mathcal}\{A\}B\}\backslash]$$

Это свойство можно изобразить графически:

$$\begin{array}{ccc}
 FA & \xrightarrow{\varepsilon_A} & GA \\
 Ff \downarrow & & \downarrow Gf \\
 FB & \xrightarrow{\varepsilon_B} & GB
 \end{array}$$

По смыслу ясно, что если у нас есть три структуры данных (или три функтора), и мы просто переложили данные из первой во вторую, а затем переложили данные из второй в третью, ничего не меняя, то итоговое преобразование, которое составлено из последовательного применения переключивания данных, также не меняет данные. Это говорит о том, что композиция двух естественных преобразований также является естественным преобразованием. Также мы можем составить тождественное преобразование: для двух одинаковых функторов $(F : \mathcal{A} \rightarrow \mathcal{B})$, это будет семейство тождественных стрелок в \mathcal{B} . Получается, что для двух категорий \mathcal{A} и \mathcal{B} мы можем составить категорию $\text{Ftr}(\mathcal{A}, \mathcal{B})$, в которой объектами будут функторы из \mathcal{A} в \mathcal{B} , а стрелками будут естественные преобразования. Поскольку естественные преобразования являются стрелками, которые соединяют функторы, мы будем обозначать их как обычные стрелки. Так запись $(\eta : F \rightarrow G)$ обозначает преобразование (η) , которое переводит функтор (F) в функтор (G) .

Интересно, что изначально создатели теории категорий Саунддерс Маклейн и Сэмюэль Эйленберг придумали понятие естественного преобразования, а затем, чтобы дать ему обоснование было придумано понятие функтора, и наконец для того чтобы дать обоснование функторам были придуманы категории. Категории содержат объекты и стрелки, для стрелок есть операция композиции. Также для каждого объекта есть тождественная стрелка. Функторы являются стрелками в категории, в которой объектами являются другие категории. А естественные преобразования являются стрелками в категории, в которой объектами являются функторы. Получается такая иерархия структур.

Монады

Монадой называют эндофунктор $(T: \mathcal{A} \rightarrow \mathcal{A})$, для которого определены два естественных преобразования $(\eta: I \rightarrow T)$ и $(\mu: TT \rightarrow T)$ и выполнены два свойства:

- $(T \eta_A \circ \mu_A) = id_{TA}$
- $(T \mu_A \circ \mu_A) = \mu_{TA}$

Преобразование (η) – это функция `return`, а преобразование (μ) – это функция `join`. В теории категорий в классе `Monad` другие методы. Перепишем эти свойства в виде функций Haskell:

```
join . fmap return = id
join . fmap join   = join . join
```

Порядок следования аргументов изменился, потому что мы пользуемся обычной композицией (через точку). Выражение $(T \eta_A)$ означает применение функтора (T) к стрелке (η_A) . Ведь преобразование это семейство стрелок, которые параметризованы объектами категории. На языке Haskell это означает применить `fmap` к полиморфной функции (функции с параметром).

Также эти свойства можно изобразить графически:

$$\begin{array}{ccc} TTTA & \xrightarrow{T\mu_A} & TTA \\ \mu_{TA} \downarrow & & \downarrow \mu_A \\ TA & \xrightarrow{T\eta_A} TTA \xrightarrow{\mu_A} & TA \\ & & TTA \xrightarrow{\mu_A} TA \end{array}$$

Категория Клейсли

Если у нас есть монада (T) , определённая в категории (\mathcal{A}) , то мы можем построить в этой категории категорию специальных стрелок вида $(A \rightarrow TA)$. Эту категорию называют категорией Клейсли.

- Объекты категории Клейсли \mathcal{A}_T – это объекты исходной категории \mathcal{A} .
- Стрелки в \mathcal{A}_T это стрелки из \mathcal{A} вида $(A \rightarrow TB)$, мы будем обозначать их $(A \rightarrow_T B)$
- Композиция стрелок $(f : A \rightarrow_T B)$ и $(g : B \rightarrow_T C)$ определена с помощью естественных преобразований монады (T) :

$$[f \circ_T g] \circ \mu_B = f \circ \mu_{TB}$$

Значок (\circ_T) указывает на то, что слева от равно композиция в \mathcal{A}_T . Справа от знака равно используется композиция в исходной категории \mathcal{A} .

- Тожественная стрелка – это естественное преобразование η .

Можно показать, что категория Клейсли действительно является категорией и свойства операций композиции и тождества выполнены.

Дуальность

Интересно, что если в категории \mathcal{A} перевернуть все стрелки, то снова получится категория. Попробуйте нарисовать граф со стрелками, и затем мысленно переверните направление всех стрелок. Все пути исходного графа перейдут в перевёрнутые пути нового графа. При этом пути будут проходить через те же точки. Сохранятся композиции стрелок, только все они будут перевёрнуты. Такую категорию обозначают \mathcal{A}^{op} . Но оказывается, что переворачивать мы можем не только категории но и свойства категорий, или утверждения о категориях, эту операцию называют *дуализацией*. Определим её:

$\backslash(\text{dual}\backslash A\backslash)$	$\backslash(\backslash\text{quad} = \backslash\text{quad}\backslash) \backslash(A\backslash)$	$\backslash(\backslash\text{quad}\backslash)$ если $\backslash(A\backslash)$ является объектом
$\backslash(\text{dual}\backslash x\backslash)$	$\backslash(\backslash\text{quad} = \backslash\text{quad}\backslash) \backslash(x\backslash)$	если $\backslash(x\backslash)$ обозначает стрелку
$\backslash(\text{dual}\backslash (f : A \rightarrow B)\backslash)$	$\backslash(\backslash\text{quad} = \backslash(\text{dual}\backslash f : B \rightarrow A)\backslash)$	$\backslash(A\backslash)$ и $\backslash(B\backslash)$ поменялись местами
$\backslash(\text{dual}\backslash (f : \mathbf{f}; \backslash:g)\backslash)$	$\backslash(\backslash\text{quad} = \backslash(\text{dual}\backslash g : \mathbf{f}; \backslash:\text{dual}\backslash f)\backslash)$	$\backslash(f\backslash)$ и $\backslash(g\backslash)$ поменялись местами
$\backslash(\text{dual}\backslash (\text{id}_A)\backslash)$	$\backslash(\backslash\text{quad} = \backslash(\text{id}_A)\backslash)$	

Есть такое свойство: если в исходной категории $\backslash(\mathcal{A}\backslash)$ выполняется какое-то утверждение, то в перевёрнутой категории $\backslash(\mathcal{A}^{\text{op}}\backslash)$ выполняется перевёрнутое (дуальное) свойство. Часто в теории категорий из одних понятий получают другие дуализацией. При этом мы можем не проверять свойства для нового понятия - они будут выполняться автоматически. К дуальным понятиям обычно добавляют приставку “ко”. Приведём пример, получим понятие комонады.

Для начала вспомним определение монады. Монада – это эндифунктор (функтор, у которого совпадают начало и конец или домен и кодомен) $\backslash(T : \mathcal{A} \rightarrow \mathcal{A}\backslash)$ и два естественных преобразования $\backslash(\eta : I \rightarrow T\backslash)$ и $\backslash(\mu : TT \rightarrow T\backslash)$, такие что выполняются свойства:

- $\backslash(T \eta : \mathbf{f}; \backslash:\mu = \text{id}\backslash)$
- $\backslash(T \mu : \mathbf{f}; \backslash:\mu = \mu : \mathbf{f}; \backslash:\mu\backslash)$

Дуализируем это определение. Комонада – это эндифунктор $\backslash(T : \mathcal{A} \rightarrow \mathcal{A}\backslash)$ и два естественных преобразования $\backslash(\eta : T \rightarrow I\backslash)$ и $\backslash(\mu : T \rightarrow TT\backslash)$, такие, что выполняются свойства

- $\backslash(\mu : \mathbf{f}; \backslash:T \eta = \text{id}\backslash)$
- $\backslash(\mu : \mathbf{f}; \backslash:T \mu = \mu : \mathbf{f}; \backslash:\mu\backslash)$

Мы просто переворачиваем домены и кодомены в стрелках и меняем порядок в композиции. Проверьте, сошлись ли типы. Попробуйте нарисовать графическую схему свойств комонады и сравните со схемой для монады.

Можно также определить и категорию коКлейсли. В категории коКлейсли все стрелки имеют вид $\backslash(TA \rightarrow B\backslash)$. Теперь дуализируем композицию из категории Клейсли:

$$\backslash[f \backslash \mathbf{;}_{\{T\}}\backslash:g = f \backslash \mathbf{;}\backslash:Tg \backslash \mathbf{;}\backslash:\mu\backslash]$$

Теперь получим композицию в категории коКлейсли:

$$\backslash[g \backslash \mathbf{;}_{\{T\}}\backslash:f = \mu \backslash \mathbf{;}\backslash:Tg \backslash \mathbf{;}\backslash:f\backslash]$$

Мы перевернули цепочки композиций слева и справа от знака равно. Проверьте, сошлись ли типы. Не забывайте, что в этом определении $\backslash(\eta\backslash)$ и $\backslash(\mu\backslash)$ естественные преобразования для комонады. Нам не нужно проверять, является ли категория коКлейсли действительно категорией. Нам не нужно опять проверять свойства стрелки тождества и ассоциативности композиции, если мы уже проверили их для монады. Следовательно, перевёрнутое утверждение будет выполняться в перевёрнутой категории коКлейсли. В этом основное преимущество определения через дуализацию.

Этим приёмом мы можем воспользоваться и в Haskell, дуализируем класс `Monad`:

```
class Monad m where
    return  :: a -> m a
    (>>=)   :: m a -> (a -> m b) -> m b
```

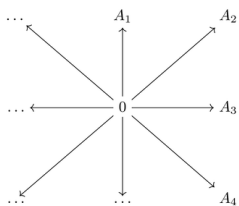
Перевернём все стрелки:

```
class Comonad c where
    coreturn  :: c a -> a
    cobind    :: c b -> (c b -> a) -> c a
```

Начальный и конечный объекты

Начальный объект

Представим, что в нашей категории есть такой объект $\backslash(0\backslash)$, который соединён со всеми объектами, причём стрелка начинается из этого объекта, и для каждого объекта может быть только одна стрелка, которая соединит данный объект с $\backslash(0\backslash)$. Графически эту ситуацию можно изобразить так:



Такой объект называют *начальным* (initial object). Его принято обозначать нулём, словно это начало отсчёта. Для любого объекта A из категории \mathcal{A} с начальным объектом 0 существует и только одна стрелка $f : 0 \rightarrow A$. Можно сказать, что начальный объект определяет функцию, которая переводит объекты A в стрелки $f : 0 \rightarrow A$. Эту функцию обозначают специальными скобками $(\cdot | A)$, она называется *катаморфизмом* (catamorphism).

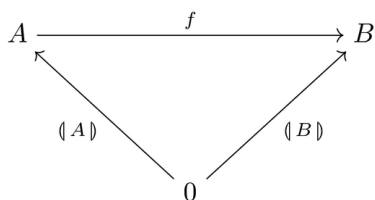
$$(\cdot | A) = f : 0 \rightarrow A$$

У начального объекта есть несколько важных свойств. Они очень часто встречаются в разных вариациях, в понятиях, которые определяются через понятие начального объекта:

$(\cdot 0) = \text{id}_0$	\quad	\quad	\quad
$(f, g : 0 \rightarrow A)$	\rightarrow	$(f = g)$	\quad
$(f : A \rightarrow B)$	\rightarrow	$(\cdot A, \cdot B)$	\quad
		$\mathbf{f} = (\cdot B)$	\quad

Эти свойства следуют из определения начального объекта. Свойство тождества говорит о том, что стрелка, ведущая из начального объекта в начальный, является тождественной стрелкой. В самом деле, по определению начального объекта для каждого объекта может быть только одна стрелка, которая начинается в 0 и заканчивается в этом объекте. Стрелка $(\cdot | 0)$ начинается в 0 и заканчивается в 0 , но у нас уже есть одна такая стрелка - по определению категории для каждого объекта определена тождественная стрелка - значит эта стрелка является единственной.

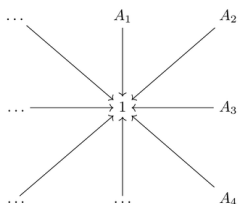
Второе свойство следует из единственности стрелки, ведущей из начального объекта в данный. Третье свойство лучше изобразить графически:



Поскольку стрелки $((\hspace{-1.8pt}|\hspace{-1.8pt}|, A, |\hspace{-1.8pt}|))$ и (f) можно соединить, то должна быть определена стрелка $((\hspace{-1.8pt}|\hspace{-1.8pt}|, A, |\hspace{-1.8pt}|) \vdash \mathbf{f} : 0 \rightarrow B)$, но поскольку в категории с начальным объектом из начального объекта (0) в объект (B) может вести лишь одна стрелка, то стрелка $((\hspace{-1.8pt}|\hspace{-1.8pt}|, A, |\hspace{-1.8pt}|) \vdash \mathbf{f})$ должна совпадать с $((\hspace{-1.8pt}|\hspace{-1.8pt}|, B, |\hspace{-1.8pt}|))$.

Конечный объект

Дуализируем понятие начального объекта. Пусть в категории (\mathcal{A}) есть объект (1) , такой что для любого объекта (A) существует и только одна стрелка, которая начинается из этого объекта и заканчивается в объекте (1) . Такой объект называют *конечным* (terminal object):



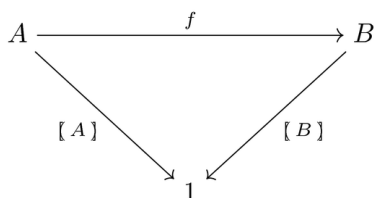
Конечный объект определяет в категории функцию, которая ставит в соответствие объектам стрелки, которые начинаются из данного объекта и заканчиваются в конечном объекте. Такую функцию называют *анаморфизмом* (anamorphism), и обозначают специальными скобками $([\hspace{-2.4pt}(\hspace{-2.4pt}, \cdot, \hspace{-2.4pt})\hspace{-2.4pt}])$, которые похожи на перевёрнутые скобки для катаморфизма:

$\backslash[\hspace{-2.4pt}(\backslash,A\backslash,)\hspace{-2.4pt}] = f : A \rightarrow 1$

Можно дуализировать и свойства:

$\backslash([\hspace{-2.4pt}(\backslash,1\backslash,)\hspace{-2.4pt}] = \text{id}_1)$	$\backslash(\text{quad})$	тождество
$\backslash(f, g : A \rightarrow 1)$	$\backslash(\rightarrow)$	$\backslash(f = g)$ уникальность
$\backslash(f : A \rightarrow B)$	$\backslash(\rightarrow)$	$\backslash(f \backslash : \text{mathbf{f}}; \backslash : [\hspace{-2.4pt}(\backslash,B\backslash,)\hspace{-2.4pt}]) = [\hspace{-2.4pt}(\backslash,A\backslash,)\hspace{-2.4pt}]$ слияние (fusion)

Приведём иллюстрацию для свойства слияния:



Сумма и произведение

Давным-давно, когда мы ещё говорили о типах, мы говорили, что типы конструируются с помощью двух базовых операций: суммы и произведения. Сумма говорит о том, что значение может быть либо одним значением, либо другим. А произведение обозначает сразу несколько значений. В Haskell есть два типа, которые представляют собой сумму и произведение в общем случае. Тип для суммы это **Either**:

```
data Either a b = Left a | Right b
```

Произведение в самом общем виде представлено кортежами:

```
data (a, b) = (a, b)
```

В теории категорий сумма и произведение определяются как начальный и конечный объекты в специальных категориях. Теория категорий изучает объекты по тому, как они взаимодействуют с остальными объектами. Взаимодействие обозначается с помощью стрелок. Специальные свойства стрелок определяют объект.

Например, представим, что мы не можем заглядывать внутрь суммы типов. Как бы мы могли взаимодействовать с объектом, который представляет собой сумму двух типов $(A+B)$? Нам необходимо уметь создавать объект типа $(A+B)$ из объектов A и B и извлекать их из суммы. Создание объектов происходит с помощью двух специальных конструкторов:

$\text{inl} : A \rightarrow A+B$

$\text{inr} : B \rightarrow A+B$

Также нам хочется как-то извлекать значения. По смыслу внутри суммы $(A+B)$ хранится либо объект A , либо объект B , и мы не можем заранее знать какой из них, поскольку внутреннее содержание $(A+B)$ от нас скрыто, но мы знаем, что это только A или B . Это говорит о том, что если у нас есть две стрелки $A \rightarrow C$ и $B \rightarrow C$, то мы как-то можем построить $(A+B) \rightarrow C$. У нас есть операция:

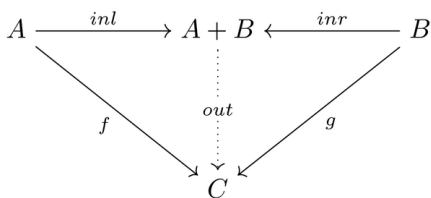
$\text{out}(f,g) : A+B \rightarrow C \quad f : A \rightarrow C, g : B \rightarrow C$

При этом для того, чтобы стрелки inl , inr и out были согласованы необходимо, чтобы выполнялись свойства:

$\text{inl} \circ \text{out}(f,g) = f$

$\text{inr} \circ \text{out}(f,g) = g$

для любых функций f и g . Графически это свойство можно изобразить так:



Итак, суммой двух объектов A и B называется объект $A+B$ и две стрелки $\text{inl} : A \rightarrow A+B$ и $\text{inr} : B \rightarrow A+B$ такие, что для любых двух стрелок $f : A \rightarrow C$ и $g : B \rightarrow C$ существует стрелка $\text{out}(f,g) : A+B \rightarrow C$ такая, что $\text{inl} \circ \text{out}(f,g) = f$ и $\text{inr} \circ \text{out}(f,g) = g$.

$\rightarrow C$) и $(g : B \rightarrow C)$ определена одна и только одна стрелка $(h : A+B \rightarrow C)$ такая, что выполнены свойства:

$$[inl : \mathbf{A} \rightarrow A+B] \quad [inr : \mathbf{B} \rightarrow A+B]$$

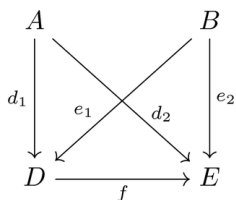
В этом определении объект $(A+B)$ вместе со стрелками (inl) и (inr) , определяет функцию, которая по некоторому объекту (C) и двум стрелкам (f) и (g) строит стрелку (h) , которая ведёт из объекта $(A+B)$ в объект (C) . Этот процесс определения стрелки по объекту напоминает определение начального элемента. Построим специальную категорию, в которой объект $(A+B)$ будет начальным. Тогда функция (out) будет катаморфизмом.

Функция (out) принимает две стрелки и возвращает третью. Посмотрим на типы:

$$[f : A \rightarrow C \quad \quad \quad inl : A \rightarrow A+B]$$

$$[g : B \rightarrow C \quad \quad \quad inr : B \rightarrow A+B]$$

Каждая из пар стрелок в столбцах указывают на один и тот же объект, а начинаются они из двух разных объектов (A) и (B) . Определим категорию, в которой объектами являются пары стрелок $((a_1, a_2))$, которые начинаются из объектов (A) и (B) и заканчиваются в некотором общем объекте (D) . Эту категорию ещё называют клином. Стрелками в этой категории будут такие стрелки $(f : (d_1, d_2) \rightarrow (e_1, e_2))$, что стрелки в следующей диаграмме коммутируют (не важно по какому пути идти из двух разных точек).

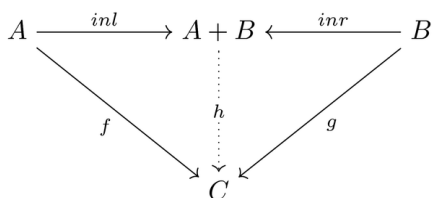


Композиция стрелок – это обычная композиция в исходной категории, в которой определены объекты (A) и (B) , а тождественная стрелка для каждого объекта – это тождественная стрелка для того

объекта, в котором сходятся обе стрелки. Можно проверить, что это действительно категория.

Если в этой категории есть начальный объект, то мы будем называть его суммой объектов (A) и (B) . Две стрелки, которые содержит этот объект мы будем называть (inl) и (inr) , а общий объект, в котором эти стрелки сходятся, будем называть $(A+B)$. Теперь если мы выпишем определение для начального объекта, но вместо произвольных стрелок и объектов подставим наш конкретный случай, то мы получим как раз исходное определение суммы.

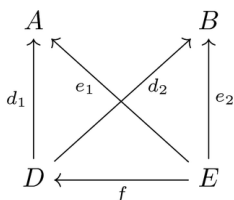
Начальный объект $((inl : A \rightarrow A+B, inr : B \rightarrow A+B))$ ставит в соответствие любому объекту $((f : A \rightarrow C, g : B \rightarrow C))$ стрелку $(h : A+B \rightarrow C)$ такую, что выполняются свойства:



А как на счёт произведения? Оказывается, что произведение является дуальным понятием по отношению к сумме. Его иногда называют косуммой, или сумму называют копроизведением.

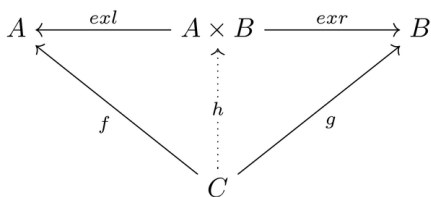
Дуализируем категорию, которую мы строили для суммы.

У нас есть категория (\mathcal{A}) и в ней выделено два объекта (A) и (B) . Объектами новой категории будут пары стрелок $((a_1, a_2))$, которые *начинаются* в общем объекте (C) , а заканчиваются в объектах (A) и (B) . Стрелками в этой категории будут стрелки исходной категории $(h : (e_1, e_2) \rightarrow (d_1, d_2))$ такие, что следующая диаграмма коммутрует:



Композиция и тождественные стрелки позаимствованы из исходной категории \mathcal{A} . Если в этой категории существует *конечный* объект, то мы будем называть его произведением объектов A и B . Две стрелки этого объекта обозначаются как (exl, exr) , а общий объект из которого они начинаются мы назовём $A \times B$. Теперь распишем определение конечного объекта для нашей категории пар стрелок с общим началом.

Конечный объект $((exl : A \times B \rightarrow A, \text{ } exr : A \times B \rightarrow B))$ ставит в соответствие любому объекту категории $((f : C \rightarrow A, \text{ } g : C \rightarrow B))$ стрелку $(h : C \rightarrow A \times B)$. При этом выполняются свойства:



Итак, мы определили сумму, а затем на автомате, перевернув все утверждения, получили определение произведения. Но что это такое? Соответствует ли оно интуитивному понятию произведения?

Так же как и в случае суммы в теории категорий мы определяем понятие через то, как мы можем с ним взаимодействовать. Посмотрим, что нам досталось от абстрактного определения. У нас есть обозначение произведения типов $(A \times B)$. Две стрелки (exl) и (exr) . Также у нас есть способ получить по двум функциям $(f : C \rightarrow A)$ и $(g : C \rightarrow B)$ стрелку $(h : C \rightarrow A \times B)$. Для начала посмотрим на типы стрелок конечного объекта:

$[exl : A \times B \rightarrow A]$

$[exr : A \times B \rightarrow B]$

По типам видно, что эти стрелки разбивают пару на составляющие. По смыслу произведения мы точно знаем, что у нас есть в $(A \times B)$ и объект A и объект B . Эти стрелки позволяют нам извлекать компоненты пары. Теперь посмотрим на анаморфизм:

$$\backslash[[\hspace{-2.4pt}(\backslash,f,g\backslash,)\hspace{-2.4pt}]] : C \rightarrowtail A \times B \quad f : C \rightarrowtail A, \backslash g : C \rightarrowtail B \backslash]$$

Эта функция позволяет строить пару по двум функциям и начальному значению. Но, поскольку здесь мы ничего не вычисляем, а лишь связываем объекты, мы можем по паре стрелок, которые начинаются из общего источника связать источник с парой конечных точек $\backslash (A \times B \backslash)$.

При этом выполняются свойства:

$$\backslash[[\hspace{-2.4pt}(\backslash,f,g\backslash,)\hspace{-2.4pt}]] \backslash:\mathbf{f};\backslash:\mathrm{exl} = f \backslash]$$

$$\backslash[[\hspace{-2.4pt}(\backslash,f,g\backslash,)\hspace{-2.4pt}]] \backslash:\mathbf{f};\backslash:\mathrm{exr} = g \backslash]$$

Эти свойства говорят о том, что функции построения пары и извлечения элементов из пары согласованы. Если мы положим значение в первый элемент пары и тут же извлечём его, то это тоже само если бы мы не использовали пару совсем. То же самое и со вторым элементом.

Экспонента

Если представить, что стрелки это функции, то может показаться, что все наши функции являются функциями одного аргумента. Ведь у стрелки есть только один источник. Как быть если мы хотим определить функцию нескольких аргументов, что она связывает? Если в нашей категории определено произведение объектов, то мы можем представить функцию двух аргументов, как стрелку, которая начинается из произведения:

$$\backslash[(+) : \mathrm{Num} \times \mathrm{Num} \rightarrowtail \mathrm{Num} \backslash]$$

Но в лямбда-исчислении нам были доступны более гибкие функции, функции могли принимать на вход функции и возвращать функции. Как с этим обстоят дела в теории категорий? Если перевести определение функций высшего порядка на язык теории категорий, то мы получим стрелки, которые могут связывать другие стрелки. Категория с функциями высшего порядка может содержать свои

стрелки в качестве объектов. Стрелки как объекты обозначаются с помощью степени, так запись (B^A) означает стрелку $(A \rightarrow B)$. При этом нам необходимо уметь интерпретировать стрелку, мы хотим уметь подставлять значения. Если у нас есть объект (B^A) , то должна быть стрелка

$[\text{eval} : B^A \times A \rightarrow B]$

На языке функций можно сказать, что стрелка (eval) принимает функцию высшего порядка $(A \rightarrow B)$ и значение типа (A) , а возвращает значение типа (B) . Объект (B^A) называют экспонентой. Теперь дадим формальное определение.

Пусть в категории (\mathcal{A}) определено произведение.

Экспонента – это объект (B^A) вместе со стрелкой $(\text{eval} : B^A \times A \rightarrow B)$ такой, что для любой стрелки $(f : C \times A \rightarrow B)$ определена стрелка $(\text{curry}(f) : C \rightarrow B^A)$ при этом следующая диаграмма коммутует:

$$\begin{array}{ccc} C & & C \times A \\ \text{curry}(f) \downarrow & & \downarrow (\text{curry}(f), \text{id}) \quad \searrow f \\ B^A & & B^A \times A \longrightarrow B \end{array}$$

Давайте разберёмся, что это всё означает. По смыслу стрелка $(\text{curry}(f))$ это каррированная функция двух аргументов. Вспомните о функции `curry` из Haskell. Диаграмма говорит о том, что если мы каррированием функции двух аргументов получим функцию высшего порядка $(C \rightarrow B^A)$, а затем с помощью функции (eval) получим значение, то это всё равно, что подставить два значения в исходную функцию. Запись $((\text{curry}(f), \text{id}))$ означает параллельное применение двух стрелок внутри пары:

$[(f, g) : A \times A' \rightarrow B \times B' , \quad \text{quad } f : A \rightarrow B , \quad g : A' \rightarrow B']$

Так применив стрелки $(\text{curry}(f) : C \rightarrow B^A)$ и $(\text{id} : A \rightarrow A)$ к паре $(C \times A)$, мы получим пару $(B^A \times A)$. Применение здесь условное мы подразумеваем применение

в функциональной аналогии, в теории категорий происходит связывание пар объектов с помощью стрелки $((f, g))$.

Интересно, что и экспоненту можно получить как конечный объект в специальной категории. Пусть есть категория (\mathcal{A}) и в ней определено произведение объектов (A) и (B) . Построим категорию, в которой объектами являются стрелки вида:

$$[C \times A \rightarrow B]$$

где (C) – это произвольный объект исходной категории. Стрелкой между объектами $(c : C \times A \rightarrow B)$ и $(d : D \times A \rightarrow B)$ в этой категории будет стрелка $(f : C \rightarrow D)$ из исходной категории, такая, что следующая диаграмма коммутует:

$$\begin{array}{ccc} C & & C \times A \\ \downarrow f & & \downarrow (f, id) \\ D & & D \times A \end{array} \quad \begin{array}{c} \nearrow c \\ \longrightarrow \end{array} \quad \begin{array}{c} B \\ \end{array}$$

Если в этой категории существует конечный объект, то он является экспонентой. А функция (curry) является анаморфизмом для экспоненты.

Краткое содержание

Теория категорий изучает понятия через то, как эти понятия взаимодействуют друг с другом. Мы забываем о том, как эти понятия реализованы, а смотрим лишь на свойства связей.

Мы узнали, что такое категория. Категория это структура с объектами и стрелками. Стрелки связывают объекты. Причём связи могут соединяться. Также считается, что объект всегда связан сам с собой. Мы узнали, что есть такие категории, в которых сами объекты являются категориями, а стрелки в таких категориях мы называли функторами. Также мы узнали, что сами функторы могут стать объектами в некоторой категории, тогда стрелки в этой категории мы называли естественными преобразованиями.

Мы узнали, что такое начальный и конечный объект и как с помощью этих понятий можно определить сумму и произведение типов. Также мы узнали как в теории категорий описываются функции высших порядков.

Упражнения

- Проверьте аксиомы категории (ассоциативность и тождество) для категории функторов и категории естественных преобразований.
- Изоморфизмом называют такие стрелки $(f:A \rightarrow B)$ и $(g:B \rightarrow A)$, для которых выполнено свойство:
 $[f \circ g = \text{id}_A] \quad [g \circ f = \text{id}_B]$
Объекты (A) и (B) называют изоморфными, если они связаны изоморфизмом, это обозначают так: $(A \cong B)$.
Докажите, что все начальные и конечные элементы изоморфны.
- Поскольку сумма и произведение типов являются начальным и конечным объектами в специальных категориях для них также выполняются свойства тождества, уникальности и слияния. Выпишите эти свойства для суммы и произведения.
- Подумайте как можно определить экземпляр класса `Comonad` для потоков:

```
data Stream a = a :& Stream a
```

Можно ли придумать экземпляр для класса `Monad`?

- Дуальную категорию для категории (\mathcal{A}) обозначают (\mathcal{A}^{op}) . Если (F) является функтором в категории (\mathcal{A}^{op}) , то в исходной категории его называют *контравариантным* функтором. Выпишите определение функтора в (\mathcal{A}^{op}) , а затем с помощью дуализации получите свойства контравариантного функтора в исходной категории (\mathcal{A}) .

Категориальные типы

В этой главе мы узнаем как в теории категорий определяются типы. В теории категорий типы определяются как начальные и конечные объекты в специальных категориях, которые называются алгебрами функторов. Для понимания этой главы хорошо освежить в памяти главу о структурной рекурсии, там где мы говорили о свёртках и развёртках.

Программирование в стиле оригами

Оригами – состоит из двух слов “свёртка” и “бумага”. При программировании в стиле оригами все функции строятся через функции свёртки и развёртки. Есть даже такие языки программирования, в которых это единственный способ определения рекурсии. Этот стиль очень хорошо подходит для ленивых языков программирования, поскольку в связке:

```
fold f . unfold g
```

функции свёртки и развёртки работают синхронно. Функция развёртки не производит новых элементов до тех пор пока они не понадобятся во внешней функции свёртки.

Помните в одной из глав мы говорили о том, что рекурсивные функции можно определять через функцию `fix`.

Например так выглядит рекурсивная функция сложения всех чисел от одного до `n`:

```
sumInt :: Int -> Int
sumInt 0 = 0
sumInt n = n + sumInt (n-1)
```

Эту функцию мы можем переписать с помощью функции `fix`. При вычислении `fix f` будет составлено значение

```
f (f (f (f ...)))
```

Теперь перепишем функцию `sumInt` через `fix`:

```
sumInt = fix $ \f n ->
  case n of
    0   -> 0
    n   -> n + f (n - 1)
```

Смотрите лямбда функция в аргументе `fix` принимает функцию и число, а возвращает число. Тип этой функции `(Int -> Int) -> (Int -> Int)`. После применения функции `fix` мы как раз и получим функцию типа `Int -> Int`. В лямбда функции рекурсивный вызов был заменён на вызов функции-параметра `f`.

