Московский Авиационный Институт

(Национальный Исследовательский Университет)

**Факультет информационных технологии и прикладной математики**

Кафедра вычислительной математики и программирования

**КУРСОВАЯ РАБОТА**

**по курсу**

**“Информатика”**

**II семестр**

**«Функциональное программирование. Язык программирования Haskell»**

**Реферат.**

Студент: Дубинин А. О.

Группа: 08-103Б, № по списку 6

Руководитель: Никулин С.П.,  
 доцент каф.806

Оценка:\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Дата:\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Подпись:\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Москва, 2018**

Содержание

[Введение 3](#_Toc508783421)

[Функциональное программирование 4](#_Toc508783422)

[История 5](#_Toc508783423)

[Концепции 6](#_Toc508783424)

[Функции высших порядков 6](#_Toc508783425)

[Чистые функции 6](#_Toc508783426)

[Рекурсия 7](#_Toc508783427)

[ФП в нефункциональных языках 8](#_Toc508783428)

[Стили программирования 9](#_Toc508783429)

[Особенности 10](#_Toc508783430)

[Сильные стороны 10](#_Toc508783431)

[Недостатки 11](#_Toc508783432)

[Характеристики языка Haskell 13](#_Toc508783433)

[Реализации языка 14](#_Toc508783434)

[Примеры 17](#_Toc508783435)

[Калькулятор 17](#_Toc508783436)

[Числа Фибоначчи 17](#_Toc508783437)

[Простые числа 19](#_Toc508783438)

[Приложения, написанные на языке Haskell 21](#_Toc508783439)

[Вывод 22](#_Toc508783440)

[Используемая литература 23](#_Toc508783441)

# Введение

Функциональное программирование — раздел дискретной математики и парадигма программирования, в которой процесс вычисления трактуется как вычисление значений функций в математическом понимании последних (в отличие от функций как подпрограмм в процедурном программировании).

Haskell (hæskəl) — стандартизированный чистый функциональный язык программирования общего назначения. Является одним из самых распространённых языков программирования с поддержкой отложенных вычислений. Система типов — полная, сильная, статическая, с автоматическим выводом типов, основанная на системе типов Хиндли — Милнера. Поскольку язык функциональный, то основная управляющая структура — это функция.

# Функциональное программирование

Функциональное программирование противопоставляется парадигме императивного программирования, которая описывает процесс вычислений как последовательное изменение состояний (в значении, подобном таковому в теории автоматов). При необходимости, в функциональном программировании вся совокупность последовательных состояний вычислительного процесса представляется явным образом, например, как список.

Функциональное программирование предполагает обходиться вычислением результатов функций от исходных данных и результатов других функций, и не предполагает явного хранения состояния программы. Соответственно, не предполагает оно и изменяемость этого состояния (в отличие от императивного, где одной из базовых концепций является переменная, хранящая своё значение и позволяющая менять его по мере выполнения алгоритма).

На практике отличие математической функции от понятия «функции» в императивном программировании заключается в том, что императивные функции могут опираться не только на аргументы, но и на состояние внешних по отношению к функции переменных, а также иметь побочные эффекты и менять состояние внешних переменных. Таким образом, в императивном программировании при вызове одной и той же функции с одинаковыми параметрами, но на разных этапах выполнения алгоритма, можно получить разные данные на выходе из-за влияния на функцию состояния переменных. А в функциональном языке при вызове функции с одними и теми же аргументами мы всегда получим одинаковый результат: выходные данные зависят только от входных. Это позволяет средам выполнения программ на функциональных языках кешировать результаты функций и вызывать их в порядке, не определяемом алгоритмом и распараллеливать их без каких-либо дополнительных действий со стороны программиста (что обеспечивают функции без побочных эффектов — чистые функции[⇨]).

Лямбда-исчисление являются основой для функционального программирования, многие функциональные языки можно рассматривать как «надстройку» над ними[1].