Оказывается, что этот приём может быть применён и для рекурсивных типов данных. Мы можем создать обобщённый тип, который обозначает рекурсивный тип:

```
newtype Fix f = Fix { unFix :: f (Fix f) }
```

В этой записи мы получаем уравнение неподвижной точки `Fix f = f (Fix f)`, где `f` это некоторый тип с параметром. Определим тип целых чисел:

```
data N a = Zero | Succ a
  deriving (Show, Eq)

type Nat = Fix N
```

Теперь создадим несколько конструкторов:

```
zero :: Nat
zero = Fix Zero

succ :: Nat -> Nat
succ = Fix . Succ
```

Сохраним эти определения в модуле `Fix.hs` и посмотрим в интерпретаторе на значения и их типы, `ghc` не сможет вывести экземпляр `Show` для типа `Fix`, потому что он зависит от типа с параметром, а не от конкретного типа. Для решения этой проблемы нам придётся определить экземпляры вручную и подключить несколько расширений языка. Помните в главе о ленивых вычислениях мы подключали расширение `BangPatterns`? Нам понадобятся:

```
{-# Language FlexibleContexts, UndecidableInstances #-}
```

Теперь определим экземпляры для `Show` и `Eq`:

```
instance Show (f (Fix f)) => Show (Fix f) where
  show x = "(" ++ show (unFix x) ++ ")"

instance Eq (f (Fix f)) => Eq (Fix f) where
  a == b = unFix a == unFix b
```

Определим списки-оригами:

```
data L a b = Nil | Cons a b
  deriving (Show, Eq)
```

```
type List a = Fix (L a)
```

```
nil :: List a
nil = Fix Nil
```

```
infixr 5 `cons`
```

```
cons :: a -> List a -> List a
cons a = Fix . Cons a
```

В типе `L` мы заменили рекурсивный тип на параметр. Затем в записи `List a = Fix (L a)` мы производим замыкание по параметру. Мы бесконечно вкладываем тип `L a` во второй параметр. Так получается рекурсивный тип для списков. Составим какой-нибудь список:

```
*Fix> :r
[1 of 1] Compiling Fix                ( Fix.hs, interpreted )
Ok, modules loaded: Fix.
*Fix> 1 `cons` 2 `cons` 3 `cons` nil
(Cons 1 (Cons 2 (Cons 3 (Nil))))
```

Спрашивается, зачем нам это нужно? Зачем нам записывать рекурсивные типы через тип `Fix`? Оказывается, при такой записи мы можем построить универсальные функции `fold` и `unfold` - они будут работать для любого рекурсивного типа.

Помните, как мы составляли функции свёртки? Мы строили воображаемый класс, в котором сворачиваемый тип заменялся на параметр. Например, для списка мы строили свёртку так:

```
class [a] b where
  (:) :: a -> b -> b
  []  :: b
```

После этого мы легко получали тип для функции свёртки:

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

Она принимает методы воображаемого класса, в котором тип записан с параметром, а возвращает функцию из рекурсивного типа в тип параметра.

Сейчас мы выполняем эту процедуру замены рекурсивного типа на параметр в обратном порядке. Сначала мы строим типы с параметром, а затем получаем из них рекурсивные типы с помощью конструкции **Fix**. Теперь методы класса с параметром это наши конструкторы исходных типов, а рекурсивный тип записан через **Fix**. Если мы сопоставим два способа, то мы сможем получить такой тип для функции свёртки:

```
fold :: (f b -> b) -> (Fix f -> b)
```

Смотрите, функция свёртки по-прежнему принимает методы воображаемого класса с параметром, но теперь класс перестал быть воображаемым, он стал типом с параметром. Результатом функции свёртки будет функция из рекурсивного типа **Fix f** в тип параметр.

Аналогично строится и функция `unfold`:

```
unfold :: (b -> f b) -> (b -> Fix f)
```

В первой функции мы указываем один шаг разворачивания рекурсивного типа, а функция развёртки рекурсивно распространяет этот один шаг на потенциально бесконечную последовательность применений этого одного шага.

Теперь давайте определим эти функции. Но для этого нам понадобится от типа `f` одно свойство - он должен быть функтором. Опираясь на это свойство, мы будем рекурсивно обходить этот тип.

```
fold :: Functor f => (f a -> a) -> (Fix f -> a)
fold f = f . fmap (fold f) . unFix
```

Проверим эту функцию по типам. Для этого нарисуем схему композиции:

$$\text{Fix } f \xrightarrow{f} f \text{ (Fix } f) \xrightarrow{\text{fmap (fold } f)} f \text{ a} \xrightarrow{f} a$$

Сначала мы разворачиваем обёртку **Fix** и получаем значение типа `f` (**Fix f**), затем с помощью `fmap` мы внутри типа `f` рекурсивно

вызываем функцию свёртки и в итоге получаем значение `f a`, на последнем шаге мы выполняем свёртку на текущем уровне вызовом функции `f`.

Аналогично определяется и функция `unfold`. Только теперь мы сначала развернём первый уровень, затем рекурсивно вызовем развёртку внутри типа `f` и только в самом конце завернём всё в тип `Fix`:

```
unfold :: Functor f => (a -> f a) -> (a -> Fix f)
unfold f = Fix . fmap (unfold f) . f
```

Схема композиции:

$$\text{Fix } f \xleftarrow{\text{Fix}} f \text{ (Fix } f) \xleftarrow{\text{fmap (unfold } f)} f \text{ a} \xleftarrow{f} a$$

Возможно, вы уже догадались о том, что функция `fold` дуальна по отношению к функции `unfold`. Это особенно наглядно отражается на схеме композиции: при переходе от `fold` к `unfold` мы просто перевернули все стрелки и заменили разворачивание типа `Fix` на заворачивание в `Fix`.

Определим несколько функций для натуральных чисел и списков в стиле оригами. Для начала сделаем `L` и `N` экземпляром класса `Functor`:

```
instance Functor N where
  fmap f x = case x of
    Zero    -> Zero
    Succ a  -> Succ (f a)

instance Functor (L a) where
  fmap f x = case x of
    Nil      -> Nil
    Cons a b -> Cons a (f b)
```

Это всё что нам нужно для того чтобы начать пользоваться функциями свёртки и развёртки! Определим экземпляр `Num` для натуральных чисел:

```
instance Num Nat where
  (+) a = fold $ \x -> case x of
    Zero    -> a
    Succ x  -> succ x
```

```

(*) a = fold $ \x -> case x of
    Zero    -> zero
    Succ x  -> a + x

fromInteger = unfold $ \n -> case n of
    0    -> Zero
    n    -> Succ (n-1)

abs = undefined
signum = undefined

```

Сложение и умножение определены через свёртку, а функция построения натурального числа из числа типа `Integer` определена через развёртку. Сравните с теми функциями, которые мы писали в главе про структурную рекурсию. Теперь мы не передаём отдельно две функции, на которые мы будем заменять конструкторы. Эти функции закодированы в типе с параметром. Для того, чтобы этот код заработал, нам придётся добавить ещё два расширения `TypeSynonymInstances` `FlexibleInstances` - наши рекурсивные типы являются синонимами, а не новыми типами. В рамках стандарта Haskell мы можем определять экземпляры только для новых типов и для того, чтобы обойти это ограничение, мы добавляем ещё два расширения.

```

*Fix> succ $ 1+2
(Succ (Succ (Succ (Succ (Zero)))))
*Fix> ((2 * 3) + 1) :: Nat
(Succ (Succ (Succ (Succ (Succ (Succ (Zero)))))))
*Fix> 2+2 == 2*(2::Nat)
True

```

Определим функции на списках. Для начала зададим две вспомогательные функции, которые извлекают голову и хвост списка:

```

headL :: List a -> a
headL x = case unFix x of
    Nil      -> error "empty list"
    Cons a _ -> a

tailL :: List a -> List a
tailL x = case unFix x of
    Nil      -> error "empty list"
    Cons a b -> b

```

Теперь определим несколько новых функций:

```

mapL :: (a -> b) -> List a -> List b
mapL f = fold $ \x -> case x of
    Nil      -> nil
    Cons a b  -> f a `cons` b

takeL :: Int -> List a -> List a
takeL = curry $ unfold $ \(n, xs) ->
    if n == 0 then Nil
    else Cons (headL xs) (n-1, tailL xs)

appendL :: List a -> List a -> List a
appendL a b = fold (\x -> case x of
    Nil -> b
    Cons x' y' -> x' `cons` y') a

iterateL :: (a -> a) -> a -> List a
iterateL f = unfold $ \a -> Cons a $ f a

```

Сравните эти функции с теми, что мы определяли в главе о структурной рекурсии. Проверим, работают ли эти функции:

```

*Fix> :r
[1 of 1] Compiling Fix                ( Fix.hs, interpreted )
Ok, modules loaded: Fix.
*Fix> takeL 3 $ iterateL (+1) zero
(Cons (Zero) (Cons (Succ (Zero)) (Cons (Succ (Succ (Zero))) (Nil))))
*Fix> let x = 1 `cons` 2 `cons` 3 `cons` nil
*Fix> mapL (+10) $ x `appendL` x
(Cons 11 (Cons 12 (Cons 13 (Cons 11 (Cons 12 (Cons 13 (Nil)))))))

```

Обратите внимание на то, что с большими буквами мы пишем `Cons` и `Nil`, когда хотим закодировать функции для свёртки-развёртки, а с маленькой буквы пишем значения рекурсивного типа. Надеюсь, что вы разобрались на примерах как устроены функции `fold` и `unfold`, потому что теперь мы перейдём к теории, которая за этим стоит.

ИНДУКТИВНЫЕ И КОИНДУКТИВНЫЕ ТИПЫ

С точки зрения теории категорий функция свёртки является катаморфизмом, а функция развёртки – анаморфизмом. Напомню, что катаморфизм – это функция, которая ставит в соответствие объектам категории с начальным объектом стрелки, которые начинаются из начального объекта, а заканчиваются в данном объекте. Анаморфизм – это перевёрнутый наизнанку катаморфизм.

Начальным и конечным объектом будет рекурсивный тип. Вспомним тип свёртки:

```
fold :: Functor f => (f a -> a) -> (Fix f -> a)
```

Функция свёртки строит функции, которые ведут из рекурсивного типа в произвольный тип, поэтому в данном случае рекурсивный тип будет начальным объектом. Функция развёртки строит из произвольного типа данный рекурсивный тип: на языке теории категорий она строит стрелку из произвольного объекта в рекурсивный, а это означает, что рекурсивный тип будет конечным объектом.

```
unfold :: Functor f => (a -> f a) -> (a -> Fix f)
```

Категории, которые определяют рекурсивные типы таким образом, называются (ко)алгебрами функторов. Видите, в типе и той, и другой функции стоит требование о том, что f является функтором. Катаморфизм и анаморфизм отображают объекты в стрелки. По типу функций `fold` и `unfold` мы можем сделать вывод, что объектами в нашей категории для свёрток будут стрелки вида

$f\ a \rightarrow a$

или для развёрток:

$a \rightarrow f\ a$

А стрелками будут обычные функции одного аргумента. Теперь дадим более формальное определение.

Эндифунктор $(F : \mathcal{A} \rightarrow \mathcal{A})$ определяет стрелки $(\alpha : FA \rightarrow A)$, которые называются (F) -алгебрами. Стрелку $(h : A \rightarrow B)$ называют (F) -гомоморфизмом, если следующая диаграмма коммутрует:

$$\begin{array}{ccc} FA & \xrightarrow{\alpha} & A \\ Fh \downarrow & & \downarrow h \\ FB & \xrightarrow{\beta} & B \end{array}$$

Или, можно сказать по-другому, для (F) -алгебр $(\alpha : FA \rightarrow A)$ и $(\beta : FB \rightarrow B)$ выполняется:

$$Fh \circ \beta = \alpha \circ h$$

Это свойство совпадает со свойством естественного преобразования – только вместо одного из функторов мы подставили тождественный функтор (I) . Определим категорию $\text{Alg}(F)$ для категории \mathcal{A} и эндифунктора $(F : \mathcal{A} \rightarrow \mathcal{A})$

- Объектами являются (F) -алгебры $(A \rightarrow A)$, где (A) – объект категории \mathcal{A}
- Два объекта $(\alpha : FA \rightarrow A)$ и $(\beta : FB \rightarrow B)$ соединяет (F) -гомоморфизм $(h : A \rightarrow B)$. Это такая стрелка из \mathcal{A} , для которой выполняется:

$$Fh \circ \beta = \alpha \circ h$$

- Композиция и тождественная стрелка взяты из категории \mathcal{A} .

Если в этой категории есть начальный объект $(in_F : FT \rightarrow T)$, то определён катаморфизм, который переводит объекты $(A \rightarrow A)$ в стрелки $(T \rightarrow A)$. Причём следующая диаграмма коммутрует:

$$\begin{array}{ccc} FT & \xrightarrow{in_F} & T \\ F(\alpha) \downarrow & & \downarrow (\alpha) \\ FA & \xrightarrow{\alpha} & A \end{array}$$

Этот катаморфизм и будет функцией свёртки для рекурсивного типа (T) . Понятие $\text{Alg}(F)$ можно перевернуть и получить категорию $\text{CoAlg}(F)$.

- Объектами являются (F) -коалгебры $(A \rightarrow FA)$, где (A) – объект категории \mathcal{A}

- Два объекта $(\alpha : A \rightarrow FA)$ и $(\beta : B \rightarrow FB)$ соединяет (F) -когомоморфизм. Это такая стрелка из (\mathcal{A}) , для которой выполняется:

$$[h : \alpha = \beta : Fh]$$

- Композиция и тождественная стрелка взяты из категории (\mathcal{A}) .

Если в этой категории есть конечный объект, то его называют $(out_F : T \rightarrow FT)$. Тогда определён анаморфизм, который переводит объекты $(A \rightarrow FA)$ в стрелки $(A \rightarrow T)$.

Причём следующая диаграмма коммутует:

$$\begin{array}{ccc} T & \xrightarrow{in_F} & FT \\ [\alpha] \downarrow & & \downarrow F[\alpha] \\ A & \xrightarrow{\alpha} & FA \end{array}$$

Если для категории (\mathcal{A}) и функтора (F) определены стрелки (in_F) и (out_F) , то они являются взаимнообратными и определяют изоморфизм $(T \cong FT)$. Часто объект (T) в случае $(\mathbf{Alg}(F))$ обозначают (μ_F) , поскольку начальный объект определяется функтором (F) , а в случае $(\mathbf{CoAlg}(F))$ обозначают (ν_F) .

Типы, которые являются начальными объектами, принято называть индуктивными, а типы, которые являются конечными объектами – коиндуктивными.

Существование начальных и конечных объектов

Мы говорили, что если начальный(конечный) объект существует, а когда он существует? Рассмотрим один важный случай. Если категория является категорией, в которой объектами являются полные частично упорядоченные множества, а стрелками являются монотонные функции, такие категории называют (\mathbf{CPO}) , и функтор – полиномиальный, то начальный и конечный объекты существуют.

Полные частично упорядоченные множества

Оказывается на значениях можно ввести частичный порядок. Порядок называется частичным, если отношение \leq определено не для всех элементов, а лишь для некоторых из них. Частичный порядок на значениях отражает степень неопределённости значения. Самый маленький объект это полностью неопределённое значение \bot . Любое значение типа содержит больше определённости чем \bot .

Для того чтобы не путать упорядочивание значений по степени определённости с обычным числовым порядком, пользуются специальным символом \sqsubseteq . Запись

$a \sqsubseteq b$

означает, что b более определено (или информативнее) чем a .

Так для логических значений определены два нетривиальных сравнения:

$\text{data Bool} = \text{True} \mid \text{False}$

$\bot \sqsubseteq \text{True}$ $\bot \sqsubseteq \text{False}$

Мы будем называть нетривиальными сравнения в которых, компоненты слева и справа от \sqsubseteq не равны. Например ясно, что $\text{True} \sqsubseteq \text{True}$ или $\bot \sqsubseteq \bot$. Это тривиальные сравнения и мы их будем лишь подразумевать.

Считается, что если два значения определены полностью, то мы не можем сказать какое из них информативнее. Так к примеру для логических значений мы не можем сказать какое значение более определено True или False .

Рассмотрим пример по-сложнее. Частично определённые значения:

$\text{data Maybe a} = \text{Nothing} \mid \text{Just a}$

\bot	\sqsubseteq	Nothing
\bot	\sqsubseteq	$\text{Just } \bot$
\bot	\sqsubseteq	Just a
Just a	\sqsubseteq	Just b , если $a \sqsubseteq b$

Если вспомнить как происходит вычисление значения, то значение $\backslash(a\backslash)$ менее определено чем $\backslash(b\backslash)$, если взрывное значение $\backslash(\backslash bot\backslash)$ в $\backslash(a\backslash)$ находится ближе к корню значения, чем в $\backslash(b\backslash)$. Итак получается, что в категории $\backslash(\textbf{Hask})\backslash$ объекты это множества с частичным порядком. Что означает требование монотонности функции?

Монотонность в контексте операции $\backslash(\sqsubseteq)\backslash$ говорит о том, что чем больше определён вход функции тем больше определён выход:

$$\backslash[a \sqsubseteq b \quad \Rightarrow \quad f\ a \sqsubseteq f\ b\]$$

Это требование накладывает запрет на возможность проведения сопоставления с образцом по значению $\backslash(\backslash bot\backslash)$. Иначе мы можем определять немонотонные функции вроде:

```
isBot :: Bool -> Bool
isBot undefined = True
isBot _         = undefined
```

Полнота частично упорядоченного множества означает, что у любой последовательности $\backslash(x_n\backslash)$

$$\backslash[x_0 \sqsubseteq x_1 \sqsubseteq x_2 \sqsubseteq \dots\]$$

есть значение $\backslash(x\backslash)$, к которому она сходится. Это значение называют супремумом множества. Что такое полные частично упорядоченные множества мы разобрались. А что такое полиномиальный функтор?

Полиномиальный функтор

Полиномиальный функтор – это функтор который построен лишь с помощью операций суммы, произведения, постоянных функторов, тождественного фуктора и композиции функторов. Определим эти операции:

- Сумма функторов $\backslash(F\backslash)$ и $\backslash(G\backslash)$ определяется через операцию суммы объектов:

$$\backslash[(F+G)X = FX + GX\]$$

- Произведение функторов $\backslash(F\backslash)$ и $\backslash(G\backslash)$ определяется через операцию произведения объектов:

$$\[(F \times G)X = FX \times GX\]$$

- Постоянный функтор отображает все объекты категории в один объект, а стрелки в тождественную стрелку этого объекта, мы будем обозначать постоянный функтор подчёркиванием:

$$\begin{array}{lll} \[(\underline{A})X\] & \[(=\)] & \[(A)\] \\ \[(\underline{A})f\] & \[(=\)] & \[(id_A)\] \end{array}$$

- Тождественный функтор оставляет объекты и стрелки неизменными:

$$\begin{array}{lll} \[(IX)\] & \[(=\)] & \[(X)\] \\ \[(If)\] & \[(=\)] & \[(f)\] \end{array}$$

- Композиция функторов $\[(F)\]$ и $\[(G)\]$ это последовательное применение функторов

$$\[FGX = F(GX)\]$$

По определению функции построенные с помощью этих операций называют полиномиальными. Определим несколько типов данных с помощью полиномиальных функторов. Определим логические значения:

$$\[Bool = \mu(\underline{1} + \underline{1})\]$$

Объект $\[(1)\]$ обозначает любую константу, это конечный объект исходной категории. Нам не важны имена конструкторов, но важна структура типа. $\[(\mu)\]$ обозначает начальный объект в $\[(F)\]$ -алгебре.

Определим натуральные числа:

$$\[Nat = \mu(\underline{1} + I)\]$$

Эта запись обозначает начальный объект для $\[(F)\]$ -алгебры с функтором $\[(F = \underline{1} + I)\]$. Посмотрим на определение списка:

$$\[List_A = \mu(\underline{1} + \underline{A} \times I)\]$$

Список это начальный объект $\[(F)\]$ -алгебры $\[(\underline{1} + \underline{A} \times I)\]$. Также можно определить бинарные деревья:

```
\[BTree_A = \mu(\underline{A} + I \times I)\]
```

Определим потоки:

```
\[Stream_A = \nu (\underline{A} \times I)\]
```

Потоки являются конечным объектом (F) -коалгебры, где $(F = \underline{A} \times I)$.

Гиломорфизм

Оказывается, что с помощью катаморфизма и анаморфизма мы можем определить функцию `fix`, то есть мы можем выразить любую рекурсивную функцию с помощью структурной рекурсии.

Функция `fix` строит бесконечную последовательность применений некоторой функции `f`.

```
f (f (f ...))
```

Сначала с помощью анаморфизма мы построим бесконечный список, который содержит функцию `f` во всех элементах:

```
repeat f = f : f : f : ...
```

А затем заменим конструктор `:` на применение. В итоге мы получим такую функцию:

```
fix :: (a -> a) -> a
fix = foldr ($) undefined . repeat
```

Убедимся, что эта функция работает:

```
Prelude> let fix = foldr ($) undefined . repeat
Prelude> take 3 $ fix (1:)
[1,1,1]
Prelude> fix (\f n -> if n==0 then 0 else n + f (n-1)) 10
55
```

Теперь давайте определим функцию `fix` через функции `cata` и `ana`:

```
cata :: Functor f => (f a -> a) -> Fix f -> a
cata = fold
ana :: Functor f => (a -> f a) -> a -> Fix f
ana = unfold
fix :: (a -> a) -> a
fix = cata (\(Cons f a) -> f a) . ana (\a -> Cons a a)
```

Эта связка анаморфизм с последующим катаморфизмом встречается так часто, что ей дали специальное имя. *Гиломорфизм* называют функцию:

```
hylo :: Functor f => (f b -> b) -> (a -> f a) -> (a -> b)
hylo phi psi = cata phi . ana psi
```

Отметим, что эту функцию можно выразить и по-другому:

```
hylo :: Functor f => (f b -> b) -> (a -> f a) -> (a -> b)
hylo phi psi = phi . (fmap $ hylo phi psi) . psi
```

Этот вариант более эффективен по расходу памяти: мы не строим промежуточное значение `Fix f`, а сразу обрабатываем значения в функции `phi` по ходу их построения в функции `psi`. Давайте введём инфиксную операцию гиломорфизм для этого определения:

```
(>>) :: Functor f => (a -> f a) -> (f b -> b) -> (a -> b)
psi >> phi = phi . (fmap $ hylo phi psi) . psi
```

Теперь давайте скроем одноимённую функцию из `Prelude` и определим несколько рекурсивных функций с помощью гиломорфизма. Начнём с функции вычисления суммы чисел от нуля до данного числа:

```
sumInt :: Int -> Int
sumInt = range >> sum
```

```
sum x = case x of
  Nil      -> 0
  Cons a b -> a + b
```

```
range n
  | n == 0    = Nil
  | otherwise = Cons n (n-1)
```

Сначала мы создаём в функции `range` список всех чисел от данного числа до нуля, а затем в функции `sum` складываем значения. Теперь мы можем легко определить функцию вычисления факториала:

```
fact :: Int -> Int
fact = range >> prod
```

```
prod x = case x of
  Nil      -> 1
  Cons a b -> a * b
```


Напишем функцию, которая извлекает из потока n-тый элемент.
Сначала определим тип для потока:

```
type Stream a = Fix (S a)

data S a b = a :& b
    deriving (Show, Eq)

instance Functor (S a) where
    fmap f (a :& b) = a :& f b
```

```
headS :: Stream a -> a
headS x = case unFix x of
    (a :& _) -> a
```

```
tails :: Stream a -> Stream a
tails x = case unFix x of
    (_ :& b) -> b
```

Теперь функцию извлечения элемента:

```
getElem :: Int -> Stream a -> a
getElem = curry (enum >> elem)
    where elem ((n, a) :& next)
            | n == 0      = a
            | otherwise   = next
    enum (a, st) = (a, headS st) :& (a-1, tails st)
```

В функции `enum` мы добавляем к элементам потока убывающую последовательность чисел, она стартует из данного числа. Элемент, который нам нужен, будет содержать в этой последовательности число ноль. В функции `elem` мы как раз и извлекаем тот элемент, рядом с которым хранится число ноль. Обратите внимание на то, что рекурсия встроена в этот алгоритм: если данное число не равно нулю, мы просто извлекаем следующий элемент.

С помощью этой функции мы можем вычислить n-тое число из ряда чисел Фибоначчи. Сначала создадим поток чисел Фибоначчи:

```
fibs :: Stream Int
fibs = ana (\(a, b) -> a :& (b, a+b)) (0, 1)
```

Теперь просто извлечём n-тый элемент из потока чисел Фибоначчи:

```
fib :: Int -> Int
fib = flip getElem fibs
```

Вычислим поток всех простых чисел. Мы будем вычислять его по алгоритму “решето Эратосфена”. В начале алгоритма у нас есть поток целых чисел и известно, что первое число является простым.

2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 ...

В процессе этого алгоритма мы вычёркиваем все не простые числа. Сначала мы ищем первое незачёркнутое число и помещаем его в результирующий поток, а на следующий шаг алгоритма мы передаём исходный поток, в котором зачёркнуты все числа кратные тому, что мы положили последним:

2

3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, ...

Теперь мы ищем первое незачёркнутое число и помещаем его в результат. А на следующий шаг рекурсии передаём поток, в котором зачёркнуты все числа кратные новому простому числу:

2, 3

4, 5, 6, 7, 8, 9, 10, 12, 13, 14, 15, ...

И так далее. На каждом шаге мы будем получать одно простое число. Зачёркивание мы будем имитировать с помощью типа `Maybe`. Всё начинается с потока целых чисел, в котором не зачёркнуто ни одно число:

```
nums :: Stream (Maybe Int)
nums = mapS Just $ iterateS (+1) 2
mapS :: (a -> b) -> Stream a -> Stream b
mapS f = ana $ \xs -> (f $ headS xs) :& tailS xs
iterateS :: (a -> a) -> a -> Stream a
iterateS f = ana $ \x -> x :& f x
```

В силу ограничений системы типов Haskell мы не можем определить экземпляр `Functor` для типа `Stream`, поскольку `Stream` является не самостоятельным типом, а типом-синонимом. Поэтому нам приходится определить функцию `mapS`. Определим шаг рекурсии:

```
primes :: Stream Int
primes = ana erato nums
```

```
erato xs = n :& dropWhileS isNothing (erase n xs)
  where n = fromJust $ headS xs
```

Переменная `n` содержит первое незачёркнутое число на данном шаге. Функции `isNothing` и `fromJust` взяты из стандартного модуля `Data.Maybe`. Нам осталось определить лишь две функции. Это аналог функции `dropWhile` на списках: она удаляет из начала списка все элементы, которые удовлетворяют некоторому предикату. Вторая функция `erase` вычёркивает все числа в потоке кратные данному.

```
dropWhileS :: (a -> Bool) -> Stream a -> Stream a
dropWhileS p = psi >> phi
  where phi ((b, xs) :& next) = if b then next else xs
        psi xs = (p $ headS xs, xs) :& tailS xs
```

В этой функции мы сначала генерируем список пар, который содержит значения предиката и остатки списка, а затем находим в этом списке первый такой элемент, значение которого равно `False`.

```
erase :: Int -> Stream (Maybe a) -> Stream (Maybe a)
erase n xs = ana phi (0, xs)
  where phi (a, xs)
    | a == 0    = Nothing :& (a', tailS xs)
    | otherwise = headS xs :& (a', tailS xs)
    where a' = if a == n-1 then 0 else (a+1)
```

В функции `erase` мы заменяем на `Nothing` каждый элемент, порядок следования которого кратен аргументу `n`. Проверим, что у нас получилось:

```
*Fix> primes
(2 :& (3 :& (5 :& (7 :& (11 :& (13 :& (17 :& (19 :& (23 :&
(29 :& (31 :& (37 :& (41 :& (43 :& (47 :& (53 :& (59 :&
(61 :& (67 :& (71 :& (73 :& (79 :& (83 :& (89 :& (97 :&
(101 :& (103 :& (107 :& (109 :& (113 :& (127 :& (131 :&
...)
```

Краткое содержание

В этой главе мы узнали, что любая рекурсивная функция может быть выражена через структурную рекурсию. Мы узнали как в теории категорий определяются типы. Типы являются начальными и конечными объектами в специальных категориях, которые называются алгебрами функторов. Слоган теории категорий гласит:

Управляющие структуры определяются структурой типов.

Определив тип, мы получаем вместе с ним две функции структурной рекурсии, это катаморфизм (для начальных объектов) и анаморфизм (для конечных объектов). С помощью катаморфизма мы можем сворачивать значение данного типа в значения любого другого типа, а с помощью анаморфизма мы можем разворачивать значения данного типа из значений любого другого типа. Также мы узнали, что категория Hask является категорией CPO , категорией полных частично упорядоченных множеств.

Упражнения

- Потренируйтесь в определении рекурсивных функций через гиломорфизм. Попробуйте переписать как можно больше определений из главы о структурной рекурсии в терминах типа `Fix` и функций `cata`, `ana` и `hylo`. Также потренируйтесь на стандартных функциях из модуля `Prelude`. Определите новые типы через `Fix`, например, деревья из модуля `Data.Tree`. Попробуйте свои силы на функциях по-сложнее, например, алгоритме эвристического поиска.
- Определите монадные версии рекурсивных функций:

```
cataM :: (Monad m, Traversable t) => (t a -> m a) -> Fix t -> m a
anaM  :: (Monad m, Traversable t) => (a -> m (t a)) -> (a -> m (Fix t))

hyloM :: (Monad m, Traversable t) => (t b -> m b) -> (a -> m (t a)) ->
(a -> m b)
```

С помощью этих функций мы, например, можем преобразовывать дерево выражения и при этом обновлять какое-нибудь состояние или читать из общего окружения.

В этом определении стоит новый класс `Traversable`. Разберитесь с ним самостоятельно. Немного подскажу. Этот класс появился вместе с классом `Applicative`. Когда разработчики поняли о существовании полезной абстракции, которая ослабляет класс `Monad`, они также обратили внимание на функцию `sequence`:

```
sequence :: Monad m => [m a] -> m [a]
sequence = foldr (liftM2 (·)) (return [])
```

Эту функцию можно записать с помощью одних лишь методов класса `Applicative`. Поэтому ограничение в контексте функции избыточно. Класс `Traversable` предназначен для устранения этой неточности. Посмотрим на основной метод класса:

```
class (Functor t, Foldable t) => Traversable t where
    traverse :: Applicative f => (a -> f b) -> t a -> f (t b)
```

Тип очень похож на тип функции `mapM`. И не случайно, ведь `mapM` определяется через `sequence`. Только теперь вместо списка стоит более общий тип. Это тип `Foldable`, который определяет список как нечто, на чём можно проводить операции свёртки.

Дополнительные возможности

В этой главе мы рассмотрим некоторые дополнительные возможности языка и расширения, которые часто используются в серьёзных программах. Можно писать программы и без них, но с ними гораздо легче и увлекательней.

Пуд сахара

В этом разделе мы рассмотрим специальный синтаксический сахар, который позволяет более кратко записывать операции для некоторых структур.

Сахар для списков

Перечисления

Для класса `Enum` определён специальный синтаксис составления последовательностей перечисляемых значений. Так, например, мы можем составить список целых чисел от нуля до десяти:

```
Prelude> [0 .. 10]
[0,1,2,3,4,5,6,7,8,9,10]
```

А так мы можем составить бесконечную последовательность положительных чисел:

```
Prelude> take 20 $ [0 .. ]
[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19]
```

Мы можем составлять последовательности с определённым шагом. Так можно выделить все чётные положительные числа:

```
Prelude> take 20 $ [0, 2 .. ]
[0,2,4,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38]
```

А так мы можем составить убывающую последовательность чисел:

```
Prelude> [10, 9 .. 0]
[10,9,8,7,6,5,4,3,2,1,0]
```

Что интересно, в списке могут находиться не только числа, но и любые значения из класса `Enum`. Например, определим тип:

```
data Day = Monday | Tuesday | Wednesday | Thursday
         | Friday | Saturday | Sunday
deriving (Show, Enum)
```

Теперь мы можем написать:

```
*Week> [Friday .. Sunday]
[Friday,Saturday,Sunday]
*Week> [ Monday .. ]
[Monday,Tuesday,Wednesday,Thursday,Friday,Saturday,Sunday]
```

Также шаг последовательности может быть и дробным:

```
*Week> [0, 0.5 .. 4]
[0.0,0.5,1.0,1.5,2.0,2.5,3.0,3.5,4.0]
```

Генераторы списков

Генераторы списков (list comprehensions) объединяют в себе функции преобразования и фильтрации списков. Они записываются так:

```
[ f x | x <- list, p x]
```

В этой записи мы фильтруем список `list` предикатом `p` и преобразуем результат функцией `f`. Например, возведём в квадрат все чётные элементы списка:

```
Prelude> [x*x | x <- [1 .. 10], even x]
[4,16,36,64,100]
```

Предикатов может быть несколько. Так, например, мы можем оставить лишь положительные чётные числа:

```
Prelude> [x | x <- [-10 .. 10], even x, x >= 0]
[0,2,4,6,8,10]
```

Также элементы могут браться из нескольких списков, посмотрим на все возможные комбинации букв из пары слов:

```
Prelude> [ [x,y] | x <- "Hello", y <- "World" ]
["HW","Ho","Hr","Hl","Hd","eW","eo","er","el",
 "ed","lW","lo","lr","ll","ld","lW","lo","lr",
 "ll","ld","oW","oo","or","ol","od"]
```

Сахар для монад, do-нотация

Монады используются столь часто, что для них придумали специальный синтаксис, который облегчает подстановку специальных значений в функции нескольких переменных. Монады позволяют комбинировать специальные функции вида

```
a -> m b
```

Если бы эти функции выглядели как обычные функции:

```
a -> b
```

их можно было бы свободно комбинировать с другими функциями. А так нам постоянно приходится пользоваться методами класса `Monad`. Очень часто функции с побочными эффектами имеют вид:

```
a1 -> a2 -> a3 -> ... -> an -> m b
```

А теперь представьте, что вам нужно подставить специальное значение третьим аргументом такой функции и затем передать ещё в одну такую же функцию. Для облегчения участи программистов было придумано специальное окружение `do`, в котором специальные функции комбинируются так, словно они являются обычными. Для этого используется обратная стрелка. Посмотрим, как определяется функция `sequence` в окружении `do`:

```
sequence :: [m a] -> m [a]
sequence []      = return []
sequence (mx:mxs) = do
  x <- mx
  xs <- sequence mxs
  return (x:xs)
```

Во втором уравнении сначала мы говорим вычислителю словом `do` о том, что выражения записаны в мире монады `m`. Запись с перевёрнутой стрелкой `x <- mx` означает, что мы далее в `do`-блоке можем пользоваться значением `x` так, словно оно имеет тип просто `a`, но не `m a`. Смотрите, в этом определении мы сначала извлекаем первый элемент списка, затем извлекаем хвост списка, приведённый к типу `m [a]`, и в самом конце мы соединяем голову и хвост и оборачиваем результат в специальное значение.

Например, мы можем построить функцию, которая дважды читает строку со стандартного ввода и затем возвращает объединение двух строк:

```
getLine2 :: IO String
getLine2 = do
  a <- getLine
  b <- getLine
  return (a ++ b)
```

В **do**-нотации можно вводить локальные переменные с помощью слова :

```
t = do
  b <- f a
  c <- g b
  let x = c + b
      y = x + c
  return y
```

Посмотрим, как **do**-нотация переводится в выражение, составленное с помощью методов класса **Monad**:

```
do
  a <- ma      =>   ma >>= (\a -> exp)
  exp

do
  exp1         =>   exp1 >> exp2
  exp2

do
  let x = fx   =>   let x = fx
                  y = fy       y = fy
  exp          in   exp
```

Переведём с помощью этих правил определение для второго уравнения из функции `sequence`

```
sequence (mx:mxs) = do
  x <- mx                mx >>= (\x -> do
  xs <- sequence mxs    =>   xs <- sequence mxs    =>
  return (x:xs)          return (x:xs))

=>   mx >>= (\x -> sequence mxs >>= (\xs -> return (x:xs)))
```

do или **Applicative**?

С появлением класса **Applicative** во многих случаях **do**-нотация теряет свою ценность. Так, например, любой **do**-блок вида:

```
f mx my = do
  x <- mx
  y <- my
  return (op x y)
```

можно записать гораздо короче:

```
f = liftA2 op
```

Например, напомним функцию, которая объединяет два файла в один:

```
appendFiles :: FilePath -> FilePath -> FilePath -> IO ()
```

С помощью **do**-нотации:

```
appendFiles file1 file2 resFile = do
  a <- readFile file1
  b <- readFile file2
  writeFile resFile (a ++ b)
```

А теперь с помощью класса **Applicative**:

```
appendFiles file1 file2 resFile = writeFile resFile =<<
  liftA2 (++) (readFile file1) (readFile file2)
```

Расширения

Расширение появляется в ответ на проблему, с которой трудно или невозможно справиться в рамках стандарта Haskell. Мы рассмотрим несколько наиболее часто используемых расширений. Расширения подключаются с помощью специального комментария. Он помещается в начале модуля. Расширение действует только в текущем модуле.

```
{-# LANGUAGE ExtentionName1, ExtentionName2, ExtentionName3 #-}
```

Обратите внимание на символ решётки, обрамляющего комментарий. Слово **LANGUAGE** говорит компилятору о том, что мы хотим воспользоваться расширениями с именами **ExtentionName1**, **ExtentionName2**, **ExtentionName3**. Такой комментарий называется *прагмой* (pragma). Часто компилятор ghc в случае ошибки предлагает нам подключить расширение, в котором ошибка уже не будет ошибкой, а возможностью языка. Он говорит: возможно, вы имели в виду расширение **XXX**. Например, попробуйте загрузить в интерпретатор модуль:

```
module Test where
```

```
class Multi a b where
```

В этом случае мы увидим ошибку:

```
Prelude> :l Test
[1 of 1] Compiling Test                ( Test.hs, interpreted )

Test.hs:3:0:
  Too many parameters for class `Multi'
  (Use -XMultiParamTypeClasses to allow multi-parameter classes)
  In the class declaration for `Multi'
Failed, modules loaded: none.
```

Компилятор сообщает нам о том, что у нас слишком много параметров в классе `Multi`. В рамках стандарта Haskell можно создавать лишь классы с одним параметром. Но за сообщением мы видим подсказку: если мы воспользуемся расширением `-XMultiParamTypeClasses`, то всё будет хорошо. В этом сообщении имя расширения закодировано в виде флага. Мы можем запустить `ghc` или `ghci` с этим флагом и тогда расширение будет активировано, и модуль загрузится. Попробуем:

```
Prelude> :q
Leaving GHCi.
$ ghci -XMultiParamTypeClasses
Prelude> :l Test
[1 of 1] Compiling Test                ( Test.hs, interpreted )
Ok, modules loaded: Test.
*Test>
```

Модуль загрузился! У нас есть и другая возможность подключить модуль с помощью прагмы `LANGUAGE`. Имя расширения записано во флаге после символов `-X`. Добавим в модуль `Test` расширение с именем `MultiParamTypeClasses`:

```
{-# LANGUAGE MultiParamTypeClasses #-}
module Test where
class Multi a b where
```

Теперь загрузим `ghci` в обычном режиме:

```
*Test> :q
Leaving GHCi.
$ ghci
Prelude> :l Test
[1 of 1] Compiling Test                ( Test.hs, interpreted )
Ok, modules loaded: Test.
```

Обобщённые алгебраические типы данных

Предположим, что мы хотим написать компилятор небольшого языка. Наш язык содержит числа и логические значения. Мы можем складывать числа и умножать. Для логических значений определена конструкция `if-then-else`. Определим тип синтаксического дерева для этого языка:

```
data Exp = ValTrue
         | ValFalse
         | If Exp Exp Exp
         | Val Int
         | Add Exp Exp
         | Mul Exp Exp
         deriving (Show)
```

В этом определении кроется одна проблема. Наш тип позволяет нам строить бессмысленные выражения вроде `Add ValTrue (Val 2)` или `If (Val 1) ValTrue (Val 22)`. Наш тип `Exp` включает в себя все хорошие выражения и много плохих. Эта проблема проявится особенно ярко, если мы попытаемся определить функцию `eval`, которая вычисляет значения для нашего языка. Получается, что тип этой функции:

```
eval :: Exp -> Either Int Bool
```

Для решения этой проблемы были придуманы *обобщённые алгебраические типы данных* (generalised algebraic data types, GADTs). Они подключаются расширением `GADTs`. Помните, когда-то мы говорили, что типы можно представить в виде классов. Например, определение для списка

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

можно мысленно переписать так:

```
data List a where
  Nil  :: List a
  Cons :: a -> List a -> List a
```

Так вот в GADT определения записываются именно в таком виде. Обобщение заключается в том, что теперь на месте произвольного параметра `a` мы можем писать конкретные типы. Определим тип `Exp`

```
{-# LANGUAGE GADTs #-}
```

```
data Exp a where
  ValTrue    :: Exp Bool
  ValFalse   :: Exp Bool
  If         :: Exp Bool -> Exp a -> Exp a -> Exp a
  Val        :: Int -> Exp Int
  Add        :: Exp Int -> Exp Int -> Exp Int
  Mul        :: Exp Int -> Exp Int -> Exp Int
```

Теперь у нашего типа `Exp` появился параметр, через который мы кодируем дополнительные ограничения на типы операций. Теперь мы не сможем составить выражение `Add ValTrue ValFalse`, потому что оно не пройдет проверку типов.

Определим функцию `eval`:

```
eval :: Exp a -> a
eval x = case x of
  ValTrue    -> True
  ValFalse   -> False
  If p t e    -> if eval p then eval t else eval e
  Val n      -> n
  Add a b     -> eval a + eval b
  Mul a b     -> eval a * eval b
```

Если `eval` получит логическое значение, то будет возвращено значение типа `Bool`, а на значение типа `Exp Int` будет возвращено целое число. Давайте убедимся в этом:

```
*Prelude> :l Exp
[1 of 1] Compiling Exp                ( Exp.hs, interpreted )
Ok, modules loaded: Exp.
*Exp> let notE x = If x ValFalse ValTrue
*Exp> let squareE x = Mul x x
*Exp>
*Exp> eval $ squareE $ If (notE ValTrue) (Val 1) (Val 2)
4
*Exp> eval $ notE ValTrue
False
*Exp> eval $ notE $ Add (Val 1) (Val 2)

<interactive>:1:14:
  Couldn't match expected type `Bool' against inferred type `Int'
  Expected type: Exp Bool
  Actual type: Exp Int
  In the return type of a call of `Add'
  In the second argument of `($)', namely `Add (Val 1) (Val 2)'
```

Сначала мы определили две вспомогательные функции. Затем вычислили несколько значений. Haskell очень часто применяется для

построения компиляторов. Мы рассмотрели очень простой язык, но в более сложном случае суть останется прежней. Дополнительный параметр позволяет нам закодировать в параметре тип функций нашего языка. Спрашивается: зачем нам дублировать вычисления в функции `eval`? Зачем нам сначала кодировать выражение конструкторами, чтобы только потом получить то, что мы могли вычислить и напрямую.

При таком подходе у нас есть полный контроль за деревом выражения: мы можем проводить дополнительную оптимизацию выражений, если нам известны некоторые закономерности. Ещё функция `eval` может вычислять совсем другие значения. Например, она может по виду выражения составлять код на другом языке. Возможно, этот язык гораздо мощнее Haskell по вычислительным способностям, но беднее в плане выразительности, гибкости синтаксиса. Тогда мы будем в функции `eval` проецировать разные конструкции Haskell в конструкции другого языка. Такие программы называются *предметно-ориентированными языками программирования* (domain specific languages). Мы кодируем в типе `Exp` некоторую область и затем надстраиваем над типом `Exp` разные полезные функции. На самом последнем этапе функция `eval` переводит всё дерево выражения в значение или код другого языка.

Отметим, что не так давно было предложено другое решение этой задачи. Мы можем закодировать типы функций в классе:

```
class E exp where
  true  :: exp Bool
  false :: exp Bool
  iff   :: exp Bool -> exp a -> exp a -> exp a
  val   :: exp Int -> exp Int
  add   :: exp Int -> exp Int -> exp Int
  mul   :: exp Int -> exp Int -> exp Int
```

Преимуществом такого подхода является модульность. Мы можем спокойно разделить выражение на две составляющие части:

```
class (Log exp, Arith exp) => E exp
```

```
class Log exp where
  true  :: exp Bool
  false :: exp Bool
  iff   :: exp Bool -> exp a -> exp a -> exp a
```

```
class Arith exp where
  val      :: Int -> exp Int
  add      :: exp Int -> exp Int -> exp Int
  mul      :: exp Int -> exp Int -> exp Int
```

Интерпретация дерева выражения в этом подходе заключается в создании экземпляра класса. Например, создадим класс-вычислитель `Eval`:

```
newtype Eval a = Eval { runEval :: a }

instance Log Eval where
  true  = Eval True
  false = Eval False
  iff p t e = if runEval p then t else e

instance Arith Eval where
  val      = Eval
  add a b = Eval $ runEval a + runEval b
  mul a b = Eval $ runEval a * runEval b

instance E Eval
```

Теперь проведём такую же сессию вычисления значений, но давайте теперь сначала определим их в тексте программы:

```
notE :: Log exp => exp Bool -> exp Bool
notE x = iff x false true

squareE :: Arith exp => exp Int -> exp Int
squareE x = mul x x

e1 :: E exp => exp Int
e1 = squareE $ iff (notE true) (val 1) (val 2)

e2 :: E exp => exp Bool
e2 = notE true
```

Загрузим в интерпретатор:

```
*Exp> :r
[1 of 1] Compiling Exp           ( Exp.hs, interpreted )
Ok, modules loaded: Exp.
*Exp> runEval e1
4
*Exp> runEval e2
False
```

Получились такие же результаты, и в этом случае нам не нужно подключать никаких расширений. Теперь создадим тип-принтер, он будет распечатывать выражение:

```
newtype Print a = Print { runPrint :: String }

instance Log Print where
  true    = Print "True"
  false   = Print "False"
  iff p t e = Print $ "if (" ++ runPrint p ++ ") {"
    ++ runPrint t ++ "}"
    ++ "{" ++ runPrint e ++ "}"

instance Arith Print where
  val n    = Print $ show n
  add a b = Print $ "(" ++ runPrint a ++ ")+(" ++ runPrint b ++ ")"
  mul a b = Print $ "(" ++ runPrint a ++ ")*(" ++ runPrint b ++ ")"

instance E Print
```

Теперь распечатаем предыдущие выражения:

```
*Exp> :r
[1 of 1] Compiling Exp                ( Exp.hs, interpreted )
Ok, modules loaded: Exp.
*Exp> runPrint e1
"(if (if (True) {False}{True}) {1}{2})*(if (if (True) {False}{True}) {1}{2})"
*Exp> runPrint e2
"if (True) {False}{True}"
```

При таком подходе нам не пришлось ничего менять в выражениях: мы просто заменили тип выражения, и оно автоматически подстроилось под нужный результат. Подробнее об этом подходе можно почитать на сайте <http://okmij.org/ftp/tagless-final/course/course.html> или в статье Жака Каре (Jacques Carette), Олега Киселёва (Oleg Kiselyov) и Чунг-Че Шена (Chung-chieh Shan) *Finally Tagless, Partially Evaluated*.

Семейства типов

Семейства типов позволяют выражать зависимости типов. Например, представим, что класс определяет не только методы, но и типы. Причём новые типы зависят от конкретного экземпляра класса. Посмотрим, например, на определение линейного пространства из библиотеки `vector-space`:


```
class AdditiveGroup v where
  zeroV    :: v
  (^+^)    :: v -> v -> v
  negateV  :: v -> v
```

```
class AdditiveGroup v => VectorSpace v where
  type Scalar v    :: *
  (*^)             :: Scalar v -> v -> v
```

Линейное пространство – это математическая структура, объектами которой являются вектора и скаляры. Для векторов определена операция сложения, а для скаляров – операции сложения и умножения. Кроме того, определена операция умножения вектора на скаляр. При этом должны выполняться определённые свойства. Мы не будем подробно на них останавливаться, но вкратце заметим, что эти свойства говорят о том, что мы действительно пользуемся операциями сложения и умножения. В классе `VectorSpace` мы видим новую конструкцию – объявление типа. Мы говорим, что есть производный тип, который следует из `v`. Далее через двойное двоеточие мы указываем его вид. В данном случае это простой тип без параметров.

Вид (`kind`) – это тип типа. Простой тип без параметра обозначается звёздочкой. Тип с параметром обозначается как функция `* -> *`. Если бы тип принимал два параметра, то он обозначался бы `* -> * -> *`. Также параметры могут быть не простыми типами, а типами с параметрами, например, тип, который обозначает композицию типов:

```
newtype O f g a = O { unO :: f (g a) }
```

имеет вид `(* -> *) -> (* -> *) -> * -> *`.

Определим класс векторов на двумерной сетке и сделаем его экземпляром класса `VectorSpace`. Для начала создадим новый модуль с активным расширением `TypeFamilies` и запишем в него классы для линейного пространства

```
{-# Language TypeFamilies #-}
module Point2D where

class AdditiveGroup v where
  ...
```

Теперь определим новый тип:

```
data V2 = V2 Int Int
    deriving (Show, Eq)
```

Сделаем его экземпляром класса `AdditiveGroup`:

```
instance AdditiveGroup V2 where
    zeroV      = V2 0 0
    (V2 x y) ^+^ (V2 x' y') = V2 (x+x') (y+y')
    negateV (V2 x y)      = V2 (-x) (-y)
```

Мы складываем и вычитаем значения в каждом из элементов кортежа. Нейтральным элементом относительно сложения будет кортеж, состоящий из двух нулей. Теперь определим экземпляр для класса `VectorSpace`. Поскольку кортеж состоит из двух целых чисел, скаляр также будет целым числом:

```
instance VectorSpace V2 where
    type Scalar V2 = Int
    s ^* (V2 x y) = V2 (s*x) (s*y)
```

Попробуем вычислить что-нибудь в интерпретаторе:

```
*Prelude> :l Point2D
[1 of 1] Compiling Point2D          ( Point2D.hs, interpreted )
Ok, modules loaded: Point2D.
*Point2D> let v = V2 1 2
*Point2D> v ^+^ v
V2 2 4
*Point2D> 3 ^* v ^+^ v
V2 4 8
*Point2D> negateV $ 3 ^* v ^+^ v
V2 (-4) (-8)
```

Семейства типов дают возможность организовывать вычисления на типах. Посмотрим на такой классический пример. Реализуем в типах числа Пеано. Нам понадобятся два типа: один для обозначения нуля, а другой для обозначения следующего элемента:

```
{-# Language TypeFamilies, EmptyDataDecls #-}
module Nat where

data Zero
data Succ a
```

Значения этих типов нам не понадобятся, поэтому мы воспользуемся расширением `EmptyDataDecls`, которое позволяет определять типы без значений. Значениями будут комбинации типов. Мы определим

операции сложения и умножения для чисел. Для начала определим сложение:

```
type family Add a b :: *
```

```
type instance Add a Zero      = a
type instance Add a (Succ b)  = Succ (Add a b)
```

Первой строчкой мы определили семейство типов `Add`, у которого два параметра. Определение семейства типов начинается с ключевой фразы `type family`. За двоеточием мы указали тип семейства. В данном случае это простой тип без параметра. Далее следуют зависимости типов для семейства `Add`. Зависимости типов начинаются с ключевой фразы `type instance`. В аргументах мы словно пользуемся сопоставлением с образцом, но на этот раз на типах. Первое уравнение

```
type instance Add a Zero      = a
```

говорит о том, что если второй аргумент имеет тип ноль, то мы вернём первый аргумент. Совсем как в обычном функциональном определении сложения для натуральных чисел Пеано. Во втором уравнении мы составляем рекурсивное уравнение:

```
type instance Add a (Succ b)  = Succ (Add a b)
```

Точно также мы можем определить и умножение:

```
type family Mul a b :: *
```

```
type instance Mul a Zero      = Zero
type instance Mul a (Succ b)  = Add a (Mul a b)
```

При этом нам придётся подключить ещё одно расширение `UndecidableInstances`, поскольку во втором уравнении мы подставили одно семейство типов в другое. Этот флаг часто используется в сочетании с расширением `TypeFamilies`. Семейства типов фактически позволяют нам определять функции на типах. Это ведёт к тому, что алгоритм вывода типов становится неопределённым. Если типы правильные, то компилятор сможет это установить, но если они окажутся неправильными, то может возникнуть такая ситуация, что компилятор заикнется и будет бесконечно долго искать соответствие одного типа другому. Теперь проверим результаты. Для

этого мы создадим специальный класс, который будет переводить значения-типы в обычные целочисленные значения:

```
class Nat a where
  toInt :: a -> Int

instance Nat Zero where
  toInt _ = 0

instance Nat a => Nat (Succ a) where
  toInt x = 1 + toInt (proxy x)

proxy :: f a -> a
proxy = undefined
```

Мы определили для каждого значения-типа экземпляр класса `Nat`, в котором мы можем переводить типы в числа. Функция `proxy` позволяет нам извлечь значение из типа-конструктора `Succ`: так мы поясняем компилятору тип значения. При этом мы нигде не пользуемся значениями типов `Zero` и `Succ`, ведь у этих типов нет значений.

Теперь посмотрим, что у нас получилось:

```
Prelude> :l Nat
*Nat> let x = undefined :: (Mul (Succ (Succ (Succ Zero))) (Succ (Succ Zero)))
*Nat> toInt x
6
```

Видно, что с помощью класса `Nat` мы можем извлечь значение, закодированное в типе. Зачем нам эти странные типы-значения? Мы можем использовать их в двух случаях. Мы можем кодировать значения в типе или проводить более тонкую проверку типов.

Помните, когда-то мы определяли функции для численного интегрирования. Там точность метода была жёстко задана в тексте программы:

```
dt :: Fractional a => a
dt = 1e-3

-- метод Эйлера
int :: Fractional a => a -> [a] -> [a]
int x0 ~(f:fs) = x0 : int (x0 + dt * f) fs
```

В этом примере мы можем создать специальный тип потоков, у которых шаг дискретизации будет закодирован в типе.

```
data Stream n a = a :& Stream n a
```

Параметр `n` кодирует точность. Теперь мы можем извлекать точность из типа:

```
dt :: (Nat n, Fractional a) => Stream n a -> a
dt xs = 1 / (fromIntegral $ toInt $ proxy xs)
  where proxy :: Stream n a -> n
        proxy = undefined
```

```
int :: (Nat n, Fractional a) => a -> Stream n a -> Stream n a
int x0 ~(f:&fs) = x0 :& int (x0 + dt fs * f) fs
```

Теперь посмотрим, как мы можем сделать проверку типов более тщательной. Представим, что у нас есть тип матриц. Известно, что сложение определено только для матриц одинаковой размерности, а для умножения матриц число столбцов одной матрицы должно совпадать с числом строк другой матрицы. Мы можем отразить все эти зависимости в целочисленных типах:

```
data Mat n m a = ...
```

```
instance Num a => AdditiveGroup (Mat n m a) where
  a ^+^ b      = ...
  zeroV       = ...
  negateV a    = ...
```

```
mul :: Num a => Mat n m a -> Mat m k a -> Mat n k a
```

При таких определениях мы не сможем сложить матрицы разных размеров. Причём ошибка будет вычислена до выполнения программы. Это освобождает от проверки границ внутри алгоритма умножения матриц. Если алгоритм запустился, то мы знаем, что размеры аргументов соответствуют.

Скоро в `ghc` появится поддержка чисел на уровне типов. Это будет специальное расширение `TypeLevelNats`, при включении которого можно будет пользоваться численными литералами в типах, также будут определены операции-семейства типов на численных типах с привычными именами `+`, `*`.

Классы с несколькими типами

Рассмотрим несколько полезных расширений, относящихся к определению классов и экземпляров классов. Расширение `MultiParamTypeClasses` позволяет объявлять классы с несколькими аргументами. Например, взгляните на такой класс:

```
class Iso a b where
  to      :: a -> b
  from    :: b -> a
```

Так мы можем определить изоморфизм между типами `a` и `b`

Экземпляры классов для синонимов

Расширение `TypeSynonymInstances` позволяет определять экземпляры для синонимов типов. Мы уже пользовались этим расширением, когда определяли рекурсивные типы через тип `Fix`: там нам нужно было определить экземпляр `Num` для синонима `Nat`:

```
type Nat = Fix N

instance Num Nat where
```

В рамках стандарта все суперклассы должны быть простыми. Все они имеют вид `T a`. Если мы хотим использовать суперклассы с составными типами, нам придётся подключить расширение `FlexibleContexts`. Этим расширением мы пользовались, когда определяли экземпляр `Show` для `Fix`:

```
instance Show (f (Fix f)) => Show (Fix f) where
  show x = "(" ++ show (unFix x) ++ ")"
```

Функциональные зависимости

Класс можно представить как множество типов, для которых определены данные операции. С появлением расширения `MultiParamTypeClasses` мы можем определять операции класса для нескольких типов. Так наше множество классов превращается в отношение. Наш класс связывает несколько типов между собой. Если из одной компоненты отношения однозначно следует другая, такое отношение принято называть функцией. Например, обычную функцию одного аргумента можно представить как множество пар $(x, f\ x)$.

Для того чтобы множество таких пар было функцией, необходимо, чтобы выполнялось свойство:

```
forall x, y. x == y => f x == f y
```

Для одинаковых входов мы получаем одинаковые выходы. С функциональными зависимостями мы можем ввести такое ограничение на классы с несколькими аргументами. Рассмотрим практический пример. Библиотека **Boolean** определяет обобщённые логические значения,

```
class Boolean b where
  true, false :: b
  notB         :: b -> b
  (&&*), (||*) :: b -> b -> b
```

Логические значения определены в терминах простейших операций, теперь мы можем обобщить связку **if-then-else** и классы **Eq** и **Ord**:

```
class Boolean bool => IfB bool a | a -> bool where
  ifB :: bool -> a -> a -> a
```

```
class Boolean bool => EqB bool a | a -> bool where
  (==*), (/=*) :: a -> a -> bool
```

```
class Boolean bool => OrdB bool a | a -> bool where
  (<*), (>=), (>*), (<=*) :: a -> a -> bool
```

Каждый из классов определён на двух типах. Один из них играет роль обычных логических значений, а второй тип – это такой же параметр, как и в обычных классах из модуля **Prelude**. В этих определениях нам встретилась новая конструкция: за переменными класса через разделитель “или” следует что-то похожее на тип функции. В этом типе мы говорим, что из типа *a* следует тип *bool*, или тип *a* однозначно определяет тип *bool*. Эта информация помогает компилятору выводить типы. Если он встретит в тексте выражение *v = a <* b* и тип одного из аргументов *a* или *b* известен, то тип *v* будет определён по зависимости.

Зачем нам может понадобиться такая система классов? Например, с ней мы можем определить экземпляр **Boolean** для предикатов или функций вида *a -> Bool* и затем определить три остальных класса для функций вида *a -> b*. Мы сравниваем не отдельные логические

значения, а функции, которые возвращают логические значения. Так в выражении `ifB c t e` функция `c` играет роль “маски”: если на данном значении функция `c` вернёт истину, то мы воспользуемся значением функции `t`, иначе возьмём результат из функции `e`. Например, так мы можем определить функцию модуля:

```
*Boolean> let absolute = ifB (>0) id negate
*Boolean> map absolute [-10 .. 10]
[10,9,8,7,6,5,4,3,2,1,0,1,2,3,4,5,6,7,8,9,10]
```

Мы можем указать несколько зависимостей (через запятую) или зависимость от нескольких типов (через пробел, слева от стрелки):

```
class C a b c | a -> b, b c -> a where
...
```

Отметим, что многие функциональные зависимости можно выразить через семейства типов. Пример из библиотеки `Boolean` можно было бы записать так:

```
class Boolean a where
  true, false      :: a
  (&&*), (||*)      :: a -> a -> a
```

```
class Boolean (B a) => IfB a where
  type B a :: *
  ifB :: (B a) -> a -> a -> a
```

```
class IfB a => EqB a where
  (==*), (/=*) :: a -> a -> B a
```

```
class IfB a => OrdB a where
  (<*), (>*), (>=*), (<=*) :: a -> a -> B a
```

Исторически первыми в Haskell появились функциональные зависимости. Поэтому некоторые пакеты на `Hackage` определены в разных вариантах. Семейства типов используются более охотно.

Ограничение мономорфизма

В Haskell мы можем не писать типы функций - они будут выведены компилятором автоматически. Но написание типов функций считается признаком хорошего стиля, поскольку по типам можно догадаться, чем функция занимается. Но есть в правиле вывода типов одно исключение. Если мы напишем


```
f = show
```

то компилятор сообщит нам об ошибке. Это выражение приводит к ошибке, которая вызвана ограничением мономорфизма. Мы говорили о нём в главе о типах. Часто в сильно обобщённых библиотеках, с большими зависимостями в типах, выписывать типы крайне неудобно. Например, в библиотеке создания парсеров `Parsec` с этим ограничением приходится писать огромные объявления типов для крохотных выражений. Что-то вроде:

```
fun :: (Stream s m t, Show t) => ParsecT s u m a -> ParsecT s u m [a]
fun = g . h (q x) y
```

И так для любого выражения. В этом случае лучше просто выключить ограничение, добавив в начало файла:

```
{-# Language NoMonomorphismRestriction #-}
```

Полиморфизм высших порядков

Когда мы говорили об `ST` нам встретилась функция с необычным типом:

```
runST :: (forall s. ST s a) -> a
```

Слово `forall` обозначает для любых. Любой полиморфный тип в Haskell подразумевает, что он определён для любых типов. Например, когда мы пишем:

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

На самом деле мы пишем:

```
reverse :: forall a. [a] -> [a]
map      :: forall a b. (a -> b) -> [a] -> [b]
```

По названию слова `forall` может показаться, что оно несёт в себе много свободы. Оно говорит о том, что функция определена для любых типов. Но если присмотреться, то эта свобода оказывается жёстким ограничением. “Для любых” означает, что мы не можем делать никаких предположений о внутренней природе значения. Мы не можем разбирать такие значения на составляющие части. Мы можем

только подставлять их в новые полиморфные функции (как в `map`), отбрасывать (как `const`) или перекладывать из одного места в другое (как в `swap` или `reverse`). Мы можем немного смягчить ограничение, если укажем в контексте функции какие классы определены для значений данного типа.

Все стандартные полиморфные типы имеют вид:

```
fun :: forall a b .. z. Expr(a, b, ..., z)
```

Причём `Expr` не содержит `forall`, а только стрелки и применение новых типов к параметрам. Такой тип называют полиморфным типом первого порядка (`rank`). Если `forall` стоит справа от стрелки, то его можно вынести из выражения, например, следующие выражения эквивалентны:

```
fun :: forall a. a -> (forall b. b -> b)
fun :: forall a b. a -> (b -> b)
```

Так мы можем привести нестандартный тип к стандартному. Если же `forall` встречается слева от стрелки, как в функции `runST`, то его уже нельзя вынести. Это приводит к повышению порядка полиморфизма. Порядок полиморфизма определяется как самый максимум среди всех подвыражений, что стоят слева от стрелки плюс один. Так в типе

```
runST :: (forall s. ST s a) -> a
```

слева от стрелки стоит тип первого порядка, прибавив единицу, получим порядок для всего выражения. Если вдруг нам захочется воспользоваться такими типами, мы можем включить одно из расширений:

```
{-# Language Rank2Types #-}
{-# Language RankNTypes #-}
```

В случае рангов произвольного порядка алгоритм вывода типов может не завершиться. В этом случае нам придётся помогать компилятору, расставляя типы сложных функций вручную.

Лексически связанные типы

Мы уже привыкли к тому, что когда мы пишем

```
swap :: (a, b) -> (b, a)
```

компилятор понимает, что `a` и `b` указывают на один и тот же тип слева и справа от стрелки. При этом типы `a` и `b` не обязательно разные. Иногда нам хочется расширить действие контекста функции и распространить его на всё тело функции. Например, ранее в этой главе, когда мы имитировали числа через типы, для того чтобы извлечь число из типа, мы пользовались трюком с функцией `proxy`:

```
instance Nat a => Nat (Succ a) where
  toInt x = 1 + toInt (proxy x)
```

```
proxy :: f a -> a
proxy = undefined
```

Единственное назначение функции `proxy` – это передача информации о типе. Было бы гораздо удобнее написать:

```
instance Nat a => Nat (Succ a) where
  toInt x = 1 + toInt (undefined :: a)
```

Проблема в том, что по умолчанию любой полиморфный тип в Haskell имеет первый ранг – компилятор читает нашу запись как `(x :: forall a. a)`, и получается, что мы говорим: `x` имеет любой тип, какой захочешь! Не очень полезная информация. Компилятор заблудился и спрашивает у нас: “куда пойти?” А мы ему: “да куда захочешь”. Как раз для таких случаев существует расширение `ScopedTypeVariables`. Оно связывает тип, объявленный в заголовке класса/функции с типами, которые встречаются в теле функции. В случае функций есть одно отличие от случая с классами. Если мы хотим расширить действие переменной из объявления типа функции, необходимо упомянуть её в слове `forall` в стандартном положении (как для типа первого порядка). У нас был ещё один пример с `proxy`:

```
dt :: (Nat n, Fractional a) => Stream n a -> a
dt xs = 1 / (fromIntegral $ toInt $ proxy xs)
  where proxy :: Stream n a -> n
        proxy = undefined
```

В этом случае мы пишем:

```
{-# Language ScopedTypeVariables #-}
...
```

```
dt :: forall n. (Nat n, Fractional a) => Stream n a -> a
dt xs = 1 / (fromIntegral $ toInt (undefined :: n))
```

Обратите внимание на появление `forall` в определении типа.