# История

Лямбда-исчисление стало теоретической базой для описания и вычисления функций. Являясь математической абстракцией, а не языком программирования, оно составило базис почти всех языков функционального программирования на сегодняшний день. Сходное теоретическое понятие, комбинаторная логика, является более абстрактным, нежели λ-исчисления и было создано раньше. Эта логика используется в некоторых эзотерических языках, например в Unlambda. И λ-исчисление, и комбинаторная логика были разработаны для более ясного и точного описания принципов и основ математики.

**Haskell**

Haskell принадлежит к семейству языков ML. Непосредственно на него оказал большое влияние язык Миранда, разработанный в 1985 году Дэвидом Тёрнером. Миранда была первым чистым функциональным языком, имевшим коммерческую поддержку, и была относительно популярна в 1980-х годах, но оставалась собственническим программным обеспечением. Это затрудняло развитие и исследования возможностей ленивого функционального программирования, поэтому буквально за пару лет появилось более десятка схожих языков. Чтобы объединить усилия разных разработчиков в 1987 году на конференции по функциональным языкам программирования и компьютерной архитектуре в Орегоне (FPCA’87) было решено создать комитет для разработки открытого стандарта.

В 1990 году была предложена первая версия языка, Haskell 1.0. В дальнейшем работа комитета продолжилась, и в 1999 году был опубликован «The Haskell 98 Report»[5], который стал стабильным стандартом языка на много лет. Язык, однако, продолжал бурно развиваться, компилятор GHC был фактическим стандартом в отношении новых возможностей.

Разработка новых версий языка идёт открыто, этот процесс получил название Haskell’[6] (Haskell Prime [ˈhæskəl praɪm], «Хаскелл-штрих»). Все желающие могут выдвигать свои предложения к обсуждению, предложения обсуждаются в течение года, комитет отбирает и объявляет предложения, которые готов принять, формируется новый комитет и к концу года готовится новая версия языка. Таким образом, новые версии языка теперь могут появляться каждый год. Планируется объявлять некоторые ревизии «значительными» и поддерживать такие ревизии на протяжении длительного времени.

Версия языка Haskell 2010 — была объявлена в конце 2009 г[7], но последней «значительной» версией (стандартом) остаётся Haskell 98.

# Концепции

Некоторые концепции и парадигмы специфичны для функционального программирования и в основном чужды императивному программированию (включая объектно-ориентированное программирование). Тем не менее, языки программирования обычно представляют собой гибрид нескольких парадигм программирования, поэтому «большей частью императивные» языки программирования могут использовать какие-либо из этих концепций.

## Функции высших порядков

Функции высших порядков — это такие функции, которые могут принимать в качестве аргументов и возвращать другие функции.[13] Математики такую функцию чаще называют оператором, например, оператор взятия производной или оператор интегрирования.

Функции высших порядков позволяют использовать карринг — преобразование функции от пары аргументов в функцию, берущую свои аргументы по одному. Это преобразование получило своё название в честь Х. Карри.

Следующий исходный код, написанный на Haskell, содержит функцию высшего порядка, которая в качестве первого аргумента принимает функцию. В результате на экран будет выведено «100» (результат вычисления (7+3)×(7+3)).

f func x **=** (func x) \* (func x)

main **=** putStrLn (show (f (+3) 7))

## Чистые функции

Чистыми называют функции, которые не имеют побочных эффектов ввода-вывода и памяти (они зависят только от своих параметров и возвращают только свой результат). Чистые функции обладают несколькими полезными свойствами, многие из которых можно использовать для оптимизации кода:

* Если результат чистой функции не используется, её вызов может быть удален без вреда для других выражений.
* Результат вызова чистой функции может быть мемоизирован, то есть сохранен в таблице значений вместе с аргументами вызова. Если в дальнейшем функция вызывается с этими же аргументами, её результат может быть взят прямо из таблицы, не вычисляясь (иногда это называется принципом прозрачности ссылок). Мемоизация, ценой небольшого расхода памяти, позволяет существенно увеличить производительность и уменьшить порядок роста некоторых рекурсивных алгоритмов.
* Если нет никакой зависимости по данным между двумя чистыми функциями, то порядок их вычисления можно поменять или распараллелить (говоря иначе вычисление чистых функций удовлетворяет принципам thread-safe)
* Если весь язык не допускает побочных эффектов, то можно использовать любую политику вычисления. Это предоставляет свободу компилятору комбинировать и реорганизовывать вычисление выражений в программе (например, исключить древовидные структуры).