Попробуйте скомпилировать пример без него или переместите его в другое место. Во многих случаях применения этого расширения можно избежать с помощью стандартной функции `asTypeOf`, посмотрим на определение из `Prelude`:

```
asTypeOf :: a -> a -> a
asTypeOf x y = x
```

Фактически это функция `const`, оба типа которой одинаковы. Она часто используется в инфиксной форме для фиксации типа первого аргумента:

```
q = f $ x `asTypeOf` var
```

Получается очень наглядно, словно это предложение обычного языка.

И другие удобства и украшения

Стоит упомянуть несколько расширений. Они лёгкие для понимания – в основном служат украшению записи или для сокращения рутинного кода.

Директива `deriving` может использоваться только с несколькими стандартными классами, но если мы определили тип-обёртку через `newtype` или просто синоним, то мы можем очень просто определить новый тип экземпляром любого класса, который доступен завёрнутому типу. Как раз для этого существует расширение `GeneralizedNewtypeDeriving`:

```
newtype MyDouble = MyDouble Double
    deriving (Show, Eq, Enum, Ord, Num, Fractional, Floating)
```

Мы говорили о том, что обычные числа в Haskell перегружены, но иногда возникает необходимость в перегруженных строках. Как раз для этого существует расширение `OverloadedStrings`. При этом за обычной записью строк может скрываться любой тип из класса:

```
class IsString a where
    fromString :: String -> a
```

Расширение `TypeOperators` позволяет определять инфиксные имена не только для конструкторов данных, но и для самих типов, синонимов типов и даже классов:

```
data a :+: b = Left a | Right b
```

Краткое содержание

В этой главе мы затронули малую часть возможностей, которые предоставляются системой `ghc`. `Haskell` является полигоном для испытания самых разнообразных идей. Это экспериментальный язык. Но в практических целях в 1998 году был зафиксирован стандарт языка, который обычно называют `Haskell98`. Любое расширение подключается с помощью специальной прагмы `Language`. Новый стандарт `Haskell Prime` включит в себя наиболее устоявшиеся расширения. Также мы рассмотрели несколько полезных классов и синтаксических конструкций, которые, возможно, облегчают написание программ.

Упражнения

Это была справочная глава, присмотритесь к рассмотренным возможностям и подумайте, какие нужны вам, а какие нет. Возможно, вы вовсе не будете ими пользоваться, но некоторые из них могут встретиться вам в чужом коде или в библиотеках.

Средства разработки

В этой главе мы познакомимся с основными средствами разработки больших программ. Мы научимся устанавливать и создавать библиотеки, писать документацию.

Пакеты

В Haskell есть ещё один уровень организации данных, мы можем объединять модули в *пакеты* (package). Также как и модули пакеты могут зависеть от других пакетов, если они пользуются модулями этих пакетов. Одним пакетом мы уже пользовались и довольно часто, это пакет `base`, который содержит все стандартные модули, например такие как `Prelude`, `Control.Applicative` или `Data.Function`. Для создания и установки пакетов существует приложение `cabal`. Оно определяет протокол организации и распространения модулей Haskell.

Создание пакетов

Предположим, что мы написали программу, которая состоит из нескольких модулей. Пусть все модули хранятся в директории с именем `src`. Для того чтобы превратить набор модулей в пакет, нам необходимо поместить в одну директорию с `src` два файла:

- `имяПакета.cabal` – файл с описанием пакета.
- `Setup.hs` – файл с инструкциями по установке пакета

.cabal

Посмотрим на простейший файл с описанием библиотеки, этот файл находится в одной директории с той директорией, в которой содержатся все модули приложения и имеет расширение `.cabal`:

```
Name      : Foo
Version    : 1.0
```

Library

```
build-depends : base
exposed-modules : Foo
```

Сначала идут свойства пакета. Общий формат определения свойства:

ИмяСвойства : Значение

В примере мы указали имя пакета `Foo`, и версию `1.0`. После того, как мы указали все свойства, мы определяем будет наш пакет библиотекой или исполняемой программой или возможно он будет и тем и другим. Если пакет будет библиотекой, то мы помещаем за набором атрибутов слово `Library`, а если это исполняемая программа, то мы помещаем слово `Executable`, после мы пишем описание модулей пакета, зависимости от других пакетов, какие модули будут видны пользователю. Формат составления описаний в этой части такой же как и в самом начале файла. Сначала идёт зарезервированное слово-атрибут, затем через двоеточие следует значение. Обратите внимание на отступы за словом `Library`, они обязательны и сделаны с помощью *пробелов*, `cabal` не воспринимает табуляцию.

Файл `.cabal` может содержать комментарии, они делаются также как и в Haskell, закомментированная строка начинается с двойного тире.

Setup.hs

Файл `Setup.hs` содержит информацию о том как устанавливается библиотека. При установке могут использоваться другие программы и библиотеки. Пока мы будем пользоваться простейшим случаем:

```
import Distribution.Simple
main = defaultMain
```

Этот файл позволяет нам создавать библиотеки и приложения, которые созданы только с помощью Haskell. Это не так уж и мало!

Создаём библиотеки

Типичный файл `.cabal` для библиотеки выглядит так:

```
Name:           pinocchio
Version:        1.1.1
Cabal-Version:  >= 1.2
License:        BSD3
License-File:   LICENSE
Author:         Mister Geppetto
Homepage:       http://pinocchio.sourceforge.net/
```

```
Category:      AI
Synopsis:      Tools for creation of woodcrafted robots
Build-Type:    Simple
```

Library

```
Build-Depends: base
Hs-Source-Dirs: src/
Exposed-modules:
  Wood.Robot.Act, Wood.Robot.Percept, Wood.Robot.Think
Other-Modules:
  Wood.Robot.Internals
```

Этим файлом мы описали библиотеку с именем `pinocchio`, версия 1.1.1, она использует версию `cabal` не ниже 1.2. Библиотека выпущена под лицензией BSD3. Файл с лицензией находится в текущей директории под именем `LICENSE`. Автор библиотеки `Mister Geppetto`. Подробнее узнать о библиотеке можно на её домашней странице <http://pinocchio.sourceforge.net/>. Атрибут `Category` указывает на широкую отрасль знаний, к которой принадлежит наша библиотека. В данном случае мы описываем библиотеку для построения роботов из дерева, об этом мы пишем в атрибуте `Synopsis` (краткое описание), поэтому наша библиотека принадлежит к категории искусственный интеллект или сокращённо `AI`. Последний атрибут `Build-Type` указывает на тип сборки пакета. Мы будем пользоваться значением `Simple`, который соответствует сборке с помощью простейшего файла `Setup.hs`, который мы рассмотрели в предыдущем разделе.

После описания пакета, идёт слово `Library`, ведь мы создаём библиотеку. Далее в атрибуте `Build-Depends` мы указываем зависимости для нашего пакета. Здесь мы перечисляем все пакеты, которые мы используем в своей библиотеке. В данном случае мы пользовались лишь стандартной библиотекой `base`. В атрибуте `hs-source-dirs` мы указываем, где искать директорию с исходным кодом библиотеки. Затем мы указываем три внешних модуля, они будут доступны пользователю после установки библиотеки (атрибут `Exposed-Modules`), и внутренние скрытые модули (атрибут `Other-Modules`).

Создаём исполняемые программы

Типичный файл `.cabal` для исполняемой программы:


```
Name:          micro
Version:       0.0
Cabal-Version: >= 1.2
License:       BSD3
Author:        Tony Reeds
Synopsis:       Small programming language
Build-Type:    Simple
```

```
Executable micro
  Build-Depends: base, parsec
  Main-Is:       Main.hs
  Hs-Source-Dirs: micro
```

```
Executable micro-repl
  Main-Is:       Main.hs
  Build-Depends: base, parsec
  Hs-Source-Dirs: repl
  Other-Modules: Utils
```

В этом файле мы описываем две программы. Компилятор языка и интерпретатор языка `micro`. Если сравнить этот файл с файлом для библиотеки, то мы заметим лишь один новый атрибут. Это `Main-Is`. Он указывает в каком модуле содержится функция `main`. После установки этого пакета будут созданы два исполняемых файла. С именами `micro` и `micro-repl`.

Установка пакета

Пакеты устанавливаются с помощью команды `install`. Необходимо перейти в директорию пакета, ту, в которой находятся два служебных файла (`.cabal` и `Setup.hs`) и директория с исходниками, и запустить команду:

```
cabal install
```

Если мы нигде не ошиблись в описании пакета, не перепутали табуляцию с пробелами при отступах, или указали без ошибок все зависимости, то пакет успешно установится. Если это библиотека, то мы сможем подключать экспортируемые ей модули в любом другом модуле, просто указав их в директиве `import`. При этом нам уже не важно, где находятся модули библиотеки. Мы имеем возможность импортировать их из любого модуля. Если же пакет был исполняемой программой, будут созданы бинарные файлы программ. В конце `cabal` сообщит нам куда он их положил.

Иногда возникают проблемы с пакетами, которые генерируют исполняемые файлы, а затем с их помощью устанавливают другие пакеты. Проблема возникает из-за того, что `cabal` может положить бинарный файл в директорию, которая не видна следующим программам, которые хотят продолжить установку. В этом случае необходимо либо переложить созданные бинарные файлы в директорию, которая будет им видна, или добавить директорию с новыми бинарными файлами в `PATH` (под UNIX, Linux). Переменная операционной системы `PATH` содержит список всех путей, в которых система ищет исполняемые программы, если путь не указан явно. Посмотреть содержание `PATH` можно, вызвав:

```
$ echo $PATH
```

Появится строка директорий, которые записаны через двоеточие. Для того чтобы добавить директорию `/data/dir` в `PATH` необходимо написать:

```
$ PATH=$PATH:/data/dir
```

Эта команда добавит директорию в `PATH` для текущей сессии в терминале, если мы хотим записать её навсегда, мы добавим эту команду в специальный скрытый файл `.bashrc`, он находится в домашней директории пользователя. Под Windows добавить директорию в `PATH` можно с помощью графического интерфейса. Кликните правой кнопкой мыши на иконку `My Computer` (Мой Компьютер), в появившемся меню выберите вкладку `Properties` (Свойства). Появится окно `System Properties` (Свойства системы), в нём выберите вкладку `Advanced` и там нажмите на кнопку `Environment variables` (Переменные среды). И в этом окне будет строка `Path`, её мы и хотим отредактировать, добавив необходимые нам пути.

Давайте потренируемся и создадим библиотеку и исполняемую программу. Создадим библиотеку, которая выводит на экран `Hello World`. Создадим директорию `hello`, и в ней создадим директорию `src`. Эта директория будет содержать исходный код. Главный модуль библиотеки экспортирует функцию приветствия:

```
module Hello where
```

```
import Utility.Hello(hello)
```

```
import Utility.World(world)

helloWorld = hello ++ ", " ++ world ++ "!"
```

Главный модуль программы `Main.hs` определяет функцию `main`, которая выводит текст приветствия на экран:

```
module Main where

import Hello

main = print helloWorld
```

У нас будет два внутренних модуля, каждый из которых определяет синоним для одного слова. Мы поместим их в папку `Utility`. Это модуль `Utility.Hello`

```
module Utility.Hello where
hello = "Hello"
```

И модуль `Utility.World`:

```
module Utility.World where
world = "World"
```

Исходники готовы, теперь приступим к описанию пакета. Создадим в корневой директории пакета файл `hello.cabal`.

```
Name:             hello
Version:           1.0
Cabal-Version:     >= 1.2
License:           BSD3
Author:            Anton
Synopsis:          Little example of cabal usage
Category:          Example
Build-Type:       Simple
```

```
Library
  Build-Depends:   base == 4.*
  Hs-Source-Dirs:  src/
  Exposed-modules:
    Hello
  Other-Modules:
    Utility.Hello
    Utility.World
```

```
Executable hello
  Build-Depends:   base == 4.*
  Main-Is:         Main.hs
  Hs-Source-Dirs:  src/
```

В этом файле мы описали библиотеку и программу. В строке `base == 4.*` мы указали версию пакета `base`. Запись `4.*` означает любая версия, которая начинается с четвёрки. Осталось только поместить в корневую директорию пакета файл `Setup.hs`.

```
import Distribution.Simple
main = defaultMain
```

Теперь мы можем переключиться на корневую директорию пакета и установить пакет:

```
anton@anton-desktop:~/haskell-notes/code/ch-17/hello$ cabal install
Resolving dependencies...
Configuring hello-1.0...
Preprocessing library hello-1.0...
Preprocessing executables for hello-1.0...
Building hello-1.0...
[1 of 3] Compiling Utility.World    ( src/Utility/World.hs,
dist/build/Utility/World.o )
[2 of 3] Compiling Utility.Hello     ( src/Utility/Hello.hs,
dist/build/Utility/Hello.o )
[3 of 3] Compiling Hello              ( src/Hello.hs, dist/build/Hello.o )
Registering hello-1.0...
[1 of 4] Compiling Utility.World    ( src/Utility/World.hs,
dist/build/hello/hello-tmp/Utility/World.o )
[2 of 4] Compiling Utility.Hello     ( src/Utility/Hello.hs,
dist/build/hello/hello-tmp/Utility/Hello.o )
[3 of 4] Compiling Hello              ( src/Hello.hs, dist/build/hello/hello-
tmp/Hello.o )
[4 of 4] Compiling Main                ( src/Main.hs, dist/build/hello/hello-
tmp/Main.o )
Linking dist/build/hello/hello ...
Installing library in /home/anton/.cabal/lib/hello-1.0/ghc-7.4.1
Installing executable(s) in /home/anton/.cabal/bin
Registering hello-1.0...
```

Мы видим сообщения о процессе установки. После установки в текущей директории пакета появилась директория `dist`, в которую были помещены скомпилированные файлы библиотеки. В последних строках `cabal` сообщил нам о том, что он установил библиотеку в директорию:

```
Installing library in /home/anton/.cabal/lib/hello-1.0/ghc-7.4.1
```

и исполняемый файл в директорию:

```
Installing executable(s) in /home/anton/.cabal/bin
```

С помощью различных флагов мы можем контролировать процесс установки пакета. Назначать дополнительные директории, указывать куда поместить скомпилированные файлы. Подробно об этом можно почитать в справке, выполнив в командной строке одну из команд:

```
cabal --help
cabal install --help
```

Если у вас не получилось сразу установить пакет не отчаивайтесь и читайте сообщения об ошибках из cabal, он информативно жалуется о забытых зависимостях и неспособности правильно прочитать файл с описанием пакета.

Удаление библиотеки

Установленные с помощью cabal файлы видны из любого модуля. Имена модулей регистрируются глобально. Если нам захочется установить библиотеку с уже зарегистрированным именем, произойдёт хаос. Возможно прежняя библиотека нам уже не нужна. Как нам удалить её? Посмотрим на решение для компилятора ghc. Мы можем посмотреть список всех зарегистрированных в ghc библиотек с помощью команды:

```
$ ghc-pkg list
Cabal-1.8.0.6
array-0.3.0.1
base-4.2.0.2
...
...
```

Появится длинный список с именами библиотек. Для удаления одной из них мы можем выполнить команду:

```
ghc-pkg unregister имя-библиотеки
```

Например так мы можем удалить только что установленную библиотеку hello:

```
$ ghc-pkg unregister hello
```

Репозиторий пакетов Hackage

Если у нас подключен интернет, то мы можем воспользоваться наследием сообщества Haskell и установить пакет с **Hackage**. Там расположено много-много-много пакетов. Любой разработчик Haskell

может добавить свой пакет на **Hackage**. Посмотреть на пакеты можно на сайте этого репозитория:

<http://hackage.haskell.org>

Если для вашей задачи необходимо выполнить какую-нибудь довольно общую задачу, например написать тип красно-чёрных деревьев или построить парсер или возможно вам нужен веб-сервер, поищите этот пакет на **Hackage**, он там наверняка окажется, ещё и в нескольких вариантах.

Для установки пакета с **Hackage** нужно просто написать

```
cabal install имя-пакета
```

Возможно нам нужен очень новый пакет, который был только что залит автором на **Hackage**. Тогда выполняем:

```
cabal update
```

Происходит обновление данных о загруженных на **Hackage**. Что хорошо, вы можете загрузить исходники из **Hackage**, например у вас никак не получается написать пакет, который устанавливался бы без ошибок. Просто загрузим исходники какого-нибудь пакета из **Hackage** и посмотрим на пример рабочего пакета.

Дополнительные атрибуты пакета

В файле `.cabal` также часто указывают такие атрибуты как:

Maintainer

Поле содержит адрес электронной почты технической поддержки. Это те люди, к которым посыпятся сообщения об ошибках или запросы на новые возможности.

Stability

Статус версии библиотеки (`provisional`, `experimental`, `unstable`).

Description

Подробное описание назначения пакета. Оно помещается на главную страницу пакета в документации.

Extra-Source-Files

В этом поле можно через пробел указать дополнительные файлы, включаемые в пакет. Это могут быть примеры использования, описание в формате PDF или хроника изменений и другие служебные файлы. Все эти файлы будут включены в архив проекта.

License-file

Путь к файлу с лицензией.

ghc-options

Флаги компиляции для GHC. Если в нашей библиотеке мы активно пользуемся продвинутыми прагмами оптимизации, необходимо сообщить об этом компилятору пользователя. Например, мы можем написать в этом атрибуте `-O` или `-O2`.

Установка библиотек для профилирования

Помните когда-то мы занимались профилированием? Это было в главе, посвящённой устройству GHC. Мы включали флаг `-prof` и всё шло гладко. Там мы профилировали код, в котором участвовали лишь стандартные библиотеки из пакета `base`, такие как `Prelude`. Но если мы попробуем профилировать код с какими-нибудь другими библиотеками, установленными с помощью `cabal`, GHC возмутится и скажет, что для профилирования не хватает специальной версии библиотеки `имярек`. Для того чтобы иметь возможность профилировать код, в котором участвуют другие библиотеки необходимо установить их с возможностью профилирования. Это делается при установке с помощью специального флага `--enable-library-profiling` или `--enable-executable-profiling` (если мы устанавливаем исполняемое приложение):

```
$ cabal install имярек --reinstall --enable-library-profiling
```

Библиотека будет установлена в двух экземплярах: для исполнения и профилирования. Возможно библиотека итак потребует переустановки некоторых библиотек, от которых она зависит. Повторяем эту процедуру для этих библиотек и возвращаемся к исходной библиотеке. К сожалению, избежать переустановки библиотек нельзя. Но мы можем сделать так, чтобы все будущие библиотеки устанавливались с возможностью профилирования. Для этого необходимо отредактировать файл настроек программы `cabal`. Ищем директорию, в которой `cabal` хранит свои служебные файлы. Если вы пользуетесь Linux, то скорее всего это скрытая директория `.cabal` в вашей домашней директории. Если вы пользуетесь Windows, положение директории зависит от версии системы. Но ничего, узнать её положение можно, выполнив в `ghci`

```
Prelude> :m System.Directory
Prelude System.Directory> getAppUserDataDirectory "cabal"
```

Присмотритесь к этой директории в ней вы найдёте много полезных данных. В ней находятся исполняемые программы, скомпилированные библиотеки, а также исходный код библиотек. В этой директории находится и файл `config` с настройками для `cabal`. Ищем строчку с полем `library-profiling`: `False`. Меняем значение на `True` и раскомментируем эту строчку, если она закомментирована. После этого `cabal install` будет устанавливать библиотеки для профилирования. На первых порах это вызовет массу неудобств из-за необходимости переустановки многих библиотек.

Если пакет не устанавливается

Нам нужен пакет с `Hackage`, но мы никак не можем его установить. Мы видим документацию на странице проекта. Мы понимаем, что он собрался на `Hackage`, но покаким-то причинам он никак не может установиться на нашем компьютере. Рассмотрим типичные загвоздки:

- *Пакету нужны другие пакеты*

Отметим, что очень часто пакет не может установиться из-за какого-то совсем другого пакета, от которого он зависит. И в этом случае мы не отчаиваемся включаем хакерское настроение и читаем дальше!

- *Пакету нужны исполняемые программы, которые получаются из других пакетов*

Такая ошибка возникает очень часто при установке библиотек, которые занимаются разбором (parsing) синтаксиса каких-нибудь языков. Например установка библиотечки `haskell-src-libs` может закончиться с ошибкой, вроде:

```
`alex` not found
```

или

```
`happy` not found
```

`alex` – это программа для создания лексеров на Haskell, а `happy` – это программа для создания парсеров. Смысл ошибки в том, что у нас не установлена Haskell-программа. Скорее всего она есть на Hackage. Хорошо, установим такую программу, например:

```
$ cabal install alex
```

И убедимся в том, что директория, в которую `cabal` устанавливает программы внесена в PATH. Обычно это директория `.cabal/bin`.

Теперь продолжим установку библиотеки:

```
$ cabal install haskell-src-libs
```

- *Пакету нужны Си-библиотеки*

В Haskell есть возможность использовать Си-библиотеки через FFI (Foreign Function Interface). Например, мы можем вызывать функции библиотеки OpenGL из Haskell. Но пакет, который вызывает функции Си-библиотеки, установится успешно, только если Си-библиотека будет установлена. Перед установкой пакет проверяет наличие заголовочных файлов (h-файлы в Си). И в этот момент установка может оборваться с сообщением, вроде

```
fuzzybuzzy.h not found
```

Необходимо установить Си-библиотеку `fuzzybuzzy` вместе с заголовочными файлами перед установкой пакета. В Debian/Ubuntu эта проблема очень часто решается с помощью:

```
$ sudo apt-get install libfuzzybuzzy-dev
```

Часто имя Си-библиотек похоже на `libИмяБибиблиотеки-dev`. Или можно поискать такую библиотеку с помощью:

```
$ apt-cache search fuzzybuzzy
```

- *Конфликт зависимостей*

Пакет зависит от `mtl` версии `2.0.1`, но у нас уже установлен пакет `mtl-3.0` и есть другие пакеты, которые так нужны и не могут зависеть от `mtl-2.0.1`. Как быть? Эту проблему пока невозможно решить с помощью `cabal`. Но есть программа `cabal-dev`. Обычный `cabal` устанавливает все пакеты в одно глобальное пространство пакетов. `cabal-dev` позволяет создавать локальные пространства пакетов или “песочницы” (`sandbox`). Там будут только пакеты, которые нужны только для одного проекта. Причём ни один из пакетов не будет виден глобально. Сначала установим сам `cabal-dev`:

```
$ cabal install cabal-dev
```

Пусть у нас есть проект с `cabal`-файлом. Переключимся в директорию проекта и выполним:

```
$ cabal-dev install
```

Проект установится, причём все пакеты, от которых он зависит будут установлены в созданную в проекте директорию `cabal-dev`. Для первой установки может потребоваться много времени. Также мы можем указать директорию для пакетов песочницы явно:

```
$ cabal-dev install --sandbox=/usr/path/to/local/packages
```

Мы можем устанавливать в песочницу и пакеты с Hackage. Для этого просто пишем `cabal-dev` вместо `cabal` при установке и указываем в какую песочницу установить пакет. Только недавно так устанавливал `faux` – компилятор для создания javascript-кода из Haskell-кода. Для запуска интерпретатора в песочнице наберём:

```
$ cabal-dev ghci
```

`cabal` пока так не умеет. Но как раз сейчас запущен проект для того, чтобы научить его этому.

- *Пакет устарел и зависит от древних версий base или ghc-prim*

Выходят новые версии компилятора ghc, но автор такого нужного нам пакета увлёкся какой-то совсем другой задачей и забыл про этот такой нужный нам пакет. Не беда! Мы можем написать автору. Или, если нам необходимо срочное решение, скачать исходный код пакета и поправить зависимости от base и ghc-prim или других библиотек. Вдруг повезёт и всё установится? Для установки мы переключимся на директорию пакета и установим его локально через `cabal install`.

- *Пакет зависит от операционной системы*

Например, пакет может пользоваться библиотеками, которые есть только в одной из операционных систем, но не в нашей. Такие дела, остаётся только поменять ОС или поискать другой пакет.

Создание документации с помощью Haddock

Если мы зайдём на Hackage, то там мы увидим длинный список пакетов, отсортированных по категориям. К какой категории какой пакет относится мы указываем в .cabal-файле в атрибуте **Category**. Далее рядом с именем пакета мы видим краткое описание, которое берётся из атрибута **Synopsis**. Если мы зайдём на страницу одного из пакетов, то там мы увидим страницу в таком же формате, что и документация к стандартным библиотекам. Мы видим описание пакета и ниже иерархию модулей. Мы можем зайти в заинтересовавший нас модуль и посмотреть на объявленные функции, типы и классы. В самом низу страницы находится ссылка к исходникам пакета.

“Домашняя страница” пакета была создана с помощью приложения **Haddock**. Оно генерирует документацию в формате html по специальным комментариям. **Haddock** встроен в cabal, например, мы можем сделать документацию к нашему пакету hello. Для этого нужно переключиться на корневую директорию пакета и вызвать:

```
cabal haddock
```

После этого в директории dist появится директория doc, в которой внутри директории html находится созданная документация. Мы можем открыть файл index.html, и там мы увидим “иерархию нашего”

модуля. В модуле пока нет ни одной функции: так получилось потому, что **Haddock** помещает в документацию лишь те функции, у которых есть объявление типа. Если мы добавим в модуле **Hello.hs** к единственной функции объявление типа:

```
helloWorld :: String
helloWorld = hello ++ ", " ++ world ++ "!"
```

И теперь перезапустим **haddock**, то мы увидим, что в модуле **Hello** появилась одна запись.

Комментарии к определениям

Прокомментировать любое определение можно с помощью комментария следующего вида:

```
-- | Here is the comment
helloWorld :: String
helloWorld = hello ++ ", " ++ world ++ "!"
```

Обратите внимание на значок “или” сразу после комментариев. Этот комментарий будет включен в документацию. Также можно писать комментарии после определения: для этого к комментарию добавляется значок степени:

```
helloWorld :: String
helloWorld = hello ++ ", " ++ world ++ "!"
-- ^ Here is the comment
```

К сожалению, на момент написания этих строк, **Haddock** может включать в документацию лишь латинские символы. Комментарии могут простираются на несколько строк:

```
-- | Here is the type.
-- It contains three elements.
-- That's it.
data T = A | B | C
```

Также они могут быть блочными:

```
{-|
  Here is the type.
  It contains three elements.
  That's it.
-}
data T = A | B | C
```

Мы можем комментировать не только определение целиком, но и отдельные части. Например, так мы можем пояснить отдельные аргументы у функции:

```
add :: Num a => a    -- ^ The first argument
    -> a             -- ^ The second argument
    -> a             -- ^ The return value
```

Методы класса и отдельные конструкторы типа можно комментировать как обычные функции:

```
data T
    -- | constructor A
  = A
    -- | constructor B
  | B
    -- | constructor C
  | C
```

Или так:

```
data T = A           -- ^ constructor A
      | B           -- ^ constructor B
      | C           -- ^ and so on
```

Комментарии к классу:

```
-- | C-class
class C a where
    -- | f-function
    f :: a -> a
    -- | g-function
    g :: a -> a
```

Комментарии к модулю

Комментарии к модулю помещаются перед объявлением имени модуля. Эта информация попадёт в самое начало страницы документации:

```
-- | Little example
module Hello where
```

Структура страницы документации

Если модуль большой, то его бывает удобно разделить на части, словно разделы в главе книги. Определения группируются по функциональности и помещаются в разные разделы или даже

подразделы. Структура документации определяется с помощью специальных комментариев в экспорте модуля. Посмотрим на пример:

```
-- | Little example
module Hello(
  -- * Introduction
  -- | Here is the little example to show you
  -- how to make docs with Haddock

  -- * Types
  -- | The types.
  T(..),
  -- * Classes
  -- | The classes.
  C(..),
  -- * Functions
  helloWorld
  -- ** Subfunctions1
  -- ** Subfunctions2
) where

...
```

Комментарии со звёздочкой создают раздел, а с двумя звёздочками – подраздел. Те определения, которые экспортируются за комментариями со звёздочкой попадут в один раздел или подраздел. Если сразу за комментарием со звёздочкой идёт комментарий со знаком “или”, то он будет помещён в самое начало раздела. В нём мы можем пояснить по какому принципу группируются определения в данном разделе.

Разметка

С помощью специальных символов можно выделять различные элементы текста, например, ссылки, куски кода, названия определений или модулей. **Haddock** установит необходимые ссылки и выделит элемент в документации.

При этом символы `\`, `'`, ```, `"`, `@`, `<` являются специальными. Если вы хотите воспользоваться одним из специальных символов в тексте, необходимо написать перед ним обратный слэш `\`. Также символы для обозначения комментариев `*`, `|`, `^` и `>` являются специальными, если они расположены в самом начале строки.

Параграфы

Параграфы определяются по пустой строке в комментарии. Так, например, мы можем разбить текст на два параграфа:

```
-- | The first paragraph goes here.
--
-- The second paragraph goes here.
fun :: a -> b
```

Блоки кода

Существует два способа обозначения блоков кода:

```
-- | This documentation includes two blocks of code:
--
-- @
--     f x = x + x
--     g x = x
-- @
--
-- > g x = x * 42
```

В первом варианте мы заключаем блок кода в окружение `@...@`. Так мы можем выделить целый кусок кода. Для выделения одной строки мы можем воспользоваться знаком `>`.

Примеры вычисления в интерпретаторе

В **Haddock** мы можем привести пример вычисления выражения в интерпретаторе. Это делается с помощью тройного символа `>`:

```
-- | Two examples are given bellow:
--
-- >>> 2+3
-- 5
--
-- >>> print 1 >> print 2
-- 1
-- 2
```

Строки, которые идут сразу за строкой с символом `>>>`, помечаются как результат выполнения выражения в интерпретаторе.

Имена определений

Для того чтобы выделить имя любого определения, будь то функция, тип или класс, необходимо заключить его в обычные кавычки, как

в 'T'. При этом **Haddock** установит ссылку к определению и подсветит имя в тексте. Для того чтобы сослаться на определение из другого модуля, необходимо написать его полное имя, то есть с приставкой имени модуля, например, функция `fun`, определённая в модуле `M`, имеет полное имя `M.fun` – тогда в комментариях мы обозначаем её `'M.fun'`.

Ординарные кавычки часто используются в английском языке как апострофы, в таких сочетаниях как `don't`, `isn't`. Перед такими вхождениями ординарных кавычек можно не писать обратный слэш – **Haddock** сумеет отличить их от идентификатора.

Курсив и моноширинный шрифт

Для выделения текста курсивом он заключается в окружение `....`. Для написания текста моноширинным шрифтом он заключается в окружение `@...@`.

Модули

Для обозначения модулей используются двойные кавычки, как в

```
-- | This is a reference to the "Foo" module.
```

Списки

Список без нумерации обозначается с помощью звёздочек:

```
-- | This is a bulleted list:
--
--     * first item
--
--     * second item
```

Пронумерованный список, обозначается символами `(n)` или `n`. (`n` с точкой), где `n` – некоторое целое число:

```
-- | This is an enumerated list:
--
--     (1) first item
--
--     2. second item
```


Список определений

Определения обозначаются квадратными скобками, например, комментарий:

```
-- | This is a definition list:
--
--   [@foo@] The description of @foo@.
--
--   [@bar@] The description of @bar@.
```

в документации будет выглядеть так:

foo

The description of foo.

bar

The description of bar.

Для выделения текста моноширинным шрифтом мы воспользовались окружением @...@.

URL

Ссылки на сайты включаются с помощью окружения <...>.

Ссылки внутри модуля

Для того чтобы сослаться на какой-нибудь текст внутри модуля, его необходимо отметить ссылкой. Для этого мы помещаем в том месте, на которое мы хотим сослаться, запись #label#, где label – это идентификатор ссылки. Теперь мы можем сослаться на это место из другого модуля с помощью записи "module#label", где module – имя модуля, в котором находится ссылка label.

Краткое содержание

В этой главе мы познакомились с основными элементами арсенала разработчика программ. Мы научились создавать библиотеки и документировать их.

Упражнения

Вспомните один из примеров и превратите его в библиотеку.
Например, напишите библиотеку для натуральных чисел Пеано.

Ориентируемся по карте

Рассмотрим задачу поиска маршрута на карте. У нас есть карта метро и нам нужно проложить маршрут от одной станции к другой. Карта метро— это граф, узлы обозначают станции, а рёбра соединяют соседние станции. Предположим, что мы знаем расстояния между всеми станциями и нам надо найти кратчайший путь от станции площадь Баха до станции Таинственный лес.



Схема метрополитена

Давайте переведѐм этот рисунок на Haskell. Сначала опишем имена линий и станций:

```
module Metro where
```

```
data Station = St Way Name
  deriving (Show, Eq)
```

```
data Way = Blue | Black | Green | Red | Orange
  deriving (Show, Eq)
```

```
data Name = Kosmodrom | UlBylichova | Zvezda
  | Zapad | Ineva | De | Krest | Rodnik | Vostok
  | Yug | Sirius | Til | TrollevMost | Prizrak | TainstvenniyLes
  | DnoBolota | PlBakha | Lao | Sever
  | PlShekspira
  deriving (Show, Eq)
```

Предположим, что нам известны координаты каждой из станций. По ним мы можем вычислять расстояние между станциями по прямой:

```
data Point = Point
  { px :: Double
  , py :: Double
  } deriving (Show, Eq)

place :: Name -> Point
place x = uncurry Point $ case x of
  Kosmodrom      -> (-3,7)
  UlBylichova    -> (-2,4)
  Zvezda         -> (0,1)
  Zapad          -> (1,7)
  Ineva          -> (0.5, 4)
  De             -> (0,-1)
  Krest          -> (0,-3)
  Rodnik         -> (0,-5)
  Vostok         -> (-1,-7)
  Yug            -> (-7,-1)
  Sirius         -> (-3,0)
  Til            -> (3,2)
  TrollevMost    -> (5,4)
  Prizrak        -> (8,6)
  TainstvenniyLes -> (11,7)
  DnoBolota      -> (-7,-4)
  PlBakha        -> (-3,-3)
  Lao            -> (3.5,0)
  Sever          -> (6,1)
  PlShekspira    -> (3,-3)

dist :: Point -> Point -> Double
dist a b = sqrt $ (px a - px b)^2 + (py a - py b)^2

stationDist :: Station -> Station -> Double
stationDist (St n a) (St m b)
  | n /= m && a == b = penalty
  | otherwise        = dist (place a) (place b)
  where penalty = 1
```

Расстояние между точками вычисляется по формуле Евклида (dist). Если у станций одинаковые имена, но они расположены на разных линиях мы будем считать, что расстояние между ними равно единице. Теперь нам необходимо описать связность станций. Мы опишем связность в виде функции, которая для данной станции возвращает список всех соседних с ней станций:

```
metroMap :: Station -> [Station]
metroMap x = case x of
    St Black Kosmodrom      -> [St Black UlBylichova]
    St Black UlBylichova    ->
        [St Black Kosmodrom, St Black Zvezda, St Red UlBylichova]
    St Black Zvezda         ->
        [St Black UlBylichova, St Blue Zvezda, St Green Zvezda]
    ...
```

Приведён пример заполнения только для одной линии. Остальные линии заполняются аналогично. Обратите внимание на то, что некоторые станции имеют одинаковые имена, но находятся на разных линиях.

Всё готово для того чтобы написать функцию поиска маршрута. Для этого мы воспользуемся алгоритмом A*.

Алгоритм эвристического поиска A*

Наша задача относится к задачам поиска путей на графе. Путём на графе называют такую последовательность узлов, в которой для любых двух соседних узлов существует ребро, которое их соединяет. В нашем случае графом является карта метро, узлами~– станции, рёбрами~– линии между станциями, а путями~– маршруты.

Представим, что мы находимся в узле **A** и нам необходимо попасть в узел **B** и единственное, что нам известно~– это все соседние узлы с тем, в котором мы находимся. У нас есть возможность перейти в один из соседних узлов и посмотреть нет ли среди их соседей узла **B**. В этом случае нам ничего не остаётся кроме того как бродить по карте от станции к станции в случайном порядке, пока мы не натолкнёмся на узел **B** или все узлы не кончатся. Такой поиск называют слепым.

Вот если бы у нас был компас, который в каждой точке указывал в сторону цели нам было бы гораздо проще. Такой компас принято называть *эвристикой*. Это функция, которая принимает узел и возвращает число. Чем меньше число, тем ближе узел к цели. Обычно эвристика указывает не точное расстояние до цели, поскольку мы не знаем где цель, а приблизительную оценку. Мы не знаем расстояние до цели, но догадываемся, нам кажется, что она где-то там, ещё

чуть-чуть и мы найдём её. Примером эвристики для поиска по карте может быть функция, которая вычисляет расстояние по прямой до цели. Предположим, что мы не знаем где находится цель (какая дорога к ней ведёт), но мы знаем её координаты. Также мы знаем координаты каждой вершины, в которой мы находимся. Тогда мы можем легко вычислить расстояние по прямой до цели и наш поиск станет гораздо более осмысленным.

Так находясь в точке **A** мы можем сразу пойти в тот соседний узел, который ближе всех к цели. Такой поиск называют поиском по первому лучшему приближению. В поиске A^* учитывается не только расстояние до цели, но и то расстояние, которое мы уже прошли. Мы выбираем не ту вершину, которая ближе к цели, а ту для которой полный путь до цели будет минимальным. Ведь пока мы идём мы можем запоминать какое расстояние мы уже прошли. Прибавив к этому значению, то которое мы получим с помощью эвристики мы получим полный (предполагаемый) путь до цели.

Поиск A^* гораздо лучше поиска по первому лучшему приближению. Его часто применяют в компьютерных играх для поиска пути или принятия решений.

Принято разделять поиск на графе и поиск на дереве. Если мы идём по графу, то вершины могут повторяться (они образуют циклы). В случае поиска на дереве мы считаем, что все вершины уникальны. При поиске на графе очень важно запоминать те вершины, в которых мы уже побывали. Иначе мы будем очень часто ходить кругами.

В Haskell очень удобно работать с данными, которые имеют иерархическую структуру. Их можно представить в виде дерева, обычно в таких типах у нас есть конструкторы-константы и конструкторы, которые собирают составные значения. Граф выходит за рамки этого класса данных, потому что рёбра графов могут образовывать циклы. Но мы схитрим и представим граф поиска в виде дерева. Корнем нашего дерева будет начальная точка поиска, а поддеревьями для данной вершины узла будут все вершины-соседи. В таком дереве будет очень много повторяющихся узлов, так например мы можем пойти в соседнюю вершину, потом вернуться обратно, опять пойти в ту же соседнюю вершину, и так до бесконечности. Для того,

чтобы избежать подобных ситуаций мы будем запоминать те вершины, в которых мы уже побывали и не рассматривать их, если они встретятся нам ещё раз.

Сформулируем задачу поиска в типах. У нас есть дерево поиска, которое содержит все возможные разветвления, также каждая вершина содержит значение эвристики, по нему мы знаем насколько близка данная вершина к цели. Также у нас есть специальный предикат, который определён на вершинах, по нему мы можем узнать является ли данная вершина целью. Нам нужно получить путь, или цепочку вершин, которая будет начинаться в корне дерева поиска и заканчиваться в целевой вершине.

```
search :: Ord h => (a -> Bool) -> Tree (a, h) -> Maybe [a]
```

Здесь `a` – это значение вершины и `h` – значение эвристики. Обратите внимание на зависимость `Ord h` в контексте, ведь мы собираемся сравнивать эти значения по близости к цели. При обходе дерева мы будем запоминать повторяющиеся вершины, для этого мы воспользуемся типом множество из стандартного модуля `Data.Set`. Внутри `Set` могут храниться только значения, для которых определены операции сравнения, поэтому нам придётся добавить в контекст ещё одну зависимость:

```
import Data.Tree
import qualified Data.Set as S
```

```
search :: (Ord h, Ord a) => (a -> Bool) -> Tree (a, h) -> Maybe [a]
```

Поиск будет заключаться в том, что мы будем обходить дерево от корня к узлам. При этом среди всех узлов-альтернатив мы будем просматривать узлы с наименьшим значением эвристики. В этом нам поможет специальная структура данных, которая называется *очередью с приоритетом* (priority queue). Эта очередь хранит элементы с учётом их старшинства (приоритета). Мы можем добавлять в неё элементы и извлекать элементы. При этом мы всегда будем извлекать элемент с наименьшим приоритетом. Мы воспользуемся очередями из библиотеки `fingertree`. Для начала установим библиотеку:

```
cabal install fingertree
```

Теперь посмотрим в документацию и узнаем какие функции нам доступны. Документацию к пакету можно найти на сайте <http://hackage.haskell.org/package/fingertree>. Пока отложим детальное изучение интерфейса, отметим лишь то, что мы можем добавлять элементы к очереди и извлекать элементы с учётом приоритета:

```
insert  :: Ord k => k -> v -> PQueue k v -> PQueue k v
minView :: Ord k => PQueue k v -> Maybe (v, PQueue k v)
```

Вернёмся к функции `search`. Я бы хотел обратить ваше внимание на то, как мы будем разрабатывать эту функцию. Вспомним, что Haskell – ленивый язык. Это означает, что при обработке рекурсивных типов данных, функция “углубляется” в значение лишь тогда, когда функция, которая вызвала эту функцию попросит её об этом. Это даёт нам возможность работать с потенциально бесконечными структурами данных и, что более важно, разделять сложный алгоритм на независимые составляющие.

В функции `search` нам необходимо обойти все элементы в порядке значения эвристики и остановиться в вершине, на которой целевой предикат вернёт `True`. Но для начала мы добавим к вершинам их пути из корня, для того чтобы в конце мы смогли узнать как мы попали в текущую вершину. Итак наша функция разбивается на три составляющие:

```
search :: (Ord h, Ord a) => (a -> Bool) -> Tree (a, h) -> Maybe [a]
search isGoal = findPath isGoal . flattenTree . addPath
```

выпишем типы составляющих функций и проверим код в интерпретаторе.

```
un = undefined
```

```
findPath :: (a -> Bool) -> [Path a] -> Maybe [a]
findPath = un
```

```
flattenTree :: (Ord h, Ord a) => Tree (Path a, h) -> [Path a]
flattenTree = un
```

```
addPath :: Tree (a, h) -> Tree (Path a, h)
addPath = un
```



```
data Path a = Path
  { pathEnd    :: a
  , path       :: [a]
  }
```

Обратите внимание на то как поступающие на вход данные разделились между функциями. Информация о приоритете вершин не идёт дальше функции `flattenTree`, а предикат `isGoal` используется только в функции `findPath`. Модуль прошёл проверку типов и мы можем детализировать функции дальше:

```
addPath :: Tree (a, h) -> Tree (Path a, h)
addPath = iter []
  where iter ps t = Node (Path val (reverse ps'), h) $
    iter ps' <$> subForest t
    where (val, h) = rootLabel t
          ps'      = val : ps
```

В этой функции мы просто присоединяем к данной вершине все родительские вершины, так мы составляем маршрут от данной вершины до начальной, поскольку мы всё время добавляем новые вершины в начало списка, в итоге у нас получаются перевёрнутые маршруты, поэтому перед тем как обернуть значение в конструктор `Path` мы переворачиваем список. На самом деле нам нужно перевернуть только один путь. Путь, который ведёт к цели, но за счёт того, что язык у нас ленивый, функция `reverse` будет применена не сразу, а лишь тогда, когда нам действительно понадобится значение пути. Это как раз и произойдёт лишь один раз, в самом конце программы, лишь для одного значения!

Давайте пока пропустим функцию `flattenTree` и сначала определим функцию `findPath`. Эта функция принимает все вершины, которые мы обошли если бы шли без цели (функции `isGoal`) и ищет среди них первую, которая удовлетворяет предикату. Для этого мы воспользуемся стандартной функцией `find` из модуля `Data.List`:

```
findPath :: (a -> Bool) -> [Path a] -> Maybe [a]
findPath isGoal = fmap path . find (isGoal . pathEnd)
```

Напомню тип функции `find`, она принимает предикат и список, а возвращает первое значение списка, на котором предикат вернёт `True`:

```
find :: (a -> Bool) -> [a] -> Maybe a
```

Функция `fmap` применяется из-за того, что результат функции `find` завернут в `Maybe`, это частично определённая функция. В самом деле ведь в списке может и не оказаться подходящего значения.

Осталось определить функцию `flattenTree`. Было бы хорошо определить её так, чтобы она была развёрткой для списка. Поскольку функция `find` является свёрткой (может быть определена через `fold`), вместе эти функции работали бы очень эффективно. Мы определим функцию `flattenTree` через взаимную рекурсию. Две функции будут по очереди вызывать друг друга. Одна из них будет извлекать следующее значение из очереди, а другая – проверять не встречалось ли нам уже такое значение, и добавлять новые элементы в очередь.

```
flattenTree :: (Ord h, Ord a) => Tree (Path a, h) -> [Path a]
flattenTree a = ping none (singleton a)
```

```
ping :: (Ord h, Ord a) => Visited a -> ToVisit a h -> [Path a]
ping visited toVisit
  | isEmpty toVisit = []
  | otherwise      = pong visited toVisit' a
  where (a, toVisit') = next toVisit
```

```
pong :: (Ord h, Ord a)
      => Visited a -> ToVisit a h -> Tree (Path a, h) -> [Path a]
pong visited toVisit a
  | inside a visited = ping visited toVisit
  | otherwise        = getPath a :
    ping (insert a visited) (schedule (subForest a) toVisit)
```

Типы `Visited` и `ToVisit` обозначают наборы вершин, которые мы уже посетили и которые только собираемся посетить. Не вдаваясь в подробности интерфейса этих типов, давайте присмотримся к функциям `ping` и `pong` с точки зрения функции, которая их будет вызывать, а именно функции `findPath`. Эта функция ожидает на входе список. Внутри она обходит список в поисках нужного элемента, поэтому она будет применять сопоставление с образцом, разбирая список на части. Сначала она запросит сопоставление с пустым списком, запустится функция `ping` с пустым множеством посещённых вершин (`none`) и одним элементом в очереди вершин (`singleton a`),

которые предстоит посетить. Функция `ping` проверит не является ли очередь пустой, очередь содержит один элемент, поэтому она перейдёт к следующему случаю и извлечёт из очереди один элемент (`next`), который будет передан в функцию `pong`. Функция `pong` проверит нет ли в списке уже посещённых элементов того, который был только что извлечён (`inside a visited`). Если это окажется так, то она запросит следующий элемент у функции `ping`. Если же исходный элемент окажется новым, она добавит его в список (`getPath a : ...`) и запланирует обход всех дочерних деревьев данного элемента (`schedule (subForest a) toVisit`). При первом заходе исходный элемент окажется новым и функция `findPath` поймёт, что список не пустой и остановит вычисление. Она немного передохнёт и примется за следующий случай. Там она будет извлекать первый элемент списка и сопоставлять его с предикатом. При этом первый элемент уже вычислен. Мы воспользуемся этим, убедимся в том, что он не является целью и рекурсивно вызовем функцию `find` на хвосте списка. Функция `findPath` запросит следующее значение и так далее.

Наша функция `flattenPath` не является развёрткой, но очень похожа на неё тем, что позволяет вычислять результирующий список частично. Например функция `length` требует полного обхода списка. Мы не можем использовать её с бесконечными списками. Теперь давайте разберёмся с подчинёнными функциями:

```
getPath :: Tree (Path a, h) -> Path a
getPath = fst . rootLabel
```

Функции для множества вершин, которые мы уже посетили:

```
import qualified Data.Set as S
...
```

```
type Visited a = S.Set a
```

```
none :: Ord a => Visited a
none = S.empty
```

```
insert :: Ord a => Tree (Path a, h) -> Visited a -> Visited a
insert = S.insert . pathEnd . getPath
```

```
inside :: Ord a => Tree (Path a, h) -> Visited a -> Bool
inside = S.member . pathEnd . getPath
```

Функции для очереди тех вершин, что мы только собираемся посетить:

```
import Data.Maybe
import qualified Data.PriorityQueue.FingerTree as Q
...

type ToVisit a h = Q.PQueue h (Tree (Path a, h))

priority t = (snd $ rootLabel t, t)

singleton :: Ord h => Tree (Path a, h) -> ToVisit a h
singleton = uncurry Q.singleton . priority

next :: Ord h => ToVisit a h -> (Tree (Path a, h), ToVisit a h)
next = fromJust . Q.minView

isEmpty :: Ord h => ToVisit a h -> Bool
isEmpty = Q.null

schedule :: Ord h => [Tree (Path a, h)] -> ToVisit a h -> ToVisit a h
schedule = Q.union . Q.fromList . fmap priority
```

Эти функции очень простые, они специализируют более общие функции для типов `Set` и `PQueue`, вы наверняка легко разберётесь с ними, заглянув в документацию к модулям `Data.Set` и `Data.PriorityQueue.FingerTree`.

Осталось только написать функцию, которая будет составлять дерево поиска для алгоритма A*. Она принимает функцию ветвления, а также функцию расстояния до цели и строит по ним дерево поиска:

```
astarTree :: (Num h, Ord h)
=> (a -> [(a, h)]) -> (a -> h) -> a -> Tree (a, h)
astarTree alts distToGoal s0 = unfoldTree f (s0, 0)
  where f (s, h) = ((s, heur h s), next h <$> alts s)
        heur h s = h + distToGoal s
        next h (a, d) = (a, d + h)
```

Поиск маршрутов в метро

Теперь давайте посмотрим как наша функция справится с задачей поиска маршрутов в метро:

```
metroTree :: Station -> Station -> Tree (Station, Double)
metroTree init goal = astarTree distMetroMap (stationDist goal) init
connect :: Station -> Station -> Maybe [Station]
connect a b = search (== b) $ metroTree a b
```

```
main = print $ connect (St Red Sirius) (St Green Prizrak)
```

К примеру найдём маршрут от станции “Дно Болота” до станции “Призрак”:

```
*Metro> connect (St Orange DnoBolota) (St Green Prizrak)
Just [St Orange DnoBolota,St Orange PlBakha,
      St Red PlBakha,St Red Sirius,St Green Sirius,
      St Green Zvezda,St Green Til,
      St Green TrollevMost,St Green Prizrak]
*Metro> connect (St Red PlShekspira) (St Blue De)
Just [St Red PlShekspira,St Red Rodnik,St Blue Rodnik,
      St Blue Krest,St Blue De]
*Metro> connect (St Red PlShekspira) (St Orange De)
Nothing
```

В третьем случае маршрут не был найден, поскольку у нас нет станции **De** на оранжевой ветке.

Тестирование с помощью QuickCheck

Мы проверили три случая, ещё три случая, ещё три случая, ожидаемый результат сходится с тем, что возвращает нам интерпретатор, но можем ли мы быть уверены в том, что алгоритм действительно работает? Для Haskell была разработана специальная библиотека тестирования **QuickCheck**, которая упрощает процесс проверки программ. Мы можем сформулировать свойства, которые обязательно должны выполняться, а **QuickCheck** сгенерирует случайный набор данных и проверит наши свойства на них.

Например в нашей задаче путь из **A** в **B** должен совпадать с перевёрнутым путём из **B** в **A**. Также все станции в маршруте должны быть соседними. Давайте проверим эти свойства. Для этого нам нужно сформулировать их в виде предикатов:

```
module Test where
```

```
import Control.Applicative
```

```
import Metro
```

```
prop1 :: Station -> Station -> Bool
```

```
prop1 a b = connect a b == (fmap reverse $ connect b a)
```

```
prop2 :: Station -> Station -> Bool
```

```
prop2 a b = maybe True (all (uncurry near) . pairs) $ connect a b
```

```

pairs :: [a] -> [(a, a)]
pairs xs = zip xs (drop 1 xs)
near :: Station -> Station -> Bool
near a b = a `elem` (fst <$> distMetroMap b)

```

Установим `QuickCheck`:

```
cabal install QuickCheck
```

Теперь нам нужно подсказать `QuickCheck` как генерировать случайные значения типа `Station`. `QuickCheck` тестирует функции, которые принимают значения из класса `Arbitrary` и возвращают `Bool`. Класс `Arbitrary` отвечает за генерацию случайных значений.

Основной метод `arbitrary` возвращает генератор случайных значений:

```

class Arbitrary a where
  arbitrary :: Gen a

```

Мы воспользуемся тем, что этот класс уже определён для многих стандартных типов. Кроме того класс `Gen` является монадой. Мы сгенерируем случайное целое число и отобразим его в одну из станций. Сделать это можно разными способами, мы начнём из одной станции и будем случайно блуждать по карте:

```

import Test.QuickCheck
...
instance Arbitrary Station where
  arbitrary = ($ s0) . foldr (.) id . fmap select <$> ints
    where ints = vector ==<< choose (0, 100)
          s0 = St Blue De

select :: Int -> Station -> Station
select i s = as !! mod i (length as)
    where as = fst <$> distMetroMap s

```

Мы воспользовались двумя функциями из библиотеки `QuickCheck`. Это `vector` и `choose`. Первая строит список случайных чисел заданной длины, а вторая выбирает случайное число из заданного диапазона. Теперь мы можем протестировать наши предикаты с помощью функции `quickCheck`:

```

*Test Prelude> quickCheck prop1
+++ OK, passed 100 tests.
*Test Prelude> quickCheck prop2
+++ OK, passed 100 tests.
*Test Prelude>

```

Свойства прошли тестирование на выборке из 100 комбинаций аргументов. Если нам интересно, мы можем с помощью функции `verboseCheck` посмотреть на каких именно значениях проводилось тестирование:

```
*Test Prelude> verboseCheck prop2
Passed:
St Black Kosmodrom
St Red UlBylichova
Passed:
St Black UlBylichova
St Orange Sever
Passed:
St Red Sirius
St Blue Krest
...
```

Если бы свойство не выполнилось, `QuickCheck` сообщил бы нам об этом и показал бы те элементы, для которых свойство не выполнилось. Давайте составим такое свойство искусственно. Например, проверим, находятся ли все станции на одной линии:

```
fakeProp :: Station -> Station -> Bool
fakeProp (St a _) (St b _) = a == b
```

Посмотрим, что на это скажет `QuickCheck`:

```
*Test Prelude> quickCheck fakeProp
*** Failed! Falsifiable (after 1 test):
St Green Sirius
St Blue Rodnik
```

По умолчанию `QuickCheck` проверит свойство сто раз. Для изменения этих настроек, мы можем воспользоваться функцией `quickCheckWith`, дополнительным параметром она принимает значение типа `Arg`, которое содержит параметры тестирования. Например протестируем первое свойство 500 раз:

```
*Test> quickCheckWith (stdArgs{ maxSuccess = 500 }) prop1
+++ OK, passed 500 tests.
```

Мы воспользовались стандартными настройками (`stdArgs`) и изменили один параметр.

Формирование тестовой выборки

Предположим, что мы уверены в правильной работе алгоритма для голубой и чёрной ветки метро, но сомневаемся в остальных. Как раз для этого случая в **QuickCheck** предусмотрена функция `a==>b`. Это функция обозначает условную проверку, свойство `b` будет протестировано только в том случае, если свойство `a` окажется верным. Иначе тестовые данные будут отброшены.

```
notBlueAndBlack a b = cond a && cond b ==> prop1 a b
  where cond (St a _) = a /= Blue && a /= Black
```

Далее тестируем как обычно:

```
*Test> quickCheck notBlueAndBlack
+++ OK, passed 100 tests.
```

Также с помощью функции `forAll` мы можем подсказать **QuickCheck** на каких данных тестировать свойство.

```
forAll :: (Show a, Testable prop) => Gen a -> (a -> prop) -> Property
```

Эта функция принимает генератор случайных значений и свойство, которое зависит от тех значений, которые создаются этим генератором. К примеру, пусть нас интересуют только все возможные пути между четырьмя станциями: `(St Blue De)`, `(St Red Lao)`, `(St Green Til)` и `(St Orange Sever)`. Воспользуемся функцией `elements :: [a] -> Gen a`, она как раз принимает список значений, и возвращает генератор, который случайным образом выбирает любое значение из этого списка.

```
testFor = forAll (liftA2 (,) gen gen) $ uncurry prop1
  where gen = elements [St Blue De, St Red Lao,
                        St Green Til, St Orange Sever]
```

Проверим, те ли значения попали в выборку:

```
*Test> verboseCheckWith (stdArgs{ maxSuccess = 3 }) testFor
Passed:
(St Blue De,St Orange Sever)
Passed:
(St Orange Sever,St Red Lao)
Passed:
(St Red Lao,St Red Lao)
+++ OK, passed 3 tests.
```


Мы можем настроить формирование выборки ещё одним способом. Для этого мы сделаем специальный тип обёртку над `Station` и определим для него свой экземпляр класса `Arbitrary`:

```
newtype OnlyOrange = OnlyOrange Station
newtype Only4      = Only4      Station

instance Arbitrary OnlyOrange where
  arbitrary = OnlyOrange . St Orange <$>
    elements [DnoBolota, PlBakha, Krest, Lao, Sever]

instance Arbitrary Only4 where
  arbitrary = Only4 <$> elements [St Blue De, St Red Lao,
    St Green Til, St Orange Sever]
```

После этого мы можем очень легко комбинировать различные выборки при тестировании.

```
*Test> quickCheck $ \(Only4 a) (Only4 b) -> prop1 a b
+++ OK, passed 100 tests.
*Test> quickCheck $ \(Only4 a) (OnlyOrange b) -> prop1 a b
+++ OK, passed 100 tests.
*Test> quickCheck $ \(a (OnlyOrange b) -> prop2 a b
+++ OK, passed 100 tests.
```

Классификация тестовых случаев

Мы можем попросить у `QuickCheck`, чтобы он разбил тестовую выборку на классы и в конце тестирования сообщил бы нам сколько элементов в какой класс попали. Это делается с помощью функции `classify`:

```
classify :: Testable prop => Bool -> String -> prop -> Property
```

Она принимает условие классификации, метку класса и свойство. Например так мы можем разбить выборку по типам линий:

```
prop3 :: Station -> Station -> Property
prop3 a@(St wa _) b@(St wb _) =
  classify (wa == Orange | wb == Orange) "Orange" $
  classify (wa == Black | wb == Black) "Black" $
  classify (wa == Red | wb == Red) "Red" $ prop1 a b
```

Протестируем:

```
*Test> quickCheck prop3
+++ OK, passed 100 tests:
34% Red
15% Orange
```

```
9% Black
8% Orange, Red
6% Black, Red
5% Orange, Black
```

Оценка быстродействия с помощью criterion

Недавно появилась библиотека `unordered-containers`. Она предлагает более эффективную реализацию нескольких структур из стандартной библиотеки `containers`. Например там мы можем найти тип `HashSet`. Почему бы нам не заменить на него стандартный тип `Set`?

```
cabal install unordered-containers
```

Изменения отразятся лишь на контекстах объявлений типов.

Элементы, принадлежащие множеству `HashSet`, должны быть экземплярами классов `Eq` и `Hashable`. Новый класс `Hashable` нужен для ускорения работы с данными. Давайте посмотрим на этот класс:

```
Prelude> :m Data.Hashable
Prelude Data.Hashable> :i Hashable
class Hashable a where
  hash :: a -> Int
  hashWithSalt :: Int -> a -> Int
  -- Defined in `Data.Hashable'
...
... много экземпляров
```

Обязательный метод класса `hash` даёт нам возможность преобразовать элемент в целое число. Это число называют хеш-ключом. Хеш-ключи используются для хранения элементов в хеш-таблицах. Мы не будем подробно на них останавливаться, отметим лишь то, что они позволяют очень быстро извлекать данные из контейнеров и обновлять данные.

Теперь просто скопируйте модуль `Astar.hs` измените одну строчку, и добавьте ещё одну (в шапке модуля):

```
import qualified Data.HashSet as S
import Data.Hashable
```

Попробуйте загрузить модуль в интерпретатор. `ghci` выдаст длинный список ошибок, это – хорошо. По ним вы сможете легко догадаться в каких местах необходимо заменить `Ord` а на `(Hashable a, Eq a)`.

Теперь для поиска маршрутов нам необходимо определить экземпляр класса `Hashable` для типа `Station`. В модуле `Data.Hashable` уже определены экземпляры для многих стандартных типов. Мы воспользуемся экземпляром для целых чисел.

Добавим в `driving` подчинённых типов класс `Enum` и воспользуемся им в экземпляре для `Hashable`:

```
instance Hashable Station where
    hash (St a b) = hash (fromEnum a, fromEnum b)
```

Теперь определим две функции определения маршрута:

```
import qualified AstarSet      as S
import qualified AstarHashSet as H
...

connectSet :: Station -> Station -> Maybe [Station]
connectSet a b = S.search (== b) $ metroTree a b

connectHashSet :: Station -> Station -> Maybe [Station]
connectHashSet a b = H.search (== b) $ metroTree a b
```

Как нам сравнить быстродействие двух алгоритмов? Оценка быстродействия программ, написанных на Haskell, может таить в себе подвохи. Например если мы запустим оба алгоритма в одной программе, возможно случится такая ситуация, что часть данных, одинаковая для каждого из методов будет вычислена один раз, а во втором алгоритме переиспользована, и нам может показаться, что второй алгоритм гораздо быстрее первого. Также необходимо учитывать внешние факторы. Тестовая программа вычисляется на одном компьютере, и если алгоритмы тестируются в разное время, может статься так, что мы сидели-сидели и ждали пока тест завершится, в это время работал первый алгоритм, потом нам надоело ждать, мы решили включить музыку, проверить почту, и второму алгоритму досталось меньше вычислительных ресурсов. Все эти факторы необходимо учитывать при тестировании. Как раз для этого и существует замечательная библиотека `criterion`.

Она проводит серию тестов и по ним оценивает показатели быстродействия. При этом учитывается достоверность тестов. По результатам тестирования показатели сверяются между собой, и если разброс оказывается слишком большим, программа сообщает нам: что-

то тут не чисто, данным не стоит доверять. Более того результаты оформляются в наглядные графики, мы можем на глаз оценить распределения и разброс показателей.

Основные типы criterion

Центральным элементом библиотеки является класс `Benchmarkable`. Он объединяет данные, которые можно тестировать. Среди них чистые функции (тип `Pure`) и значения с побочными эффектами (тип `IO a`).

Мы можем превращать данные в тесты (тип `Benchmark`) с помощью функции `bench`:

```
benchSource :: Benchmarkable b => String -> b -> Benchmark
```

Она добавляет к данным комментарий и превращает их в тесты. Как было отмечено, существует одна тонкость при тестировании чистых функций: чистые функции в `Haskell` могут разделять данные между собой, поэтому для независимого тестирования мы оборачиваем функции в специальный тип `Pure`. У нас есть два варианта тестирования:

Мы можем протестировать приведение результата к заголовочной нормальной форме (вспомните главу о ленивых вычислениях):

```
nf :: NFData b => (a -> b) -> a -> Pure
```

или к слабой заголовочной нормальной форме:

```
whnf :: (a -> b) -> a -> Pure
```

Аналогичные функции (`nfIO`, `whnfIO`) есть и для данных с побочными эффектами. Класс `NFData` обозначает все значения, для которых заголовочная нормальная форма определена. Этот класс пришёл в библиотеку `criterion` из библиотеки `deepseq`. Стоит отметить эту библиотеку. В ней определён аналог функции `seq`. Функция `seq` приводит значения к слабой заголовочной нормальной форме (мы заглядываем вглубь значения лишь на один конструктор), а функция `deepseq` проводит полное вычисление значения. Значение приводится к заголовочной нормальной форме.

Также нам пригодится функция группировки тестов:

```
bgroup :: String -> [Benchmark] -> Benchmark
```

С её помощью мы объединяем список тестов в один, под некоторым именем. Тестирование проводится с помощью функции `defaultMain`:

```
defaultMain :: [Benchmark] -> IO ()
```

Она принимает список тестов и выполняет их. Выполнение тестов заключается в компиляции программы. После компиляции мы получим исполняемый файл который проводит тестирование в зависимости от параметров, указываемых флагами. До них мы ещё доберёмся, а пока опишем наши тесты:

```
-- | Module: Speed.hs
module Main where

import Criterion.Main
import Control.DeepSeq

import Metro

instance NFData Station where
    rnf (St a b) = rnf (rnf a, rnf b)

instance NFData Way where
instance NFData Name where

pair1 = (St Orange DnoBolota, St Green Prizrak)
pair2 = (St Red Lao, St Blue De)

test name search = bgroup name $ [
    bench "1" $ nf (uncurry search) pair1,
    bench "2" $ nf (uncurry search) pair2]

main = defaultMain [
    test "Set" connectSet,
    test "Hash" connectHashSet]
```

Экземпляр для класса `NFData` похож на экземпляр для `Hashable`. Мы также определили метод значения через методы для типов, из которых он состоит. Класс `NFData` устроен так, что для типов из класса `Enum` мы можем воспользоваться определением по умолчанию (как в случае для `Way` и `Name`).