Хотя большинство компиляторов императивных языков программирования распознают чистые функции и удаляют общие подвыражения для вызовов чистых функций, они не могут делать это всегда для предварительно скомпилированных библиотек, которые, как правило, не предоставляют эту информацию. Некоторые компиляторы, такие как gcc, в целях оптимизации предоставляют программисту ключевые слова для обозначения чистых функций[14]. Fortran 95 позволяет обозначать функции как «pure» (чистые)[15].

## Рекурсия

В функциональных языках цикл обычно реализуется в виде рекурсии. Строго говоря, в функциональной парадигме программирования нет такого понятия, как цикл. Рекурсивные функции вызывают сами себя, позволяя операции выполняться снова и снова. Для использования рекурсии может потребоваться большой стек, но этого можно избежать в случае хвостовой рекурсии. Хвостовая рекурсия может быть распознана и оптимизирована компилятором в код, получаемый после компиляции аналогичной итерации в императивном языке программирования.[16] Стандарты языка Scheme требуют распознавать и оптимизировать хвостовую рекурсию. Оптимизировать хвостовую рекурсию можно путём преобразования программы в стиле использования продолжений при её компиляции, как один из способов.

Рекурсивные функции можно обобщить с помощью функций высших порядков, используя, например, катаморфизм и анаморфизм (или «свертка» и «развертка»). Функции такого рода играют роль такого понятия как цикл в императивных языках программирования

Следующий пример показывает рекурсию языка Haskell при реализации функции для вычисления факториала:

fac **::** Integer **->** Integer

fac 0 **=** 1

fac n | n > 0 **=** n \* fac (n - 1)

Большая часть исходного кода на языке Haskell походит на математическую нотацию в аспектах синтаксиса и использования, например, вышеприведённый пример можно переписать в виде

fac n **=** product [1..n]

что соответствует математическому определению факториала.

## ФП в нефункциональных языках

Принципиально нет препятствий для написания программ в функциональном стиле на языках, которые традиционно не считаются функциональными, точно так же, как программы в объектно-ориентированном стиле можно писать на структурных языках. Некоторые императивные языки поддерживают типичные для функциональных языков конструкции, такие как функции высшего порядка и списковые включения (list comprehensions), что облегчает использование функционального стиля в этих языках. Примером может быть функциональное программирование на Python. Другим примером является язык Ruby, который имеет возможность создания как lambda-объектов, так и возможность организации анонимных функций высшего порядка через блок с помощью конструкции yield.

В языке C указатели на функцию в качестве типов аргументов могут быть использованы для создания функций высшего порядка. Функции высшего порядка и отложенная списковая структура реализованы в библиотеках С++. В языке C# версии 3.0 и выше можно использовать λ-функции для написания программы в функциональном стиле. В сложных языках, типа Алгол-68, имеющиеся средства метапрограммирования (фактически, дополнения языка новыми конструкциями) позволяют создать специфичные для функционального стиля объекты данных и программные конструкции, после чего можно писать функциональные программы с их использованием.

# Стили программирования

Императивные программы имеют склонность акцентировать последовательности шагов для выполнения какого-то действия, а функциональные программы к расположению и композиции функций, часто не обозначая точной последовательности шагов. Простой пример двух решений одной задачи (используется один и тот же язык Python) иллюстрирует это.

*# императивный стиль*

target = [] *# создать пустой список*

**for** item **in** source\_list: *# для каждого элемента исходного списка*

trans1 = G(item) *# применить функцию G()*

trans2 = F(trans1) *# применить функцию F()*

target.append(trans2) *# добавить преобразованный элемент в список*

Функциональная версия выглядит по-другому:

*# функциональный стиль*

*# языки ФП часто имеют встроенную функцию compose()*

compose2 = **lambda** A, B: **lambda** x: A(B(x))

target = map(compose2(F, G), source\_list)

В отличие от императивного стиля, описывающего шаги, ведущие к достижению цели, функциональный стиль описывает математические отношения между данными и целью.