Теперь перейдём в командную строку, переключимся на директорию с нашим модулем и скомпилируем его:

```
$ ghc -O --make Speed.hs
```

Флаг **-O** говорит ghc, что необходимо провести оптимизацию кода. Появится исполняемый файл **Speed**. Что мы можем делать с этим файлом? Узнать это можно, запустив его с флагом **--help**:

Мы можем узнать какие функции нам доступны, набрав:

```
$ ./Speed --help
I don't know what version I am.
Usage: Speed [OPTIONS] [BENCHMARKS]
  -h, -?      --help          print help, then exit
  -G          --no-gc         do not collect garbage between iterations
  -g          --gc            collect garbage between iterations
  -I CI       --ci=CI         bootstrap confidence interval
  -l          --list          print only a list of benchmark names
  -o FILENAME --output=FILENAME report file to write to
  -q          --quiet         print less output
                  --resamples=N number of bootstrap resamples to perform
  -s N        --samples=N     number of samples to collect
  -t FILENAME --template=FILENAME template file to use
  -u FILENAME --summary=FILENAME produce a summary CSV file of all results
  -V          --version       display version, then exit
  -v          --verbose       print more output

If no benchmark names are given, all are run
Otherwise, benchmarks are run by prefix match
```

Из этих настроек самые интересные, это **-s** и **-o**. **-s** указывает число сэмплов выборке (столько раз будет запущен каждый тест). а **-o** говорит, о том в какой файл поместить результаты. Результаты представлены в виде графиков, формируется файл, который можно открыть в любом браузере. Записать данные в таблицу (например для отчёта) можно с помощью флага **-u**.

Проверим результаты:

```
./Speed -o res.html -s 100
```

Откроем файл **res.html** и посмотрим на графики. Оказалось, что для данных двух случаев первый алгоритм работал немного лучше. Но выборку из двух вариантов вряд ли можно считать убедительной. Давайте расширим выборку с помощью **QuickCheck**. Мы запустим проверку какого-нибудь свойства тем и другим методом. В итоге **QuickCheck** сам сгенерирует достаточное число случайных данных, а

criterion оценит быстродействие. Мы проверим самое первое свойство (о перевёрнутых маршрутах) на том и другом алгоритме.

```
module Main where

import Control.Applicative

import Test.QuickCheck
import Metro

instance Arbitrary Station where
    arbitrary = ($ s0) . foldr (.) id . fmap select <$> ints
    where ints = vector =<< choose (0, 100)
          s0 = St Blue De

select :: Int -> Station -> Station
select i s = as !! mod i (length as)
    where as = fst <$> distMetroMap s

prop :: (Station -> Station -> Maybe [Station])
    -> Station -> Station -> Bool
prop search a b = search a b == (reverse <$> search b a)

main = defaultMain [
    bench "Set" $ quickCheck (prop connectSet),
    bench "Hash" $ quickCheck (prop connectHashSet)]
```

В этом тесте метод **Set** также оказался совсем немного быстрее.

Как интерпретировать результаты? С левой стороны мы видим оценку плотности вероятности распределения быстродействия. Под графиком мы видим среднее (mean) и дисперсию значения (std dev). Показаны три числа. Это нижняя грань доверительного интервала, оценка величины и верхняя грань доверительного интервала (ci, confidence interval). Среднее значение показывает оценку величины, мы говорим, что алгоритм работает примерно 100 миллисекунд. Дисперсия – это разброс результатов вокруг среднего значения. С правой стороны мы видим графики с точками. Каждая точка обозначает отдельный запуск алгоритма. Количество запусков соответствует флагу **-s**. В последней строке под графиком criterion сообщает степень недоверия к результатам. В последнем опыте этот показатель достаточно высок. Возможно это связано с тем, что наш алгоритм выбора случайных станций имеет сильный разброс по времени. Ведь сначала мы генерируем случайное число n от 0 до 100, и затем начинаем блуждать по карте от начальной точке n раз.

Также может влиять то, что время работы алгоритма зависит от положения станций.

Краткое содержание

В этой главе мы реализовали алгоритм эвристического поиска A^* . Также мы узнали несколько стандартных структур данных. Это множества и очереди с приоритетом и освежили в памяти ленивые вычисления.

Мы научились проверять свойства программ (**QuickCheck**), а также оценивать быстродействие программ (**criterion**).

Упражнения

- Я говорил о том, что два варианта алгоритмов дают одинаковые результаты, но так ли это на самом деле? Проверьте это в **QuickCheck**.
- Алгоритм эвристического поиска может применяться не только для поиска маршрутов на карте. Часто алгоритм A^* применяется в играх. Встройте этот алгоритм в игру пятнашки (глава 13). Если игрок запутался и не знает как ходить, он может попросить у компьютера совет. В этой задаче альтернативы~– это вершины графа, соседние вершины~– это те вершины, в которые мы можем попасть за один ход.
Подсказка: воспользуйтесь манхэттенским расстоянием.
- Оцените эффективность двух алгоритмов поиска в игре пятнашки. Рассмотрите зависимость быстродействия от степени сложности игры.

Императивное программирование

В этой главе мы потренируемся в укрощении императивного кода. В Haskell все побочные эффекты огорожены от чистых функций бетонной стеной IO. Однажды оступившись, мы не можем свернуть с пути побочных эффектов, мы вынуждены тащить на себе груз IO до самого конца программы. Тип IO, хоть и обволакивает программу, всё же позволяет пользоваться благами чистых вычислений. От программиста зависит насколько сильна будет хватка IO. Необходимо уметь выделять точки, в которых применение побочных вычислений действительно необходимо, подключая в них чистые функции через методы классов Functor, Applicative и Monad. Тип IO похож на дорогу с контрольными пунктами, в которых необходимо отчитаться перед компилятором за “грязный код”. При неумелом проектировании написание программ, насыщенных побочными эффектами, может превратиться в пытку. Контрольные пункты будут встречаться в каждой функции.

Естественный источник побочных эффектов – это пользователь программы. Но, к сожалению, это не единственный источник. Haskell – открытый язык программирования. В нём можно пользоваться программами из низкоуровневого языка C. Основное преимущество C заключается в непревзойдённой скорости программ. Этот язык позволяет программисту работать с памятью компьютера напрямую. Но за эту силу приходится платить. Возможны очень неприятные и трудноуловимые ошибки. Утечки памяти, обращение по неверному адресу в памяти, неожиданное обновление переменных. Ещё один плюс C в том, что это язык с историей, на нём написано много хороших библиотек. Некоторые из них встроены в Haskell с помощью специального механизма FFI (foreign function interface). Обсуждение того, как устроен FFI выходит за рамки этой книги. Интересующийся читатель может обратиться к книге *Real World Haskell*. Мы же потренируемся в использовании таких библиотек. Язык C является императивным, поэтому, применяя его функций в Haskell, мы неизбежно сталкиваемся с типом IO, поскольку большинство интересных функций в C изменяют состояние своих

аргументов. В C пишут и чистые функции, такие функции переносятся в Haskell без потери чистоты, но это не всегда возможно.

В этой главе мы напишем небольшую 2D-игру, подключив две FFI-библиотеки, это графическая библиотека **OpenGL** и физический движок **Chipmunk**.

Описание игры

Игра происходит на бильярдной доске. Игрок управляет красным шаром, кликнув в любую точку экрана, он может изменить направление вектора скорости красного шара. Шар покатится туда, куда кликнул пользователь в последний раз. Из луз будут вылетать шары трёх типов: синие, зелёные и оранжевые. Столкновение красного шара с синим означает минус одну жизнь, с зелёным – плюс одну жизнь, оранжевый шар означает бонус. Если шар игрока сталкивается с оранжевым шаром все шары в определённом радиусе от места столкновения исчезают и записываются в бонусные очки, за каждый шар по одному очку, при этом шар с которым произошло столкновение не считается. Все столкновения – абсолютно упругие, поэтому при столкновении энергия сохраняется и шары никогда не останавливаются. Если шар попадает в лузу, то он исчезает. Если в лузу попал шар игрока – это означает, что игра окончена. Игрок стартует с несколькими жизнями, когда их число подходит к нулю игра останавливается. После столкновения с зелёным шаром, шар пропадает, а после столкновения с синим – нет. В итоге все против игрока, кроме зелёных и оранжевых шаров.

Основные библиотеки

Контролировать физику игрового мира будет библиотека **Chipmunk**, а библиотека **OpenGL** будет рисовать (конечно если мы её этому научим). Пришло время с ними познакомиться.

Изменяемые значения

Перед тем как мы перейдём к библиотекам нам нужно узнать ещё кое-что. В Haskell мы не можем изменять значения. Но в C это делается постоянно, а соответственно и в библиотеках написанных на C тоже.

Для того чтобы имитировать в Haskell механизм обновления значений были придуманы специальные типы. Мы можем объявить изменяемое значение и обновлять его, но только в пределах типа `IO`.

IORef

Тип `IORef` из модуля `Data.IORef` описывает изменяемые значения:

```
newIORef :: a -> IO IORef

readIORef  :: IORef a -> IO a
writeIORef :: IORef a -> a -> IO ()
modifyIORef :: IORef a -> (a -> a) -> IO ()
```

Функция `newIORef` создаёт изменяемое значение и инициализирует его некоторым значением, которые мы можем считать с помощью функции `readIORef` или обновить с помощью функций `writeIORef` или `modifyIORef`. Посмотрим как это работает:

```
module Main where

import Data.IORef

main = var >>= (\v ->
    readIORef v >>= print
  >> writeIORef v 4
  >> readIORef v >>= print)
  where var = newIORef 2
```

Теперь посмотрим на ответ `ghci`:

```
*Main> :l HelloIORef
[1 of 1] Compiling Main                ( HelloIORef.hs, interpreted )
Ok, modules loaded: Main.
*Main> main
2
4
```

Самое время вернуться к главе 17 и вспомнить о `do`-нотации. Такой императивный код гораздо нагляднее писать так:

```
main = do
  var <- newIORef 2
  x <- readIORef var
  print x
  writeIORef var 4
  x <- readIORef var
  print x
```

Эта запись выглядит как последовательность действий. Не правда ли очень похоже на обычный императивный язык. Такие переменные встречаются очень часто в библиотеках, заимствованных из C.

StateVar

В модуле `Data.StateVar` определены типы, которые накладывают ограничение на права по чтению и записи. Мы можем определять переменные доступные только для чтения (`GettableStateVar a`), только для записи (`SettableStateVar a`) или обычные изменяемые переменные (`SetVar a`).

Операции чтения и записи описываются с помощью классов:

```
class HasGetter s where
  get :: s a -> IO a

class HasSetter s where
  ($=) :: s a -> a -> IO ()
```

Тип `IORef` принадлежит и тому, и другому классу:

```
main = do
  var <- newIORef 2
  x    <- get var
  print x
  var $= 4
  x    <- get var
  print x
```

OpenGL

`OpenGL` является ярким примером библиотеки построенной на изменяемых переменных. `OpenGL` можно представить как большой конечный автомат. Каждая строка кода – это запрос на изменение состояния. Причём этот автомат является глобальной переменной. Его текущее состояние зависит от всей цепочки предыдущих команд. Параметры рисования задаются глобальными переменными (тип `StateVar`).

`OpenGL` не зависит от конкретной оконной системы, она отвечает лишь за рисование. Для того чтобы создать окно и перехватывать в нём действия пользователя нам понадобится отдельная библиотека. Для этого мы воспользуемся `GLFW`, это библиотека также пришла в

Haskell из C. Интерфейсы **GLFW** и **OpenGL** очень похожи. Мы будем обновлять различные параметры библиотеки с помощью типа **StateVar**. Давайте создадим окно и закрасим фон белым цветом:

```
module Main where

import Graphics.UI.GLFW
import Graphics.Rendering.OpenGL
import System.Exit

title = "Hello OpenGL"

width  = 700
height = 600

main = do
    initialize
    openWindow (Size width height) [] Window
    windowTitle $= title

    clearColor $= Color4 1 1 1 1

    windowCloseCallback $= exitWith ExitSuccess
    loop

loop = do
    display
    loop

display = do
    clear [ColorBuffer]
    swapBuffers
```

Мы инициализируем **GLFW**, задаём параметры окна. Устанавливаем цвет фона. Цвет имеет четыре параметра это RGB-цвета и параметр прозрачности. Затем мы говорим, что программе делать при закрытии окна. Мы устанавливаем функцию обратного вызова (callback) `windowCloseCallback`. В самом конце мы входим в цикл, который только и делает, что стирает окно цветом фона и делает рабочий буфер видимым. Что такое буфер? Буфер – это место в котором мы рисуем. У нас есть два буфера. Один мы показываем пользователю, а в другом в это в время рисуем, когда приходит время обновлять картинку мы просто меняем их местами командой `swapBuffers`.

Посмотрим, что у нас получилось:

```
$ ghc --make HelloOpenGL.hs
```

```
$ ./HelloOpenGL
```

Нарисуем упрощённое начальное положение нашей игры: прямоугольную рамку и в ней – красный шар:

```
module Main where

import Graphics.UI.GLFW
import Graphics.Rendering.OpenGL

import System.Exit

title = "Hello OpenGL"

width, height :: GLsizei

width  = 700
height = 600

w2, h2 :: GLfloat

w2 = (fromIntegral $ width) / 2
h2 = (fromIntegral $ height) / 2

dw2, dh2 :: GLdouble

dw2 = fromRational $ toRational w2
dh2 = fromRational $ toRational h2

main = do
  initialize
  openWindow (Size width height) [] Window
  windowTitle $= title

  clearColor $= Color4 1 1 1 1
  ortho (-dw2-50) (dw2+50) (-dh2-50) (dh2+50) (-1) 1

  windowCloseCallback $= exitWith ExitSuccess
  windowSizeCallback  $= (\size -> viewport $= (Position 0 0, size))

  loop

loop = do
  display
  loop

display = do
  clear [ColorBuffer]

  color black
  line (-w2) (-h2) (-w2) h2
  line (-w2) h2      w2    h2
```

```

line w2    h2    w2    (-h2)
line w2    (-h2) (-w2) (-h2)

color red
circle 0 0 10

swapBuffers

```

```

vertex2f :: GLfloat -> GLfloat -> IO ()
vertex2f a b = vertex (Vertex3 a b 0)

-- colors

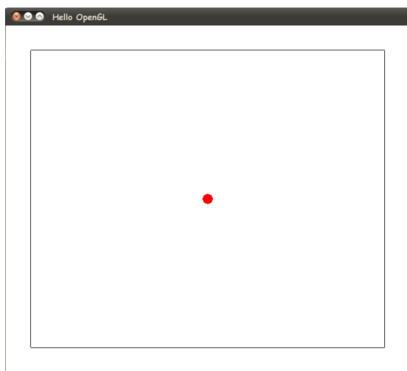
white = Color4 (0::GLfloat)
black = Color4 (0::GLfloat) 0 0 1
red   = Color4 (1::GLfloat) 0 0 1

-- primitives

line :: GLfloat -> GLfloat -> GLfloat -> GLfloat -> IO ()
line ax ay bx by = renderPrimitive Lines $ do
    vertex2f ax ay
    vertex2f bx by

circle :: GLfloat -> GLfloat -> GLfloat -> IO ()
circle cx cy rad =
    renderPrimitive Polygon $ mapM_ (uncurry vertex2f) points
    where n = 50
          points = zip xs ys
          xs = fmap (\x -> cx + rad * sin (2*pi*x/n)) [0 .. n]
          ys = fmap (\x -> cy + rad * cos (2*pi*x/n)) [0 .. n]

```



Начальное положение

Мы рисуем с помощью функции `renderPrimitive`. Она принимает метку элемента, который мы собираемся рисовать и набор вершин. Так метка `Lines` обозначает линии, а метка `Polygon` – закрашенные

многоугольники. В OpenGL нет специальной операции для рисования окружностей, поэтому нам придётся представить окружность в виде многоугольника (circle). Функция `ortho` устанавливает область видимости рисунка, шесть аргументов функции обозначают пары диапазонов по каждой из трёх координат. При этом вершины передаются не списком а в специальном `do`-блоке. За счёт этого мы можем изменить какие-нибудь параметры OpenGL во время рисования. Обратите внимание на то, как мы изменяем цвет примитива. Перед тем как рисовать примитив мы устанавливаем значение цвета (`color`).

Анимация

Оживим нашу картинку. При клике мышкой шарик игрока последует в направлении курсора. Для того чтобы картинка задвигалась нам необходимо обновлять рисунок с определённой частотой. Мы будем регулировать частоту обновления с помощью функции `sleep`, с её помощью мы можем задержать выполнение программы (время измеряется в секундах):

```
sleep :: Double -> IO ()
```

За перехват действий пользователя отвечает функции:

```
getMouseButton :: MouseButton -> IO KeyButtonState
mousePos        :: StateVar Position
```

Функция `getMouseButton` сообщает текущее состояние кнопок мыши, мы будем перехватывать положение мыши во время нажатия левой кнопки:

```
onMouse ball = do
  mb <- getMouseButton ButtonLeft
  when (mb == Press) (get mousePos >= updateVel ball)
```

Стандартная функция `when` из модуля `Control.Monad` выполняет действие только в том случае, если первый аргумент равен `True`. Для обновления положения и направления скорости шарика нам придётся воспользоваться глобальной переменной типа `IORef Ball`:

```
data Ball = Ball
  { ballPos :: Vec2d
  , ballVel :: Vec2d
  }
```


Код программы:

```
module Main where

import Control.Applicative
import Data.IORef
import Graphics.UI.GLFW
import Graphics.Rendering.OpenGL
import System.Exit
import Control.Monad

type Time = Double

title = "Hello OpenGL"

width, height :: GLsizei

fps :: Int
fps = 60

frameTime :: Time
frameTime = 1000 * ((1::Double) / fromIntegral fps)

width  = 700
height = 600

w2, h2 :: GLfloat

w2 = (fromIntegral $ width) / 2
h2 = (fromIntegral $ height) / 2

dw2, dh2 :: GLdouble

dw2 = fromRational $ toRational w2
dh2 = fromRational $ toRational h2

type Vec2d = (GLfloat, GLfloat)

data Ball = Ball
  { ballPos :: Vec2d
  , ballVel :: Vec2d
  }

initBall = Ball (0, 0) (0, 0)

dt :: GLfloat
dt = 0.3

minVel = 10

main = do
  initialize
  openWindow (Size width height) [] Window
```

```

windowTitle $= title

clearColor $= Color4 1 1 1 1
ortho (-dw2) (dw2) (-dh2) (dh2) (-1) 1

ball <- newIORef initBall

windowCloseCallback $= exitWith ExitSuccess
windowSizeCallback $= (\size -> viewport $= (Position 0 0, size))

loop ball

loop :: IORef Ball -> IO ()
loop ball = do
  display ball
  onMouse ball
  sleep frameTime
  loop ball

display ball = do
  (px, py) <- ballPos <$> get ball
  (vx, vy) <- ballVel <$> get ball
  ball $= Ball (px + dt*vx, py + dt*vy) (vx, vy)

  clear [ColorBuffer]

  color black
  line (-ow2) (-oh2) (-ow2) oh2
  line (-ow2) oh2 ow2 oh2
  line ow2 oh2 ow2 (-oh2)
  line ow2 (-oh2) (-ow2) (-oh2)

  color red
  circle px py 10

  swapBuffers
  where ow2 = w2 - 50
        oh2 = h2 - 50

onMouse ball = do
  mb <- getMouseButton ButtonLeft
  when (mb == Press) (get mousePos >=> updateVel ball)

updateVel ball pos = do
  (p0x, p0y) <- ballPos <$> get ball
  v0 <- ballVel <$> get ball
  size <- get windowSize
  let (p1x, p1y) = mouse2canvas size pos
      v1 = scaleV (max minVel $ len v0) $ norm (p1x - p0x, p1y - p0y)
  ball $= Ball (p0x, p0y) v1
  where norm v@(x, y) = (x / len v, y / len v)
        len (x, y) = sqrt (x*x + y*y)
        scaleV k (x, y) = (k*x, k*y)

```

```

mouse2canvas :: Size -> Position -> (GLfloat, GLfloat)
mouse2canvas (Size sx sy) (Position mx my) = (x, y)
  where d a b = fromIntegral a / fromIntegral b
        x = fromIntegral width * (d mx sx - 0.5)
        y = fromIntegral height * (negate $ d my sy - 0.5)

vertex2f :: GLfloat -> GLfloat -> IO ()
vertex2f a b = vertex (Vertex3 a b 0)

-- colors
... white, black, red

-- primitives
line    :: GLfloat -> GLfloat -> GLfloat -> GLfloat -> IO ()
circle  :: GLfloat -> GLfloat -> GLfloat -> IO ()

```

Теперь функция `display` принимает ссылку на глобальную переменную, которая отвечает за движение шарика. Функция `mouse2canvas` переводит координаты в окне `GLFW` в координаты `OpenGL`. В `GLFW` начало координат лежит в левом верхнем углу окна и ось `Oy` направлена вниз. Мы же переместили начало координат в центр окна и ось `Oy` направлена вверх.

Посмотрим что у нас получилось:

```

$ ghc --make Animation.hs
$ ./Animation

```

Chipmunk

Картинка ожила, но шарик движется не реалистично. Он проходит сквозь стены. Добавим в нашу программу немного физики.

Воспользуемся библиотекой `Hipmunk`

```
cabal install Hipmunk
```

Она даёт возможность вызывать из Haskell функции C-библиотеки `Chipmunk`. Эта библиотека позволяет строить двухмерные физические модели. Основным элементом модели является пространство (`Space`). К нему мы можем добавлять различные объекты. Объект состоит из двух компонент: тела (`Body`) и формы (`Shape`). Тело отвечает за такие физические характеристики как масса, момент инерции, восприимчивость к силам. По форме определяются моменты столкновения тел. Форма может состоять из нескольких примитивов:

окружностей, линий и выпуклых многоугольников. Также мы можем добавлять различные ограничения (**Constraint**) они имитируют пружинки, шарниры. Мы можем назначать выполнение **IO**-действий на столкновения.

Опишем в **Hipmunk** модель шарика бегущего в замкнутой коробке:

```
module Main where
import Data.StateVar
import Physics.Hipmunk

main = do
  initChipmunk
  space <- newSpace
  initWalls space
  ball <- initBall space initPos initVel
  loop 100 space ball

loop :: Int -> Space -> Body -> IO ()
loop 0 _ _ = return ()
loop n space ball = do
  showPosition ball
  step space 0.5
  loop (n-1) space ball

showPosition :: Body -> IO ()
showPosition ball = do
  pos <- get $ position ball
  print pos

initWalls :: Space -> IO ()
initWalls space = mapM_ (uncurry $ initWall space) wallPoints

initWall :: Space -> Position -> Position -> IO ()
initWall space a b = do
  body <- newBody infinity infinity
  shape <- newShape body (LineSegment a b wallThickness) 0
  elasticity shape $= nearOne
  spaceAdd space body
  spaceAdd space shape

initBall :: Space -> Position -> Velocity -> IO Body
initBall space pos vel = do
  body <- newBody ballMass ballMoment
  shape <- newShape body (Circle ballRadius) 0
  position body $= pos
  velocity body $= vel
  elasticity shape $= nearOne
  spaceAdd space body
  spaceAdd space shape
  return body
```

```

-----
-- inits

nearOne = 0.9999
ballMass = 20
ballMoment = momentForCircle ballMass (0, ballRadius) 0
ballRadius = 10

initPos = Vector 0 0
initVel = Vector 10 5

wallThickness = 1

wallPoints = fmap (uncurry f) [
    ((-w2, -h2), (-w2, h2)),
    ((-w2, h2), (w2, h2)),
    ((w2, h2), (w2, -h2)),
    ((w2, -h2), (-w2, -h2))]
  where f a b = (g a, g b)
        g (a, b) = H.Vector a b

h2 = 100
w2 = 100

```

Функция `initChipmunk` инициализирует библиотеку `Chipmunk`. Она должна быть вызвана один раз до любой из функций библиотеки `Hipmunk`. Функции `new[Body|Shape|Space]` создают объекты модели. Мы сделали стены неподвижными, присвоив им бесконечную массу и момент инерции (`initWall`). Упругость удара определяется переменной `elasticity`, она не может быть больше единицы. Единица обозначает абсолютно упругое столкновение. В документации к `Hipmunk` не рекомендуют присваивать значение равное единице из-за возможных погрешностей округления, поэтому мы выбираем число близкое к единице. После инициализации элементов модели мы запускаем цикл, в котором происходит обновление модели (`step`) и печать положения шарика. Обратите внимание на то, что координаты шарика никогда не выйдут за установленные рамки.

Теперь объединим OpenGL и Hipmunk:

```

module Main where

import Control.Applicative

import Control.Applicative
import Data.StateVar
import Data.IORef

```

```

import Graphics.UI.GLFW
import System.Exit
import Control.Monad

import qualified Physics.Hipmunk as H
import qualified Graphics.UI.GLFW as G
import qualified Graphics.Rendering.OpenGL as G

title = "in the box"

-----
-- inits

type Time = Double

-- frames per second
fps :: Int
fps = 60

-- frame time in milliseconds
frameTime :: Time
frameTime = 1000 * ((1::Double) / fromIntegral fps)

nearOne = 0.9999
ballMass = 20
ballMoment = H.momentForCircle ballMass (0, ballRadius) 0
ballRadius = 10

initPos = H.Vector 0 0
initVel = H.Vector 0 0

wallThickness = 1

wallPoints = fmap (uncurry f) [
    ((-ow2, -oh2), (-ow2, oh2)),
    ((-ow2, oh2), (ow2, oh2)),
    ((ow2, oh2), (ow2, -oh2)),
    ((ow2, -oh2), (-ow2, -oh2))]
    where f a b = (g a, g b)
          g (a, b) = H.Vector a b

dt :: Double
dt = 0.5

minVel :: Double
minVel = 10

width, height :: Double

height = 500
width = 700

```

```
w2, h2 :: Double
```

```
h2 = height / 2
```

```
w2 = width / 2
```

```
ow2, oh2 :: Double
```

```
ow2 = w2 - 50
```

```
oh2 = h2 - 50
```

```
data State = State
```

```
{ stateBall    :: H.Body  
  , stateSpace  :: H.Space  
}
```

```
ballPos :: State -> StateVar H.Position
```

```
ballPos = H.position . stateBall
```

```
ballVel :: State -> StateVar H.Velocity
```

```
ballVel = H.velocity . stateBall
```

```
main = do
```

```
  H.initChipmunk
```

```
  initGLFW
```

```
  state <- newIORef =<< initState
```

```
  loop state
```

```
loop :: IORef State -> IO ()
```

```
loop state = do
```

```
  display state
```

```
  onMouse state
```

```
  sleep frameTime
```

```
  loop state
```

```
simulate :: State -> IO Time
```

```
simulate a = do
```

```
  t0 <- get G.time
```

```
  H.step (stateSpace a) dt
```

```
  t1 <- get G.time
```

```
  return (t1 - t0)
```

```
initGLFW :: IO ()
```

```
initGLFW = do
```

```
  G.initialize
```

```
  G.openWindow (G.Size (d2gli width) (d2gli height)) [] G.Window
```

```
  G.windowTitle $= title
```

```
  G.windowCloseCallback $= exitWith ExitSuccess
```

```
  G.windowSizeCallback $= (\size -> G.viewport $= (G.Position 0 0, size))
```

```
  G.clearColor $= G.Color4 1 1 1 1
```

```
  G.ortho (-dw2) (dw2) (-dh2) (dh2) (-1) 1
```

```
  where dw2 = realToFrac w2
```

```
        dh2 = realToFrac h2
```

```

initState :: IO State
initState = do
    space <- H.newSpace
    initWalls space
    ball <- initBall space initPos initVel
    return $ State ball space

initWalls :: H.Space -> IO ()
initWalls space = mapM_ (uncurry $ initWall space) wallPoints

initWall :: H.Space -> H.Position -> H.Position -> IO ()
initWall space a b = do
    body <- H.newBody H.infinity H.infinity
    shape <- H.newShape body (H.LineSegment a b wallThickness) 0
    H.elasticity shape $= nearOne
    H.spaceAdd space body
    H.spaceAdd space shape

initBall :: H.Space -> H.Position -> H.Velocity -> IO H.Body
initBall space pos vel = do
    body <- H.newBody ballMass ballMoment
    shape <- H.newShape body (H.Circle ballRadius) 0
    H.position body $= pos
    H.velocity body $= vel
    H.elasticity shape $= nearOne
    H.spaceAdd space body
    H.spaceAdd space shape
    return body

-----
-- graphics

display state = do
    drawState =<< get state
    simTime <- simulate =<< get state
    sleep (max 0 $ frameTime - simTime)

drawState :: State -> IO ()
drawState st = do
    pos <- get $ ballPos st
    G.clear [G.ColorBuffer]
    drawWalls
    drawBall pos
    G.swapBuffers

drawBall :: H.Position -> IO ()
drawBall pos = do
    G.color red
    circle x y $ d2gl ballRadius
    where (x, y) = vec2gl pos

```



```

drawWalls :: IO ()
drawWalls = do
    G.color black
    line (-dow2) (-doh2) (-dow2) dow2
    line (-dow2) dow2      dow2      doh2
    line dow2      doh2      dow2      (-doh2)
    line dow2      (-doh2) (-dow2) (-doh2)
    where dow2 = d2gl ow2
          doh2 = d2gl oh2

onMouse state = do
    mb <- G.getMouseButton ButtonLeft
    when (mb == Press) (get G.mousePosition >= updateVel state)

updateVel state pos = do
    size <- get G.windowSize
    st <- get state
    p0 <- get $ ballPos st
    v0 <- get $ ballVel st
    let p1 = mouse2canvas size pos
    ballVel st $=
        H.scale (H.normalize $ p1 - p0) (max minVel $ H.len v0)

mouse2canvas :: G.Size -> G.Position -> H.Vector
mouse2canvas (G.Size sx sy) (G.Position mx my) = H.Vector x y
    where d a b = fromIntegral a / fromIntegral b
          x = width * (d mx sx - 0.5)
          y = height * (negate $ d my sy - 0.5)

vertex2f :: G.GLfloat -> G.GLfloat -> IO ()
vertex2f a b = G.vertex (G.Vertex3 a b 0)

vec2gl :: H.Vector -> (G.GLfloat, G.GLfloat)
vec2gl (H.Vector x y) = (d2gl x, d2gl y)

d2gl :: Double -> G.GLfloat
d2gl = realToFrac

d2gli :: Double -> G.GLsizei
d2gli = toEnum . fromEnum . d2gl

...

```

Функции не претерпевшие особых изменений пропущены. Теперь наше глобальное состояние (`State`) содержит тело шара (оно пригодится нам для вычисления его положения) и пространство, в котором живёт наша модель. Стоит отметить функцию `simulate`. В ней происходит обновление состояния модели. При этом мы возвращаем время,

которое ушло на вычисление этой функции. Оно нужно нам для того, чтобы показывать новые кадры равномерно. Мы вычтем время симуляции из общего времени, которое мы можем потратить на один кадр (`frameTime`).

Боремся с IO

Кажется, что мы попали в какой-то другой язык. Это совсем не тот элегантный Haskell, знакомый нам по предыдущим главам. Столько `do` и `IO` разбросано по всему коду. И такой примитивный результат в итоге. Если так будет продолжаться и дальше, то мы можем не вытерпеть и бросить и нашу задачу и Haskell...

Не отчаивайтесь!

Давайте лучше подумаем как свести этот псевдо-Haskell к минимуму. Подумаем какие источники `IO` точно будут в нашей программе. Это инициализация `GLFW` и `Hipmunk`, клики мышью, обновление модели в `Hipmunk`, также для рисования нам придётся считывать положения шаров. Нам придётся удалять и создавать новые шары, добавляя их к пространству модели. Также в `IO` происходит отрисовка игры. `Hipmunk` будет контролировать столкновения шаров, и эти данные нам тоже надо будет считывать из глобальных переменных. Сколько всего! Голова идёт кругом.

Но помимо всего этого у нас есть логика игры. Логика игры отвечает за реакцию игрового мира на различные события. Например столкновение с “плохим” шаром влечёт к уменьшению жизней, если игрок сталкивается с бонусным шаром, определённые шары необходимо удалить. Приходит момент и мы выпускаем новый шар из лузы новый шар. Давайте подумаем как сохранить логику игры в чистоте.