Более точно, существует четыре ступени развития функционального стиля, в порядке убывания роли данных в программах:

* Рефал (для этой категории, представленной единственным языком, нет общепринятого названия);
* Аппликативные (Лисп, ML, Tcl, Rebol);
* Комбинаторные (APL/J/K, FP[en]/FL[en]);
* Бесточечные (чистые конкатенативные) (Joy, Cat, Factor, подмножество PostScript).

**Haskell** позволяет писать и в аппликативном, и в комбинаторном, и в бесточечном стилях.

# Особенности

Основной особенностью функционального программирования, определяющей как преимущества, так и недостатки данной парадигмы, является то, что в ней реализуется *модель вычислений без состояний*. Если императивная программа на любом этапе исполнения имеет состояние, то есть совокупность значений всех переменных, и производит побочные эффекты, то чисто функциональная программа ни целиком, ни частями состояния не имеет и побочных эффектов не производит. То, что в императивных языках делается путём присваивания значений переменным, в функциональных достигается путём передачи выражений в параметры функций. Непосредственным следствием становится то, что чисто функциональная программа не может изменять уже имеющиеся у неё данные, а может лишь порождать новые путём копирования и/или расширения старых. Следствием того же является отказ от циклов в пользу рекурсии.

## Сильные стороны

1. **Повышение надёжности кода**

Привлекательная сторона вычислений без состояний — повышение надёжности кода за счёт чёткой структуризации и отсутствия необходимости отслеживания побочных эффектов. Любая функция работает только с локальными данными и работает с ними всегда одинаково, независимо от того, где, как и при каких обстоятельствах она вызывается. Невозможность мутации данных при пользовании ими в разных местах программы исключает появление труднообнаруживаемых ошибок (таких, например, как случайное присваивание неверного значения глобальной переменной в императивной программе).

1. **Удобство организации модульного тестирования**

Поскольку функция в функциональном программировании не может порождать побочные эффекты, менять объекты нельзя как внутри области видимости, так и снаружи (в отличие от императивных программ, где одна функция может установить какую-нибудь внешнюю переменную, считываемую второй функцией). Единственным эффектом от вычисления функции является возвращаемый ей результат, и единственный фактор, оказывающий влияние на результат — это значения аргументов.

Таким образом, имеется возможность протестировать каждую функцию в программе, просто вычислив её от различных наборов значений аргументов. При этом можно не беспокоиться ни о вызове функций в правильном порядке, ни о правильном формировании внешнего состояния. Если любая функция в программе проходит модульные тесты, то можно быть уверенным в качестве всей программы. В императивных программах проверка возвращаемого значения функции недостаточна: функция может модифицировать внешнее состояние, которое тоже нужно проверять, чего не нужно делать в функциональных программах.

1. **Возможности оптимизации при компиляции**

Традиционно упоминаемой положительной особенностью функционального программирования является то, что оно позволяет описывать программу в так называемом «декларативном» виде, когда жесткая последовательность выполнения многих операций, необходимых для вычисления результата, в явном виде не задаётся, а формируется автоматически в процессе вычисления функций. Это обстоятельство, а также отсутствие состояний даёт возможность применять к функциональным программам достаточно сложные методы автоматической оптимизации.

1. **Возможности параллелизма**

Ещё одним преимуществом функциональных программ является то, что они предоставляют широчайшие возможности для автоматического распараллеливания вычислений. Поскольку отсутствие побочных эффектов гарантировано, в любом вызове функции всегда допустимо параллельное вычисление двух различных параметров — порядок их вычисления не может оказать влияния на результат вызова.

## Недостатки

Недостатки функционального программирования вытекают из тех же самых его особенностей. Отсутствие присваиваний и замена их на порождение новых данных приводят к необходимости постоянного выделения и автоматического освобождения памяти, поэтому в системе исполнения функциональной программы обязательным компонентом становится высокоэффективный сборщик мусора. Нестрогая модель вычислений приводит к непредсказуемому порядку вызова функций, что создает проблемы при вводе-выводе, где порядок выполнения операций важен. Кроме того, очевидно, функции ввода в своем естественном виде (например, getchar из стандартной библиотеки языка C) не являются чистыми, поскольку способны возвращать различные значения для одних и тех же аргументов, и для устранения этого требуются определенные ухищрения.