Тип `IO` обычно отвечает за связь с внешним миром, это глаза, уши, руки и ноги программы. Через `IO` мы получаем информацию из внешнего мира и отправляем её обратно. Но в нашем случае он проник в сердце программы. За обновление объектов отвечает насыщенная `IO` библиотека `Hipmunk`.

Мы постараемся побороться с `IO`-кодом так. Сначала мы выделим те параметры, которые могут быть обновлены чистыми функциями. Это

все те параметры, для которых не нужен `Hipmunk`. Этот шаг разбивает наш мир на два лагеря: “чистый” и “грязный”:

```
data World = World
  { worldPure    :: Pure
  , worldDirty   :: Dirty }
```

Чистые данные хотят как-то узнать о том, что происходит в грязных данных. Также чистые данные могут рассказать грязным, как им нужно измениться. Это приводит нас к определению двух языков запросов, на которых чистый и грязный мир общаются между собой:

```
data Query = Remove Ball | HeroVelocity H.Velocity | MakeBall Freq
```

```
data Event = Touch Ball | UserClick H.Position
```

```
data Sense = Sense
  { senseHero      :: HeroBall
  , senseBalls     :: [Ball] }
```

Через `Query` чистые данные могут рассказать грязным о том, что необходимо удалить шар из игры, обновить скорость шара игрока или создать новый шар (`Freq` отвечает за параметры создания шара). Грязные данные могут рассказать чистым на языке `Event` и `Sense` о том, что один из шаров коснулся до шара игрока, или игрок кликнул мышкой в определённой точке. Также мы сообщаем все обновлённые положения параметры шаров в типе `Sense`. Тип `Event` отвечает за события, которые происходят иногда, а тип `Sense` за те параметры, которые мы наблюдаем непрерывно (это типы глазорук), `Query` – это язык действий (это тип руконог). Нам понадобится ещё один маленький язык, на котором мы будем объясняться с `OpenGL`.

```
data Picture = Prim Color Primitive
             | Join Picture Picture
```

```
data Primitive = Line Point Point | Circle Point Radius
```

```
data Point = Point Double Double
type Radius = Double
```

```
data Color = Color Double Double Double
```

Эти три языка станут барьером, которым мы ограничим влияние `IO`. У нас будут функции:

```

percept      :: Dirty -> IO (Sense, [Event])
updatePure   :: Sense -> [Event] -> Pure -> (Pure, [Query])
react        :: [Query] -> Dirty -> IO Dirty
updateDirty  :: Dirty -> IO Dirty
picture      :: Pure -> Picture
draw         :: Picture -> IO ()

```

Вся логика игры будет происходить в чистой функции `updatePure`, обновлять модель мира мы будем в `updateDirty`. Давайте опять начнём проектирование сверху-вниз. С этими функциями мы уже можем написать основную функцию цикла игры:

```

loop :: IORef World -> IO ()
loop worldRef = do
    world <- get worldRef
    drawWorld world
    (world, dt) <- updateWorld world
    worldRef $= world
    G.addTimerCallback (max 0 $ frameTime - dt) $ loop worldRef

```

```

updateWorld :: World -> IO (World, Time)
updateWorld world = do
    t0 <- get G.elapsedTime
    (sense, events) <- percept dirty
    let (pure', queries) = updatePure sense events pure
    dirty' <- updateDirty =<< react queries dirty
    t1 <- get G.elapsedTime
    return (World pure' dirty', t1 - t0)
    where dirty = worldDirty world
          pure  = worldPure  world

```

```

drawWorld :: World -> IO ()
drawWorld = draw . picture . worldPure

```

Определяемся с типами

Давайте подумаем, из чего состоят типы `Dirty` и `Pure`. Начнём с `Pure`. Там точно будет вся информация необходимая нам для рисования картинки (ведь функция `picture` определена на `Pure`). Для рисования нам необходимо знать положения всех шаров и их типы (они определяют цвет). На картинке мы будем показывать разную статистику (данные о жизнях, бонусные очки). Также из типа `Pure` мы будем управлять созданием шаров. Так мы приходим к типу:

```
data Pure = Pure
{ pureScores    :: Scores
, pureHero      :: HeroBall
, pureBalls     :: [Ball]
, pureStat      :: Stat
, pureCreation   :: Creation
}
```

Что нам нужно знать о шаре героя? Нам нужно его положение для отрисовки и модуль вектора скорости (он понадобится нам при обновлении вектора скорости шара игрока):

```
data HeroBall = HeroBall
{ heroPos      :: H.Position
, heroVel      :: H.CpFloat
}
```

Для остальных шаров нам нужно знать только тип шара, его положение и идентификатор шара. По идентификатору потом мы сможем понять какой шар удалить из грязных данных:

```
data Ball = Ball
{ ballType     :: BallType
, ballPos      :: H.Position
, ballId       :: Id
}
```

```
data BallType = Hero | Good | Bad | Bonus
  deriving (Show, Eq, Enum)
```

```
type Id = Int
```

Статистика игры состоит из числа жизней и бонусных очков:

```
data Scores = Scores
{ scoresLives  :: Int
, scoresBonus  :: Int
}
```

Как будет происходить создание новых шаров? Если плохих шаров будет слишком много, то играть будет не интересно, игрок слишком быстро проиграет. Если хороших шаров будет слишком много, то игроку также быстро надоест. Будет очень легко. Нам необходимо поддерживать определённый баланс шаров. Создание шаров будет происходить случайным образом через равные промежутки времени, но создание нового шара будет зависеть от пропорции шаров на доске в данный момент. Если у нас слишком много плохих шаров, то скорее

всего мы создадим хороший шар и наоборот. Если общее число шаров велико, то мы не будем усложнять игроку жизнь новыми шарами, дождёмся пока какие-нибудь шары не покинут пределы поля или не будут уничтожены игроком. Эти рассуждения приводят нас к типам:

```
data Creation = Creation
  { creationStat      :: Stat
  , creationGoalStat  :: Stat
  , creationTick      :: Int
  }

data Stat = Stat
  { goodCount      :: Int
  , badCount       :: Int
  , bonusCount     :: Int
  }

data Freq = Freq
  { freqGood       :: Float
  , freqBad        :: Float
  , freqBonus      :: Float
  }
```

Поле `creationStat` содержит текущее число шаров на поле, поле `creationGoalStat` – число шаров, к которому мы стремимся. Значение типа `Freq` содержит веса вероятностей создания нового шара определённого типа. На каждом шаге мы будем прибавлять единицу к `creationTick`, как только оно достигнет определённого значения мы попробуем создать новый шар.

Перейдём к грязным данным. Там мы будем хранить информацию, необходимую для обновления модели в `Hipmunk`, и значение, в которое `GLFW` будет записывать состояние мыши, также мы будем следить за тем, кто столкнулся с шаром игрока в данный момент:

```
data Dirty = Dirty
  { dirtyHero      :: Obj
  , dirtyObjs      :: IxMap Obj
  , dirtySpace     :: H.Space
  , dirtyTouchVar  :: Sensor H.Shape
  , dirtyMouse     :: Sensor H.Position
  }

data Obj = Obj
  { objType      :: BallType
  , objShape     :: H.Shape
  , objBody      :: H.Body
  }
```

```
type Sensor a = IORef (Maybe a)
```

Особая структура `IxMap` отвечает за хранение значений вместе с индексами. Пока остановимся на самом простом представлении:

```
type IxMap a = [(Id, a)]
```

Структура проекта

Наметим структуру проекта. У нас уже есть модуль `Types.hs`. Основной цикл игры будет описан в модуле `Loop.hs`. Общие функции обновления состояния будут определены в `World.hs`, также у нас будет два модуля отвечающие за обновление чистых и грязных данных – `Pure.hs` и `Dirty.hs`. Мы выделим отдельный модуль для описания всех констант игры (`Inits.hs`). Так нам будет удобно настроить игру, когда мы закончим с кодом. Отдельный модуль `Utils` будет содержать все функции общего назначения, преобразования между типами `OpenGL` и `Hipmunk`.

Детализируем функции обновления состояния игры

Начнём с восприятия:

```
module World where
```

```
import qualified Physics.Hipmunk as H
```

```
import Data.Maybe
```

```
import Types
```

```
import Utils
```

```
import Pure
```

```
import Dirty
```

```
percept :: Dirty -> IO (Sense, [Event])
```

```
percept a = do
```

```
    hero    <- obj2hero $ dirtyHero a
```

```
    balls   <- mapM (uncurry obj2ball) $ setIds dirtyObjs a
```

```
    evts1    <- fmap maybeToList $ getTouch (dirtyTouchVar a) $ dirtyObjs a
```

```
    evts2    <- fmap maybeToList $ getClick $ dirtyMouse a
```

```
    return $ (Sense hero balls, evts1 ++ evts2)
```

```
    where setIds = zip [0..]
```

```
-- в Dirty.hs
```

```
obj2hero    :: Obj -> IO HeroBall
```

```
obj2ball    :: Id -> Obj -> IO Ball
```

```
getTouch    :: Sensor H.Shape -> IxMap Obj -> IO (Maybe Event)
getClick    :: Sensor H.Position -> IO (Maybe Event)
```

Далее мы не будем каждый раз выписывать новые неопределённые функции, мы будем просто оставлять объявления типов без определений. Итак мы написали одну функцию, и получили ещё четыре новых.

Мы сделаем предположение о том, что сначала мы реагируем на непрерывные события, а затем на дискретные. Причём к запросам на реакции могут привести только дискретные события:

```
updatePure :: Sense -> [Event] -> Pure -> (Pure, [Query])
updatePure s evts = updateEvents evts . updateSenses s
```

```
-- в Pure.hs
updateSenses :: Sense -> Pure -> Pure
updateEvents :: [Event] -> Pure -> (Pure, [Query])
```

В функции `react` мы предполагаем, что реакции мира на события независимы друг от друга. `foldQuery~` функция свёртки для типа `Query`.

```
import Control.Monad
...

react :: [Query] -> Dirty -> IO Dirty
react = foldr (<=<) return
      . fmap (foldQuery removeBall heroVelocity makeBall)

-- в Dirty.hs
removeBall    :: Ball          -> Dirty -> IO Dirty
heroVelocity  :: H.Velocity    -> Dirty -> IO Dirty
makeBall      :: Freq          -> Dirty -> IO Dirty
```

Обратите внимание на то, как мы воспользовались функциями `foldr`, `return` и `<=<` для того чтобы нанизывать друг на друга функции типа `Dirty -> IO Dirty`. Напомню, что функция `<=<~` это аналог композиции для монадных функций.

Обновление модели:

```
updateDirty :: Dirty -> IO Dirty
updateDirty = stepDirty dt

-- в Dirty.hs
stepDirty :: H.Time -> Dirty -> IO Dirty
```



```
-- в Inits.hs
dt :: H.Time
dt = 0.5
```

Функции рисования поместим в отдельный модуль `Graphics.hs`

```
-- переместим из Loop.hs в World.hs
drawWorld :: World -> IO ()
drawWorld = draw . picture . worldPure
```

```
-- в Graphics.hs
draw :: Picture -> IO ()
```

```
-- в Pure.hs
picture      :: Pure -> Picture
```

Добавим функцию инициализации игры:

```
initWorld :: IO World
initWorld = do
    dirty    <- initDirty
    (sense, events) <- percept dirty
    return $ World (initPure sense events) dirty
```

```
-- в Dirty.hs
initDirty :: IO Dirty
-- в Pure.hs
initPure  :: Sense -> [Event] -> Pure
```

Детализируем дальше

Вот так на самом интересном месте... Мы вынуждены прерваться. Я надеюсь, что вы уловили основную идею метода и сможете закончить эту игру самостоятельно. Вся логика игры будет описана в модуле `Pure.hs`. Причём в этом модуле будут только чистые функции. Осталось примерно 1000 строк кода. Я не буду выписывать своё решение, если вы где-то запнётесь или у вас что-то не будет получаться, вы можете свериться с ним (оно входит в код, что прилагается с книгой).

Краткое содержание

В этой главе мы посмотрели на две интересные библиотеки. Физический движок `Hipmunk` и графическую библиотеку `OpenGL` и

узнали метод укрощения императивного кода. Мы разделили состояние игры на две части. В одну поместили все те параметры, для которых невозможно обойтись без **IO**-функций, а в другой те параметры, которые необходимы для реализации логики игры. Все функции, отвечающие за логику игры являются чистыми. Параметры императивной части не обновляются сразу, сначала мы делаем с них снимок, потом передаём этот снимок в чистую часть, и она разбирается с тем как их обновлять. Части общаются между собой на специальных маленьких языках, которые закодированы в типах. Это язык наблюдений (**Event**), язык реакций (**Query**) и язык отрисовки игрового мира (**Picture**).

Упражнения

Закончите код игры. Или, возможно, при знакомстве с **Hipmunk** у вас появилась идея новой игры с невероятной динамикой. Ещё лучше! Напишите её. При этом продумайте проект игры так, чтобы **IO**-типы не разбежались по всей программе.

Музыкальный пример

В этой главе мы напишем музыкальный секвенсор. Мы будем переводить нотную запись в midi-файл с помощью библиотеки **HCodecs**. Она предоставляет возможность создания midi-файлов по описанию в Haskell. При этом описание напоминает описание самого формата midi. Мы же хотим подняться уровнем выше и описывать музыку нотами и композицией нот.

Музыкальная нотация

Для начала зададимся вопросом: а что же такое музыка с точки зрения нашего секвенсора? Мы ищем представление музыки, термины, в которых было бы удобно мыслить композитору. При этом необходимо понимать, что наш поиск ограничен средствами низкоуровневого представления музыки. В нашем случае это midi-файл. Так например мы можем сразу отбросить представление в виде сигналов, последовательности сэмплов, поскольку мы не сможем реализовать это представление в рамках midi. За ответом обратимся к истории.

Нотная запись в европейской традиции

В европейской традиции принято описывать музыку в виде нотной записи. Нотный лист состоит из серии нотных станков. Нотный стан состоит из пяти линеек. Каждая линейка обозначает определённую высоту. Нота состоит из обозначения длительности и высоты. Разные длительности обозначаются штрихами и цветом ноты, а высоте соответствует расположение на нотном стане.



Буквенные обозначения высоты ноты

По длительности ноты различают на: целые, половины, четверти, восьмые, шестнадцатые и так далее. Каждая последующая

длительность в два раза меньше предыдущей. Длительность измеряется в долях от такта. Такты обозначаются сплошной линией, которая перечёркивает все пять линеек нотного стана. По высоте ноты, зависят от двух целых чисел, это номер октавы и номер ступени лада. В ладе обычно всего 12 ступеней. Их обозначают разными именами. Например в латинской нотации их обозначают так:

0	1	2	3	4	5	6	7	8	9	10	11
\$C\$	\$C#\$	\$D\$	\$D#\$	\$E\$	\$F\$	\$F#\$	\$G\$	\$G#\$	\$A\$	\$A#\$	\$B\$
\$C\$	\$Db\$	\$D\$	\$Eb\$	\$E\$	\$F\$	\$Gb\$	\$G\$	\$Ab\$	\$A\$	\$Bb\$	\$B\$
\$do\$		\$re\$		\$mi\$	\$fa\$		\$sol\$		\$la\$		\$ti\$

В самом нижнем ряду расположены имена нот. Во втором и четвёртом – обозначения нот с диезами и с бемолями. Одна и та же нота может обозначаться по-разному. Буквами обозначают ноты тональности до мажор (это семь букв для семи нот), а остальные ноты получают повышением на один шаг с помощью знака диез $\$ \# \$$ или понижением на один шаг с помощью знака бемоль $\$ b \$$.

Также ноты различают по громкости. В европейской традиции считается, что громкость изменяется не часто в сравнении с высотой и длительностью, поэтому для обозначения громкости введены специальные символы, которые пишутся под нотным станом, только когда громкость изменяется.

Из этого обзора мы поняли, что единицей музыкальной записи является нота, она состоит из обозначения длительности, высоты и громкости. Высота в свою очередь состоит из обозначения октавы и ступени лада. Теперь давайте посмотрим крупным планом на протокол midi.

Протокол midi

Протокол midi появился в ответ на бурное развитие синтезаторов. Каждый из синтезаторов предлагал свои тембры, при этом люди задумались, а нужна ли синтезатору клавиатура? Вопрос кажется

абсурдным, если мы думаем об одном синтезаторе, но представьте, что у вас их десять, в каждом свой чем-то особенный тембр. При этом нам нужно десять разных тембров, но мы вынуждены таскать за собой десять примерно одинаковых клавиатур. Для того чтобы отделить тембр от управления (нажатия на клавиши игроком) был придуман протокол `midí`. Протокол `midí` описывает специфическую для нажатия на клавиши информацию. Производители тембров или генераторов тона, могут научить генератор тона понимать `midí`. При этом мы можем сделать отдельную клавиатуру, которая не имеет собственного генератора тона, но умеет посылать сообщения протокола `midí`, так мы сможем управлять десятью генераторами тона от разных производителей с помощью одной клавиатуры. Такие клавиатуры называют `midí`-клавиатурами.

Познакомимся с терминологией `midí`. Протокол `midí` рассчитан на управление синтезаторами в режиме реального времени. Можно сказать, что `midí`-файл – это история концерта или выступления, низкоуровневая нотная запись. Каждое движение игрока кодируется событием. Например нажатие на клавишу, отпускание клавиши, сила давления на клавишу в определённый момент времени, нажатие педали, поворот реле или смена тембра.

Протокол `midí` изначально задумывался как расширяемый протокол. Каждый производитель тембров имеет возможность добавить какие-то особенные настройки. При этом те сообщения, которые данный генератор тона не понимает просто игнорируются. Наш секвенсор будет понимать такие события как нажатие на клавишу и отпускание клавиши. Также у нас будут разные инструменты.

Установим библиотеку `HCodecs` с `Hackage`:

```
cabal install HCodecs
```

Теперь заглянем на страницу документации этого пакета (на сайте `Hackage`), нас интересует модуль `Codec.Midi`, ведь мы хотим создавать именно `midí`-файлы. Здесь мы видим описание протокола `midí`, закодированное в типах. Посмотрим на тип `Message`, он описывает `midí`-сообщения. В первую очередь нас интересуют конструкторы:

```
NoteOn {  
    channel :: !Channel,  
    key     :: !Key,  
    velocity :: !Velocity }
```

```
NoteOff {  
    channel :: !Channel,  
    key     :: !Key,  
    velocity :: !Velocity }
```

Восклицательные знаки перед типами означают взрывные шаблоны, о которых мы говорили в главах о ленивых вычислениях. Конструктор `NoteOn` обозначает нажатие клавиши на канале `Channel` с высотой `Key` и уровнем громкости `Velocity`. Конструктор `NoteOff` обозначает отпускание клавиши, параметры имеют тот же смысл, что и в случае `NoteOn`.

Думаю что такое высота и громкость примерно понятно, но что такое канал? Считается, что один исполнитель может управлять сразу несколькими генераторами тона. Управление распределяется по каналам. На каждом канале мы можем управлять отдельным инструментом. Немного о высоте и громкости. Они кодируются целыми числами из диапазона от 0 до 127. Ноте до первой октавы (`C`) соответствует цифра 60, ноте ля первой октавы (`A`) соответствует номер 69. Одно число кодирует сразу и октаву и ступень лада.

Может показаться странным параметр `Velocity` в конструкторе `NoteOff`, он обозначает отпускание клавиши с определённой громкостью. Обычно этот параметр игнорируется и в него записывают среднее значение 64 или начальное значение 0.

Также мы будем играть разными инструментами. Инструменты в протоколе `midі` называются программами. Мы можем установить определённый инструмент на данном канале с помощью сообщения:

```
ProgramChange {  
    channel :: !Channel,  
    preset  :: !Preset }
```

Целое число `Preset` указывает на код инструмента. Теперь посмотрим, что же такое `midі`-файл:

```
data Midi = Midi {  
    fileType :: FileType,  
    timeDiv  :: TimeDiv,  
    tracks   :: [Track Ticks] }
```

midі-файл состоит из трёх значений. Это обозначение типа файла:

```
data FileType = SingleTrack | MultiTrack | MultiPattern
```

По типу midі-файлы могут различаться на файлы с одним треком, файлы с несколькими треками, и файлы, которые содержат группы треков, которые называют узорами (pattern). По смыслу трек соответствует партии инструмента.

Тип `TimeDiv` кодирует скорость записи сообщений. Различают два варианта:

```
data TimeDiv = TicksPerBeat Int  
              | TicksPerSecond Int Int
```

Первый конструктор говорит о том, что разрешение времени закодировано в формате PPQN, он указывает на число ударов в одной четвертной длительности. Второй конструктор говорит о том, что разрешение кодируется в формате SMPTE, оно указывает на число кадров в секунде.

Теперь посмотрим, что такое трек:

```
type Track a = [(a, Message)]
```

Трек это список событий с временными отсчётами. Время в midі отсчитывается относительно предыдущего события. Например в следующей записи три события произошли одновременно и затем спустя 10 тактов произошли ещё два события:

```
[(0, e1), (0, e2), (0, e3), (10, e4), (0, e5)]
```

Музыкальная запись в виде событий

Писать музыку в виде событий midі очень неудобно, пусть даже и через `HCodecs`, необходимо придумать надстройку над протоколом midі. Я долго думал об этом и в итоге пришёл к выводу, что наиболее простой и податливый способ представления музыки на

нотном уровне реализован в языке Csound. Там ноты представлены в виде последовательности событий. Каждое событие начинается в определённый момент и длится некоторое время. Событие содержит код инструмента и набор параметров, которые могут включать в себя громкость, высоту звука и какие-то специфические для данного инструмента настройки. Обязательными параметрами события являются лишь номер инструмента, который играет ноту, начало события и длительность события. Мы ослабим эти ограничения. Событие будет содержать лишь время начала, длительность и некоторое содержание.

```
data Event t a = Event {  
    eventStart    :: t,  
    eventDur      :: t,  
    eventContent  :: a  
} deriving (Show, Eq)
```

Параметр `t` символизирует время, а параметр `a` – некоторое содержание события. Мы будем говорить, что в некоторый момент времени произошло значение типа `a` и оно длилось некоторое время. Треком мы будем называть набор событий, которые длятся определённой время:

```
data Track t a = Track {  
    trackDur      :: t,  
    trackEvents   :: [Event t a]  
}
```

Первый параметр указывает на общую длительность трека, а второй содержит события, которые произошли. Мы явно указываем длительность трека для того, чтобы иметь возможность представить тишину. Значение тишины будет выглядеть так:

```
silence t = Track t []
```

Этим мы говорим, что ничего не произошло в течение `t` единиц времени.

Преобразование событий во времени

Наши события привязаны ко времени. Мы можем ввести линейные операции, которые будут изменять расположение событий во времени. Самый простой способ изменения положения это задержка. Мы можем

задержать появление события, прибавив какое-нибудь число ко времени начала события:

```
delayEvent :: Num t => t -> Event t a -> Event t a
delayEvent d e = e{ eventStart = d + eventStart e }
```

Ещё одно простое преобразование заключается в изменении масштаба времени, в музыке или анимации этой операции соответствует перемотка. Событие начинает происходить быстрее или медленнее:

```
stretchEvent :: Num t => t -> Event t a -> Event t a
stretchEvent s e = e{
    eventStart = s * eventStart e,
    eventDur   = s * eventDur   e }
```

Для изменения масштаба времени мы умножили временные параметры на число s . Эти операции мы можем перенести и на значения типа `Track`.

```
delayTrack :: Num t => t -> Track t a -> Track t a
delayTrack d (Track t es) = Track (t + d) (map (delayEvent d) es)
```

```
stretchTrack :: Num t => t -> Track t a -> Track t a
stretchTrack s (Track t es) = Track (t * s) (map (stretchEvent s) es)
```

Класс преобразований во времени

У нас есть аналогичные операции преобразования во времени для событий и треков, это говорит о том, что мы можем ввести специальный класс, который объединит в себе эти операции. Назовём его классом `Temporal` (временной):

```
class Temporal a where
    type Dur a :: *
    dur      :: a -> Dur a
    delay    :: Dur a -> a -> a
    stretch :: Dur a -> a -> a
```

В этом классе определён один тип, который обозначает размерность времени, и три метода в дополнении к методам `delay` и `stretch` мы добавим метод `dur`, мы будем считать, что всё что происходит во времени конечно и с помощью метода `dur` мы всегда можем узнать протяжённость значения их класса `Temporal` во времени. Для определения этого класса нам придётся подключить расширение

TypeFamilies. Теперь мы легко можем определить экземпляры класса **Temporal** для **Event** и **Track**:

```
instance Num t => Temporal (Event t a) where
  type Dur (Event t a) = t
  dur      = eventDur
  delay    = delayEvent
  stretch = stretchEvent
```

```
instance Num t => Temporal (Track t a) where
  type Dur (Track t a) = t
  dur      = trackDur
  delay    = delayTrack
  stretch = stretchTrack
```

Композиция треков

Определим две полезные в музыке операции: параллельную и последовательную композицию треков. В параллельной композиции мы играем два трека одновременно:

```
(=:=) :: Ord t => Track t a -> Track t a -> Track t a
Track t es := Track t' es' = Track (max t t') (es ++ es')
```

Теперь общая длительность трека равна длительности большего из треков, а события включают в себя события каждого из треков. С помощью преобразований во времени мы можем определить последовательную композицию, для этого мы сместим второй трек на длину первого и сыграем их одновременно:

```
(+:+) :: (Ord t, Num t) => Track t a -> Track t a -> Track t a
(+:+) a b = a := delay (dur a) b
```

При этом у нас как раз и получится, что мы сначала сыграем целиком трек **a**, а затем трек **b**. Теперь определим аналоги операций **:=** и **+:+** для списков:

```
chord :: (Num t, Ord t) => [Track t a] -> Track t a
chord = foldr (=:=) (silence 0)

line :: (Num t, Ord t) => [Track t a] -> Track t a
line = foldr (+:+) (silence 0)
```

Мы можем определить в терминах этих операций циклический повтор событий:

```
loop :: (Num t, Ord t) => Int -> Track t a -> Track t a
loop n t = line $ replicate n t
```

Экземпляры стандартных классов

Мы можем сделать тип трек экземпляром класса `Functor`:

```
instance Functor (Event t) where
    fmap f e = e{ eventContent = f (eventContent e) }

instance Functor (Track t) where
    fmap f t = t{ trackEvents = fmap (fmap f) (trackEvents t) }
```

Мы можем также определить экземпляр для класса `Monoid`.

Параллельная композиция будет операцией объединения, а нейтральным элементом будет тишина, которая длится ноль единиц времени:

```
instance (Ord t, Num t) => Monoid (Track t a) where
    mappend = (==)
    mempty   = silence 0
```

Ноты в midi

С помощью типа `Track` мы можем описывать всё, что имеет свойство случаться во времени и длиться, мы можем описывать наборы событий. Операции из класса `Temporal` и операции последовательной и параллельной композиции дают нам возможность собирать сложные наборы событий из простейших. Но для того чтобы это стало музыкой, нам не хватает нот.

Так построим их. Поскольку мы собираемся играть музыку в midi, наши ноты будут содержать только три основных параметра, это номер инструмента, громкость и высота. Длительность ноты будет кодироваться в событии, эта информация уже встроена в тип `Track`.

```
data Note = Note {
    noteInstr  :: Instr,
    noteVolume :: Volume,
    notePitch  :: Pitch,
    isDrum     :: Bool }
```

Итак нота содержит код инструмента, громкость и высоту и ещё один параметр. По последнему параметру можно узнать сыграна нота на барабанах или нет. В midi ноты для ударных обрабатываются особым образом. Десятый канал выделен под ударные, при этом номер инструмента игнорируется, а вместо этого высота звука кодирует

номер ударного инструмента. Теперь определимся с типами параметров:

```
type Instr = Int
type Volume = Int
type Pitch = Int
```

Целые числа соответствуют целым числам в протоколе midi. Значения для типов **Volume** и **Pitch** лежат в диапазоне от 0 до 127.

Введём специальное обозначение для музыкального типа **Track**:

```
type Score = Track Double Note
```

Синонимы для нот

Высота ноты

Музыкантам ближе буквенные обозначения для нот нежели коды midi. Определим удобные синонимы:

```
note :: Int -> Score
note n = Track 1 [Event 0 1 (Note 0 64 (60+n) False)]
```

Эта функция строит трек, который содержит одну ноту. Нота длится одну целую длительность играется на инструменте с кодом 0, на средней громкости. Параметр функции задаёт смещение от ноты до первой октавы. Определим остальные ноты:

```
a, b, c, d, e, f, g,
  as, bs, cs, ds, es, fs, gs,
  af, bf, cf, df, ef, ff, gf :: Score

c = note 0;    cs = note 1;    d = note 2;    ds = note 3;
...
```

Первая буква содержит буквенное обозначение ноты, а вторая либо s (от англ. sharp диез) или f (от англ. flat бемоль). Все эти ноты находятся в первой октаве, но смещением высоты на 12 единиц мы легко можем смещать эти ноты в любую другую октаву:

```
higher :: Int -> Score -> Score
higher n = fmap (\a -> a{ notePitch = 12*n + notePitch a })

lower :: Int -> Score -> Score
lower n = higher (-n)
```

```
high :: Score -> Score
high = higher 1
```

```
low :: Score -> Score
low = lower 1
```

С помощью этих функций мы легко можем смещать группы нот в любую октаву. Функция `higher` принимает число октав, на которые необходимо сместить вверх высоту во всех нотах трека. Смещение высоты на 12 определяет смещение на одну октаву. Остальные функции определены в через функцию `higher`.

Длительность ноты

Пока что наши ноты длятся 1 единицу времени. Но нам бы хотелось иметь в распоряжении и другие длительности. Ноты других длительностей мы можем легко получать с помощью функции `stretch`, мы просто изменим масштаб времени и длительность всех нот изменится. Определим несколько синонимов:

```
bn, hn, qn, en, sn :: Score -> Score
```

```
-- (brewis note)    (half note)    (quater note)
bn = stretch 2;    hn = stretch 0.5;    qn = stretch 0.25;
```

```
-- (eighth note)    (sixth note)
en = stretch 0.125;    sn = stretch 0.0625;
```

Эти преобразования отвечают длительностям нот в европейской музыкальной традиции.

Громкость ноты

Пока мы умеем создавать ноты средней громкости, но мы можем определить преобразователи на манер тех, что изменяли высоту звука октавами:

```
louder :: Int -> Score -> Score
louder n = fmap $ \a -> a{ noteVolume = n + noteVolume a }
```

```
quieter :: Int -> Score -> Score
quieter n = louder (-n)
```

Смена инструмента

Изначально мы создаём ноты, которые играют на инструменте с кодом 0, в протоколе General Midi этот номер соответствует роялю. Но с помощью класса `Funcionr` мы легко можем изменить инструмент:

```
instr :: Int -> Score -> Score
instr n = fmap $ \a -> a{ noteInstr = n, isDrum = False }

drum :: Int -> Score -> Score
drum n = fmap $ \a -> a{ notePitch = n, isDrum = True }
```

Согласно протоколу midi в случае ударных инструментов высота звука кодирует инструмент. Поэтому в функции `drum` мы изменяем именно поле `notePitch`. Создадим также несколько синонимов для создания нот, которые играют на барабанах. В этом случае нам не важна высота звука но важна громкость:

```
bam :: Int -> Score
bam n = Track 1 [Event 0 1 (Note 0 n 35 True)]
```

Номер 35 кодирует “бочку”.

Паузы

Слово `silence` верно отражает смысл, но оно слишком длинное. Давайте определим несколько синонимов:

```
rest :: Double -> Score
rest = silence

wnr = rest 1;   bnr = bn wnr;   hnr = hn wnr;
qnr = qn wnr;  enr = en wnr;   snr = sn wnr;
```

Перевод в midi

Теперь мы можем составить какую нибудь мелодию:

```
q = line [c, c, hn e, hn d, bn e, chord [c, e]]
```

Мы можем составлять мелодии, но пока мы не умеем их интерпретировать. Для этого нам нужно написать функцию:

```
render :: Score -> Midi
```

Мы реализуем простейший случай. Будем считать, что у нас только 15 инструментов, а все остальные инструменты – ударные. Мы запишем нашу музыку на один трек midi-файла, распределив 15 неударных инструментов по разным каналам. Ещё одно упрощение заключается в том, что мы зададим фиксированное разрешение по времени для всех возможных мелодий. Будем считать, что 96 ударов для одной четверти нам достаточно. Принимая во внимания эти посылки мы можем написать такую функцию:

```
import qualified Codec.Midi as M

render :: Score -> Midi
render s = M.Midi M.SingleTrack (M.TicksPerBeat divisions) [toTrack s]

divisions :: M.Ticks
divisions = 96

toTrack :: Score -> M.Track
toTrack = undefined
```

Мы загрузили модуль `Codec.Midi` под псевдонимом `M`, так мы сможем отличать низкоуровневые определения от тех, что мы определили сами. Теперь перед каждым именем из модуля `Codec.Midi` необходимо писать приставку `M`.