Для преодоления недостатков функциональных программ уже первые языки функционального программирования включали не только чисто функциональные средства, но и механизмы императивного программирования (присваивание, цикл, «неявный PROGN» были уже в Лиспе). Использование таких средств позволяет решить некоторые практические проблемы, но означает отход от идей (и преимуществ) функционального программирования и написание императивных программ на функциональных языках. В чистых функциональных языках эти проблемы решаются другими средствами, например, в языке Haskell ввод-вывод реализован при помощи монад — нетривиальной концепции, позаимствованной из теории категорий.

## 

# Характеристики языка Haskell

В качестве основных характеристик языка Haskell можно выделить следующие:

* недопустимость побочных эффектов (чистота языка); возможность писать программы с побочными эффектами без нарушения парадигмы функционального программирования с помощью монад;
* статическая сильная полная типизация с автоматическим выведением типов, основанная на типизации Хиндли — Милнера;
* функции высшего порядка, в том числе лямбда-абстракции;
* частичное применение;
* ленивые вычисления (lazy evaluation);
* сопоставление с образцом (англ. pattern matching), функциональные образцы, охраняющие выражения (guards);
* параметрический полиморфизм и его объединение с ad hoc полиморфизмом в единую модель посредством классов типов;
* алгебраические типы данных, в том числе псевдобесконечные (за счёт ленивости);
* генераторы списков (list comprehensions);
* возможность интеграции с программами, реализованными на императивных языках программирования посредством открытых интерфейсов (стандартное расширение языка Foreign Function Interface (рус.)англ.[8]).

С момента принятия последнего стандарта языка (Haskell98) прошло много времени, и с тех пор ведущие реализации языка (ghc и hugs) были расширены множеством дополнительных возможностей:

* параметрический полиморфизм высших рангов за счёт квантификации переменных типа (вплоть до импредикативного) — естественно, исключающая выведение типов.
* Функциональные зависимости (FD, functional dependencies)

# Реализации языка

Есть несколько реализаций языка Haskell Некоторые реализации ориентированы на практическое применение, в то время как другие — представляют прежде всего академический интерес.

**Компиляторы и интерпретаторы**

Наиболее популярен на практике оптимизирующий компилятор GHC, который создаёт быстрый код и позволяет использовать многие расширения языка. GHC может оптимизировать как скорость, так и компактность программ, способен создавать многозадачный и параллелизованный код. В комплекте с компилятором GHC поставляется также интерактивная среда программирования GHCi со встроенным отладчиком. GHC работает в Windows, MacOS X и на нескольких юникс-подобных платформах (Linux, \*BSD, Solaris). Именно GHC является стандартным компилятором в Haskell Platform, и именно на нём в первую очередь тестируются все новые библиотеки.

Другая популярная реализация языка — интерпретатор HUGS. Он написан на Си, имеет малый размер дистрибутива и работает практически на всех платформах. HUGS предоставляет интерактивную среду программирования, но может также запускать программы на Haskell в стиле скриптовых языков. Пользователи Windows могут использовать графическую интерактивную среду WinHugs. Поскольку HUGS — это интерпретатор, то программы, запущенные в нём, выполняются медленнее, чем код, созданный большинством компиляторов языка Haskell. HUGS часто рекомендуют в качестве среды для изучения языка. HUGS полностью поддерживает стандарт языка Haskell 98, а также некоторые наиболее популярные расширения языка.