В нашей упрощённой реализации на одном канале может играть только один инструмент. В самом начале мы назначим инструмент на канал с помощью сообщения `ProgramChange`. Для этого нам необходимо понять какому инструменту какой канал соответствует. В библиотеке `HCodecs` каналы идут от нуля до 15. Девятый канал предназначен для ударных. Представим, что у нас есть функция, которая распределяет нотную запись по инструментам:

```
type MidiEvent = Event Double Note

groupInstr :: Score -> ([[MidiEvent]], [MidiEvent])
```

Эта функция принимает нотную запись, а возвращает пару. Первый элемент содержит список списков нот для неударных инструментов, каждый подсписок содержит ноты только для одного инструмента. Второй элемент пары содержит все ноты для ударных инструментов. Представим также, что у нас есть функция, которая превращает эту пару в набор midi-сообщений:

```
mergeInstr :: ([[MidiEvent]], [MidiEvent]) -> M.Track Double
```

Наши отсчёты времени записаны в виде значений типа `Double`, Нам необходимо перейти к целочисленным `Ticks`. Представим, что такая функция у нас уже есть:

```
tfmTime :: M.Track Double -> M.Track M.Ticks
```

Тогда функция `toTrack` примет вид:

```
toTrack :: Score -> M.Track M.Ticks  
toTrack = tfmTime . mergeInstr . groupInstr
```

Все три составляющие функции пока не определены. Начнём с функции `tfmTime`. Нам необходимо отсортировать события во времени для того, чтобы мы смогли перейти из абсолютных отсчётов во времени в относительные. Специально для этого в библиотеке `HCodecs` определена функция:

```
fromAbsTime :: Num a -> Track a -> Track a
```

Также нам понадобится функция:

```
type Time = Double
```

```
fromRealTime :: TimeDiv -> Ttrack Time -> Track Ticks
```

Она проводит квантование во времени. С помощью неё мы преобразуем отсчёты в `Double` в целочисленные отсчёты. С помощью этих функций мы можем определить функцию `timeDiv` так:

```
import Data.List(sortBy)  
import Data.Function (on)  
...  
  
tfmTime :: M.Track Double -> M.Track M.Ticks  
tfmTime = M.fromAbsTime . M.fromRealTime timeDiv .  
    sortBy (compare `on` fst)
```

В этой функции мы сначала сортируем события во времени, затем переходим от абсолютных единиц к относительным и в самом конце производим квантование по времени. Функция `sortBy` сортирует элементы согласно некоторой функции упорядочивания:

```
sortBy :: (a -> a -> Ordering) -> [a] -> [a]
```


Она принимает функцию упорядочивания и список. Мы воспользовались этой функцией, потому что нам необходимо отсортировать элементы списка сообщений по значению временных отсчётов. Функцию упорядочивания мы составляем с помощью специальной функции `on`, которая определена в модуле `Data.Function`. С этой функцией мы уже сталкивались, когда говорили о функциях высшего порядка, она принимает функцию двух аргументов и функцию одного аргумента и словно “подкладывает” вторую функцию под первую:

```
Prelude Data.Function> :t on
on :: (b -> b -> c) -> (a -> b) -> a -> c
```

Теперь напомним функцию `mergeInstr`. Она устанавливает инструменты на каналы и преобразует события в последовательность `midi`-сообщений. При этом мы различаем сообщения для ударных и сообщения для всех остальных инструментов:

```
mergeInstr :: ([[MidiEvent]], [MidiEvent]) -> M.Track Double
mergeInstr (instrs, drums) = concat $ drums' : instrs'
  where instrs' = zipWith setChannel ([0 .. 8] ++ [10 .. 15]) instrs
        drums'  = setDrumChannel drums
```

```
setChannel :: M.Channel -> [MidiEvent] -> M.Track Double
setChannel = undefined
```

```
setDrumChannel :: [MidiEvent] -> M.Track Double
setDrumChannel = undefined
```

Имя `instrs'` указывает на последовательность списков сообщений для каждого неударного инструмента. Функция `setChannel` принимает номер канала и список событий. По ним она строит список `midi`-сообщений. Определим эту функцию:

```
setChannel :: M.Channel -> [MidiEvent] -> M.Track Double
setChannel ch ms = case ms of
  []      -> []
  x:xs    -> (0, M.ProgramChange ch (instrId x)) : (fromEvent ch =<< ms)
```

```
instrId = noteInstr . eventContent
```

```
fromEvent :: M.Channel -> MidiEvent -> M.Track Double
fromEvent = undefined
```

Первым событием мы присоединяем событие, которое устанавливает на данном канале определённый инструмент. По построению программы

все ноты в переданном списке играют на одном и том же инструменте, поэтому мы узнаём идентификатор инструмента из первого элемента списка. У нас появилась новая неопределённая функция `fromEvent` она переводит сообщение в список `midi`-сообщений:

```
fromEvent :: M.Channel -> MidiEvent -> M.Track Double
fromEvent ch e = [
  (eventStart e, noteOn n),
  (eventStart e + eventDur e, noteOff n)]
  where n = clipToMidi $ eventContent e
        noteOn n = M.NoteOn ch (notePitch n) (noteVolume n)
        noteOff n = M.NoteOff ch (notePitch n) 0

clipToMidi :: Note -> Note
clipToMidi n = n {
  notePitch  = clip $ notePitch n,
  noteVolume = clip $ noteVolume n }
  where clip = max 0 . min 127
```

Определив эти функции, мы легко можем написать и функцию `setDrumChannel` она переводит сообщения для ударных инструментов в `midi`-сообщения:

```
setDrumChannel :: [MidiEvent] -> M.Track Double
setDrumChannel ms = fromEvent drumChannel =<< ms
  where drumChannel = 9
```

Для ударных инструментов выделен отдельный канал. Считается, что все они происходят на 10 канале. Поскольку в библиотеке `HCodecs` первый канал называется нулевым, мы будем записывать все сообщения на девятый канал.

Мы переводим событие в два `midi`-сообщения, первое говорит о том, что мы начали играть ноту, а второе говорит о том, что мы закончили её играть. Функция `clipToMidi` приводит значения для высоты и громкости в диапазон `midi`.

Нам осталось определить только одну функцию. Эта функция распределяет события по инструментам. Сначала мы разделим события на те, что играют на ударных и неударных инструментах, а затем разделим “неударные” ноты по инструментам:

```
import Control.Arrow(first, second)
import Data.List(sortBy, groupBy, partition)
```

...

```
groupInstr :: Score -> ([[MidiEvent]], [MidiEvent])
groupInstr = first groupByInstrId .
  partition (not . isDrum . eventContent) . trackEvents
  where groupByInstrId = groupBy ((==) `on` instrId) .
    sortBy (compare `on` instrId)
```

В этом определении мы воспользовались двумя новыми стандартными функциями из модуля `Data.List`. Функция `partition` разделяет список на пару списков. В первом списке находятся все те элементы, для которых заданный предикат вернул `True`, а во втором списке – все остальные элементы исходного списка:

```
Prelude Data.List> :t partition
partition :: (a -> Bool) -> [a] -> ([a], [a])
```

Функция `groupBy` превращает список в список списков:

```
Prelude Data.List> :t groupBy
groupBy :: (a -> a -> Bool) -> [a] -> [[a]]
```

Если бинарная функция на соседних элементах исходного списка вернула `True`, то они помещаются в один подсписок. Эта функция используется для того чтобы сгруппировать элементы списка по какому-нибудь признаку. При этом для того чтобы сгруппировать элементы по идентификатору инструмента, мы сначала отсортировали события по значению идентификатора. После этого значения с одинаковыми идентификаторами стали соседними и мы сгруппировали их с помощью `groupBy`.

Функция `first` применяет функцию к первому элементу пары. Вот мы и закончили, можно послушать результаты. На самом деле остались два нюанса. В функции `setChannel` мы полагаем, что мелодия начинается в момент времени `t = 0`, но на практике это может оказаться не так, мы можем сместить ноты функцией `delay` в отрицательную сторону. Тогда первые ноты будут содержать отрицательное время начала события. Но мы можем исправить эту ситуацию, сместив все ноты на время самой первой ноты, конечно смещать необходимо только в том случае если время окажется отрицательным:

```
alignEvents :: [MidiEvent] -> [MidiEvent]
alignEvents es
  | d < 0      = map (delay (abs d)) es
  | otherwise = es
  where d = minimum $ map eventStart es
```

Вызовем эту функцию сразу после функции `trackEvents` в функции `groupInstr`. Второй нюанс заключается в том, что каждый трек в midi-файле должен заканчиваться специальным сообщением, в библиотеке `HCodecs` оно обозначается с помощью конструктора `TrackEnd`. В самом конце необходимо добавить сообщение `(0, TrackEnd)`:

```
toTrack :: Score -> M.Track M.Ticks
toTrack = addEndMsg . tfmTime . mergeInstr . groupInstr

addEndMsg :: M.Track M.Ticks -> M.Track M.Ticks
addEndMsg = (++ [(0, M.TrackEnd)])
```

Теперь мы можем проверить, что у нас получилось. Создадим файл:

```
module Main where

import System
import Track
import Score
import Codec.Midi

out = (>> system "timidity tmp.mid") .
      exportFile "tmp.mid" . render
```

В функции `out` мы переводим нотную запись в значение типа `Midi`, затем сохраняем это значение в файле `tmp.mid` и в самом конце запускаем файл с помощью проигрывателя `timidity`. Вместо `timidity` вы можете воспользоваться вашим любимым проигрывателем midi-файлов. Теперь загрузим модуль `Main` в интерпретатор. Послушаем ноту до:

```
*Main> out c
```

Далее следуют сообщения из проигрывателя `timidity` и долгожданный звук. Мы слышим ноту до, сыгранную на рояле. Наберём какую-нибудь мелодию:

```
*Main> let x = line [c, hn e, hn e, low b, c]
*Main> out x
```

Сыграем в два раза быстрее, на другом инструменте:

```
*Main> out $ instr 15 $ hn x
```

Сыграем канон. Канон это когда одна и та же мелодия ведётся в разных голосах с запаздыванием. Сыграем двухголосный канон:

```
*Main> out $ instr 80 (loop 3 x) := delay 2 (instr 65 $ low $ loop 3 x)
```

Номера инструментов можно посмотреть по справке к протоколу General Midi. Это дополнение к протоколу midi определяет какие номера каким инструментам должны соответствовать. Звучит ужасно, но звучит!

Пример

Опираясь на примитивы композиции, которые мы определил в модуле **Score**, мы можем написать мелодию. Ниже приведён небольшой пример. Инструменты:

```
closedHiHat = drum 42;      rideCymbal = drum 59;      cabasa = drum 69;
maracas     = drum 70;      tom         = drum 45;
flute       = instr 73;     piano        = instr 0;
```

Ударная секция:

```
b1 = bam 100
b0 = bam 84
```

```
drums1 = loop 80 $ chord [
  tom   $ line [qn b1, qn b0, hnr],
  maracas $ line [hnr, hn b0]
]
```

```
drums2 = quieter 20 $ cabasa $ loop 120 $ en $ line [b1, b0, b0, b0, b0]
```

```
drums3 = closedHiHat $ loop 50 $ en (line [b1, loop 12 wnr])
```

```
drums = drums1 := drums2 := drums3
```

Уже сейчас мы можем загрузить эту партию в интерпретатор и послушать, вызвав `out drums`. Аккорды к мелодии:

```
c7 = chord [c, e, b]
gs7 = chord [low af, c, g]
g7 = chord [low g, low bf, f]
```

```
harmony = piano $ loop 12 $ lower 1 $ bn $ line [bn c7, gs7, g7]
```

Мелодия:

```
ac = louder 5
```

```
mel1 = bn $ line [bnr, subMel, ac $ stretch (1+1/8) e, c,  
subMel, enr]  
where subMel = line [g, stretch 1.5 $ qn g, qn f, qn g]
```

```
mel2 = loop 2 $ qn $ line [subMel, ac $ bn ds, c, d, ac $ bn c, c, c, wnr,  
subMel, ac $ bn g, f, ds, ac $ bn f, ds, ac $ bn c]  
where subMel = line [ac ds, c, d, ac $ bn c, c, c]
```

```
mel3 = loop 2 $ line [pat1 (high c) as g, pat1 g f d]  
where pat1 a b c = line [pat a, loop 3 qnr, wnr,  
pat b, qnr, hnr, pat c, qnr, hnr]  
pat x = en (x :+: x)
```

```
mel = flute $ line [mel1, mel2, mel3]
```

Добавим в конце звук тарелки:

```
cha = delay (dur mel1 + dur mel2) $ loop 10 $ rideCymbal $ delay 1 b1
```

Соберём всё вместе и послушаем:

```
res = chord [  
drums,  
harmony,  
high mel,  
louder 40 cha,  
rest 0]
```

```
main = out res
```

В конце стоит фиктивный элемент `rest 0` для того чтобы было удобно глушить инструменты комментированием.

Эффективное представление музыкальной нотации

Реализация, которую мы рассмотрели не эффективна, Мы могли бы определить тип `Track` и по-другому. Мы очень часто пользуемся операцией `delay` через операцию `line`. Так в выражении:

```
q = line [s1, s2, line [loop 2 s3, s4], s5]
```

Мы будем несколько раз обходить элемент `s3` для каждого применения `line`. К примеру сначала мы смести все элементы на 3, потом сместим на 5, потом на 10, но вместо этого мы могли бы сразу сместить все элементы на 18 за один проход. Для этого мы можем закодировать преобразования событий во времени в типе `Track`:

```
data Track t a = Track {  
    trackDur    :: t,  
    trackEvents :: TList t a
```

```
data TList t a = Empty | Single a | Append (TList t a) (TList t a)  
              | TFun (Tfm t) (TList t a)
```

```
data Tfm t = Tfm !t !t
```

Тип `TList` позволяет проводить быстрое объединение списков. Дополнительный конструктор `TFun` обозначает линейное преобразование списка во времени. Линейное преобразование кодируется двумя числами, это масштаб и смещение. Мы считаем, что события в конструкторе `Single` начинаются в момент времени 0 и длятся 1 единицу времени. Так например событие, которое произошло на 2 единице времени и длилось 4 единицы можно представить так:

```
TFun (4 2) (Single a)
```

Значение `Tfm k d` обозначает линейную функцию

$$f(x) = k \cdot x + d$$

Для того чтобы получить настоящие отсчёты по времени мы применяем её к временным координатам “не преобразованного” события, то есть события `Event 0 1 a`.

Единственное, что нам нужно для того чтобы встроить этот вариант в библиотеку это написать функцию:

```
fromTList :: TList t a -> [Event t a]
```

И конечно переопределить все функции композиции. Но все сложные функции, которые отвечают за перевод из `Track` в `Midi` останутся прежними.

Краткое содержание

В этой главе мы построили секвенсор для создания midi-файлов. Мы воспользовались библиотекой **HCodecs** и создали над ней небольшую надстройку.

В нашей библиотеке примитивными конструкциями были события, параллельная композиция (одновременное воспроизведение) и преобразование событий во времени (сдвиг и масштабирование). Все остальные операции выражались через эти простейшие операции. Отметим, что есть и другие подходы. Например в библиотеках **Haskore** и **Euterpea** примитивными конструкциями является единичное событие (без отметок во времени) и параллельная и последовательная композиции. Подход, который мы рассмотрели в более общем виде реализован в библиотеках `temporal-music-notation` и `temporal-music-notation-demo`.

Упражнения

- Попробуйте написать какую-нибудь мелодию.
- Подумайте каких операций не хватает. Например было бы удобно иметь возможность вырезать из мелодии куски. Так в примере у нас остались хвосты от ударной секции, определите операцию, которая позволяет убрать лишнее.

Приложения

Начало работы с Haskell

Компилятор

Для программирования в Haskell нам понадобится компилятор. Мы будем пользоваться наиболее развитым компилятором – GHC. Лучше всего устанавливать его вместе с Haskell Platform:

<http://hackage.haskell.org/platform/>

Haskell Platform содержит стабильную версию компилятора и много хороших, проверенных временем библиотек. Если по каким-то причинам установить Haskell Platform не удалось. Не отчаивайтесь, можно загрузить компилятор с сайта GHC:

<http://www.haskell.org/ghc/>

И далее установить все необходимые библиотеки с Hackage с помощью cabal (устанавливается отдельно с <http://www.haskell.org/cabal/>).

Среда разработки

Для Haskell существует очень мало сред разработки. Обычно на Haskell программируют в каких-нибудь продвинутых текстовых редакторах (vim, Emacs, scite, kate, notepad++). Отметим всё же среду разработки Leksah (<http://leksah.org/>), она написана на Haskell и её можно установить с Hackage.

Если вы не хотите разбираться с новым текстовым редактором или средой разработки, и вам нужна лишь подсветка синтаксиса можно воспользоваться gedit. Пишем код в gedit, сохраняем, переключаемся на ghci, пробуем, обновляем, пробуем, при случае компилируем или собираем в пакет. Всё это можно делать и в gedit.

Литература

О Haskell написано много интересных книг и статей, но все они на английском. На русском языке выходит электронный журнал “Практика

функционального программирования” (). Пока в нём доминируют два языка – это Erlang и Haskell.

Я бы хотел рассказать о тех книгах и статьях, которые мне помогли. Все они приняли активное участие в создании этой книги.

Книги

- Miran Lipovaa. Learn You A Haskell For A Great Good.

Очень хорошая книга для начинающих, Haskell в картинках.

Весёлая и познавательная книга¹

<http://learnyouahaskell.com/>

- Hal Daume III. Yet Another Haskell Tutorial.

Ещё одна очень хорошая книга для начинающих. Без картинок, но всё по делу.

- Paul Hudak. Haskell School of Expression.

Книга, которая иллюстрирует основные принципы функционального программирования на примере Haskell.

Главные достоинства – много текста об общих принципах и интересные приложения, картинки, музыка, анимация, управление роботами и всё это на Haskell.

- Paul Hudak. Haskell School of Music.

Пол Хьюдак увлекается не только Haskell, но и музыкой. Он написал книгу, которая целиком посвящена описанию музыки в Haskell:

<http://www.cs.yale.edu/homes/hudak/Papers/HSoM.pdf>

<http://haskell.cs.yale.edu/>

- Bryan O’Sullivan, Don Stewart, John Goerzen. Real World Haskell.

Очень полезная книга в помощь тем, кто хочет научиться писать настоящие, серьёзные программы. Авторы подробно

изучают вопросы, связанные с применением Haskell на практике.

<http://book.realworldhaskell.org/>

- Готовится к выходу книга Саймона Марлоу о параллельных вычислениях в Haskell. Обещает быть очень интересной, уже известно, что книга будет доступна в интернете.

Тематический сборник

Основы

- John Hughes. Why Functional Programming Matters.
- Paul Hudak, John Hughes, Simon Peyton Jones, Philip Wadler. A History of Haskell: Being Lazy With Class.
- Mark P. Jones. Functional Programming with Overloading and Higher-Order Polymorphism.
- Евгений Кирпичев. Элементы функциональных языков программирования, журнал Практика функционального программирования.
- Simon Thompson. Programming It in Haskell.
- Justin Bailey. Haskell Cheat Sheet.

Разработка программ сверху-вниз

- Дмитрий Астапов. Давно не брал я в руки шашек, журнал Практика функционального программирования.

Функторы и монады

- Conor McBride, Ross Paterson. Applicative programming with effects. Статья об аппликативных функторах.
- Philip Wadler. The Essence of Functional Programming.
Статья, в которой впервые зашла речь о применении монад в Haskell.

- Tarmo Uustalu, Varmo Vene. The Essence of Dataflow Programming.

Статья о комонадах, но есть много интересного и о монадах.

- Bulat Ziganshin. Haskell I/O inside: Down the Rabbit's Hole. Статья на HaskellWiki.
- John Launchbury, Simon Peyton Jones. Lazy functional state threads.

Статья о типе `ST`.

- Simon Peyton Jones. Tackling the Awkward Squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell.

Ленивые вычисления

- Douglas McIlroy. Power Series, Power Serious.
- Дмитрий Астапов. Реурсия+мемоизация=динамическое программирование, журнал Практика функционального программирования.
- Сергей Зефилов. Лень бояться, журнал Практика функционального программирования.
- Jerzy Karczmarczuk. Specific “scientific” data structures, and their processing.

Структурная рекурсия

- Graham Hutton. A tutorial on the universality and expressiveness of fold
- Jeremy Gibbons. Origami Programming.
- Jeremy Gibbons, Geraint Jones. The Under-Appreciated Unfold.

Лямбда-исчисление и функциональное программирование

- Шалак В.И. Шейнфинкель и комбинаторная логика.

- Paul Hudak: Conception, Evolution, and Application of Functional Programming Languages.

Длинная статья о развитии функциональных языков. Там есть главы о лямбда-исчислении.

- Бенджамин Пирс. Типы в языках программирования.

Большая книга о теории типов.

<http://newstar.rinet.ru/~goga/tapl/>

- Денис Москвин. Системы типизации лямбда-исчисления.

Курс видео-лекций.

<http://www.lektorium.tv/course/?id=22797>

- John Harrison. Introduction to Functional Programming.

Курс лекций по функциональному программированию, который читался в Университете Кембридж.

- А. Филд, П. Харрисон, Функциональное программирование, Москва “Мир”, 1993.

Большая книга для читателей, всерьёз заинтересовавшихся функциональным программированием. Прочитав её, вы сможете не только пользоваться ФП-языками но и написать такой язык самостоятельно.

- Rinus Plasmeijer and Marko van Eekelen. Functional Programming and Parallel Graph Rewriting.

В этой книге исследуются вопросы распараллеливания функциональных программ, построение компиляторов для функциональных языков.

Теория категорий

Две очень хорошие книги для начинающих:

- Maarten M. Fokkinga. Gentle Introduction to Category Theory.

Также где-то в сети есть и перевод на русский.

- Steve Awodey. Category Theory.
- Eugenia Cheng, Simon Willerton aka TheCatsters. Курс видео-лекций на youtube.

<http://www.scss.tcd.ie/Edsko.de.Vries/ct/catsters/linear.php>

<http://www.youtube.com/user/TheCatsters>

Статьи по категориальным типам:

- Varmo Vene. Categorical Programming with Inductive and Coinductive Types. Phd-диссертация.
- Erik Meijer, Graham Hutton. Bananas in Space: Extending Fold and Unfold to Exponential Types.
- Martin Erwig. Categorical Programming with Abstract Data Types.
- Martin Erwig. Metamorphic Programming: Structured Recursion for Abstract Data Types.

Практика

- Conal Elliott. Denotational design with type class morphisms.
- Johan Tibell. High Performance Haskell. Слайды с выступления.
- Johan Tibel. Faster persistent data structures through hashing. Слайды с выступления.
- Simon Marlow. Parallel and Concurrent Programming in Haskell.
- Edward Z. Yang. Блог о Haskell в картинках. Много полезной информации о лени и устройстве ghc.
<http://blog.ezyang.com/about/>
- Oleg Kiselyov. Блог в том числе и о Haskell. Много решений интересных и нетривиальных задач. <http://okmij.org/ftp/>

Как работает GHC

- Документация GHC:

<http://hackage.haskell.org/trac/ghc/wiki/Commentary>

- Don Stewart. Multi-paradigm Just-In-Time Compilation. BS Thesis, 2002.

Автор пробует компилировать Haskell-код в Java-код. При этом очень доступно объясняются внутренности STG.

- Simon Marlow, Simon Peyton Jones. The Glasgow Haskell Compiler. The Architecture of Open Source Application, Volume 2, 2012.
- Simon Marlow, Simon Peyton Jones. Making a Fast Curry: Push/Enter vs. Eval/Apply for Higher-order Languages. ICFP'04.
- Simon Peyton Jones. Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine.
- Simon Marlow, Tim Harris, Roshan P. James, Simon Peyton Jones. Parallel Generational-Copying Garbage Collection with a Block-Structured Heap. ISMM'08.
- Simon Peyton Jones, Andre Santos. A transformation-based optimizer for Haskell. Science of computer programming, 1998.
- Simon Peyton Jones, John Launchbury. Unboxed values as first citizens in a non-strict functional programming language. 1991.
- Simon Marlow, Simon Peyton Jones. Secrets of Glasgow Haskell Compiler inliner. 1999

Статья о тонкостях реализации прагмы **INLINE**.

- Simon Peyton Jones, Andrew Tolmach, Tony Hoare. Playing by the Rules, ICFP 2001

Статья о прагме **RULES**.

Встроенные проблемно-ориентированные языки (EDSL)

- Oleg Kiselyov. Implementing Explicit and Finding Implicit Sharing in EDSLs.

Чистое решение проблемы поиска дублирующих подвыражений.

- Andy Gill. Type-Safe Observable Sharing in Haskell.

Решение проблемы поиска дублирующих подвыражений с помощью расширения GHC, позволяющего проводить сравнение термов по указателям.

- Conal Elliott, Sigbjorn Finne, Oege de Moor. Compiling Embedded Languages.

Отчёт о построении EDSL для анимации.

- Bruno C.d.S. Oliveira, Andres Loh. Abstract Syntax Graphs for Domain Specific Languages.

Применение графов для кодирования дублирующих подвыражений в EDSL.

- Jacques Carette, Oleg Kiselyov and Chung-chieh Shan. Finally Tagless, Partially Evaluated. Tagless Staged Interpreters for Simpler Typed Languages.

Построение расширяемого синтаксиса с помощью классов типов.

- Wouter Sweistra. Data types a la carte.

Построение расширяемых типов. В этой статье и выше под словом “расширяемый” понимается возможность добавления к типу новых конструкторов без перекомпиляции старых.

И все-все-все

Если вдруг у вас возникли вопросы по Haskell, и рядом с вами не оказалось того, кто мог бы на них ответить, и в книгах нет ответа, вы можете спросить у сообщества Haskell, в `haskell-cafe`, там вам быстро и с радостью ответят:

<http://www.haskell.org/mailman/listinfo/haskell-cafe>

Сообщество Haskell славится радушием и терпимостью к начинающим. Там много информации о выпусках новых библиотек, конференциях, обучающих программах и просто разговоры о том-о-сём.

Также стоит отметить журнал *Monad.Reader*:

<http://themonadreader.wordpress.com/>

Обзор Hackage

Число пакетов, загруженных на Hackage, уже перевалило за 2000. В Hackage легко заблудиться. Очень часто не разберёшься какой из пакетов выбрать. К тому же многие из них заброшены или просто не подходят для использования в серьёзных приложениях. Но среди них есть и очень хорошие пакеты. Некоторые из них включены в **Haskell Platform**. Ниже приведён тематический обзор наиболее популярных пакетов.

Стандартные библиотеки

Все приведённые в этом подразделе библиотеки включены в **Haskell Platform**.

Полный список библиотек для **Haskell Platform** можно посмотреть на сайте <http://lambda.haskell.org/hp-tmp/docs>.

- **Начало-всех-начал:** `base`

Библиотека включает в себя все стандартные определения, например модули `Prelude`, `Data.List`, `Control.Monad` и многие другие.

- **Стандартные монады:** `transformers`, `mtl`

Включает монады `State`, `Writer`, `Reader` и другие.

- **Контейнеры:** `containers`

Ассоциативные массивы, множества, последовательности, деревья.

- **Массивы:** `array`

- **Графы:** `fgl`

- **Архиваторы:** `zlib`

- **Вычисление по значению:** `deepseq`

Обычная функция `seq`, позволяет привести данное выражение к слабой заголовочной нормальной форме, если нам всё же необходимо вычислить значение полностью, мы можем воспользоваться функцией `deepseq` из одноимённой библиотеки.

- **Параллельное программирование:** `stm` и `parallel`
- **Временная арифметика, календарь:** `time`
- **Парсинг:** `parsec`
- **Регулярные выражения:** `regex-base`, `regex-posix`
- **Построение структурированного текста:** `pretty`
- **Тестирование программ:** `HUnit`, `QuickCheck`
- **Управление файловой системой:** `directory`
- **Работа с путями к файлам/директориям:** `filepath`
- **Сетевые библиотеки:** `network`, `HTTP`, `cgi`.
- **Зд Графика:** `OpenGL`, `GLUT`.
- **Монадные трансформеры:** `transformers`

Мы не коснулись этой темы, но вот краткое пояснение: монадные трансформеры позволяют комбинировать несколько монад. Например, если нам нужно использовать чтение-запись в файл совместно с изменяемым состоянием.

Эффективные типы данных

- **Списки:** `dlist` – эффективное объединение списков.

Если вы часто пользуетесь операцией `++`, то необходимо заботиться о том, чтобы скобки всегда группировались вправо. Как в `a++(b++(c++d))`. Иначе время объединения из линейного превратится в квадратичное. Библиотека `dlist` предоставляет специальный тип списков, для которых не важно

как группируются скобки при объединении. Время объединения всегда будет линейным.

- **Строки:** `bytestring`

Если ваша программа загружена обработкой строк, и работает слишком медленно, рассмотрите вариант перехода со стандартных строк на тип `ByteString`, это может увеличить быстродействие на порядок.

- **Текст:** `text` или `utf8-string`

Работа с текстом в формате Unicode. Часто проблемы возникают при необходимости обработки русского текста закодированного в Unicode. Для решения этой проблемы можно воспользоваться одной из этих библиотек.

- **Двоичные данные:** `binary` или `cereal` –

Сериализация/десериализация данных.

- **Случайные числа:** `mersenne-random-pure64`

Эффективный генератор случайных чисел.

- **Ввод-вывод:** `iteratee`

Эффективная реализация ввода-вывода. Если вам нужно читать или писать данные из большого числа файлов, эта библиотека может существенно помочь.

- **Контейнеры:** `unordered-containers`

Альтернатива стандартной библиотеке `containers`. Эффективные типы `Map` и `Set`.

- **Последовательности:** `fingertree`, `seq`

Используются для работы с очередями различного типа.

- **Массивы:** `vector`

Эффективный тип для представления массивов. Замена стандартному типу `Data.Array`.

- **Самые эффективные изменяемые хэш-таблицы:** `hashtables`

- Матрицы: `hmatrix`, `repa`

Разработка программ

- Тестирование, проверка инвариантов: `QuickCheck`
- Оценка быстродействия: `criterion`
- Просмотр Core в человеческом виде: `ghc-core`
- Настройка сборки мусора: `ghc-gc-tune`
- Трассировка программ: `hat`

И все-все-все

- Парсинг: `parsec` или `attoparsec`
- Языки разметки: `pandoc`, `xhtml`, `tag soup`, `blaze-html`, `html`
- XML: `xml`, `HaXml`
- JSON: `json`, `aeson`
- Web: `happstack`, `snap`, `yesod`, `haskell`
- Сетевые библиотеки: `network`, `HTTP`, `cgi`, `curl`
- Графика: `diagrams`, `gnuplot`, `SDL`
- 3д графика: `OpenGL`, `GLFW`, `GLUT`
- Базы данных: `HDBC`
- Встраиваемые приложения реального времени с жёсткими ограничениями: `atom`
- GUI: `wxHaskell`, `gtk2hs`
- Оценка производительности программ: `criterion`
- Статистика: `statistics`
- Парсинг и генерация кода Haskell: `haskell-src-extensions`
- FRP: `reactive`, `reactive-banana`, `yampa`
- Линейная алгебра: `vector-space`, `hmatrix`

Места

Где культивируется Haskell?

Университеты

Посмотрим на университеты, в которых Haskell преподают, развивают и применяют:

- Британия: Эдинбург, Ноттингем, Оксфорд (лаборатория информатики), Глазго.
- Америка: Йельский, Коннектикут, Техас, Оклахома, Портленд, Канзас
- Нидерланды: Утрехт
- Швеция: Технологический Чалмерса, Гёттинген.
- Австралия: Новый Южный Уэльс, Западной Австралии
- и другие, полный список на http://www.haskell.org/haskellwiki/Haskell_in_education.

Компании

- Microsoft Research – разрабатывают GHC.
- Galios – ведут исследования и решают практические задачи на ФП-языках, особенно на Haskell.
- Well-Typed – решают практические задачи, консультируют и всё на Haskell. Также занимаются организацией Haskell-слётов, поддержкой стандартных библиотек.
- и другие, полный список на http://www.haskell.org/haskellwiki/Haskell_in_industry