Другие известные реализации:

* nhc98 — быстрый компилятор, поддерживающий стандарт Haskell 98, написанный на Haskell 98; доступен для всех юникс-подобных платформ, включая MacOS X и Windows/cygwin
* YHC (York Haskell Compiler) — форк nhc98, ставящий целью быть более переносимым и эффективным, поддерживает отладчик Hat; генерирует промежуточный байт-код, который можно использовать для генерации кода на других языках программирования
* UHC (Utrecht Haskell Compiler) — компилятор, поддерживающий Haskell 98 почти полностью, а также некоторые расширения; поддерживает глобальный анализ программ; имеет несколько кодогенераторов, в том числе почти полностью функциональный генератор JVM-кода, кодогенераторы для LLVM и CLR в состоянии разработки; работает на юникс-подобных системах (включая MacOS X и Windows/cygwin)
* HBI и HBC — интерпретатор и компилятор, поддерживающие стандарт Haskell 98 с некоторыми расширениями; точно работает в Linux, поддерживается слабо
* LHC (The Luxurious LHC Haskell Optimization System) — альтернативный кодогенератор для GHC, поддерживающий глобальную низкоуровневую оптимизацию программы в целом
* jhc — экспериментальный компилятор для исследования возможностей глобальной оптимизации программ
* Yale Haskell — ранняя реализация языка Haskell на Lisp

**Haskell Platform**

В 2009 году сформировалась концепция Haskell Platform — стандартного дистрибутива языка, включающего кроме компилятора (GHC), также дополнительный инструментарий (систему сборки и развёртывания пакетов Cabal) и набор популярных библиотек.

Сейчас Haskell Platform — это рекомендованный базовый дистрибутив для разработчиков. Готовые сборки Haskell Platform доступны для Windows, MacOS X и ряда дистрибутивов Linux.

**Альтернативные целевые платформы**

Большинство компиляторов языка Haskell создают непосредственно машинный код для используемой платформы, но есть несколько проектов, позволяющих производить код для виртуальных машин или генерировать код на других языках программирования. Степень зрелости и уровень поддержки подобных проектов сильно разнится.

* LLVM:
  + патч для GHC, добавляющий генератор кода для LLVM, в настоящее время включен в компилятор GHC; использование LLVM позволяет достичь большей производительности на ряде задач.
  + альтернативный кодогенератор в UHT (работоспособен частично)
* виртуальная машина Java (JVM):
  + LambdaVM — патч для GHC, добавляющий генератор байткода JVM
  + jvm-bridge — мост между Haskell и JVM (для GHC)
  + jaskell — генератор байт-кода JVM (вероятно заброшен)
  + альтернативный кодогенератор в UHC (работоспособен почти полностью)
* Common Language Runtime (.Net):
  + прототип генератора MSIL-кода для компилятора EHT/UHT
  + некоторые проекты по поддержке CLR упоминаются в GHC FAQ, на начало 2010 года их состояние не ясно
* JavaScript:
  + кодогенератор JavaScript для GHC— патч для GHC позволяющий компилировать программы в JavaScript-код
  + Ycr2Js — конвертер байт-кода компилятора YHC в JavaScript
  + Fay — компилятор подмножества Haskell в JavaScript

Несколько интересных целевых платформ доступны при использовании компилятора YHC, в частности существуют интерпретатор байт-кода YHC на Python и конвертер байт-кода YHC в Erlang Core, но эти разработки пока ещё экспериментальны. Также существуют реализации подмножеств языка на разных целевых платформах.

**Расширения языка**

* макрорасширение с контролем типов (Template Haskell);
* объектно-ориентированное программирование (O’Haskell, Haskell++ и Mondrian).

Расширения реализаций языка (относится к GHC):

* развитие системы типизации;
* многопоточность;
* параллельные вычисления;
* распределённые вычисления;
* транзакционная память.

# Примеры

## Калькулятор

Простейший калькулятор для вычисления выражений в обратной польской записи может быть определён на языке Haskell при помощи одной функции:

calc **::** String **->** Float

calc **=** head . foldl f [] . words

**where**

f **::** [Float] **->** String **->** [Float]

f (x:y:zs) "+" **=** (y + x):zs

f (x:y:zs) "-" **=** (y - x):zs

f (x:y:zs) "\*" **=** (y \* x):zs

f (x:y:zs) "/" **=** (y / x):zs

f (x:y:zs) "FLIP" **=** y:x:zs

f (x:zs) "ABS" **=** (abs x):zs

f xs y **=** read y : xs

Исходная строка со входным выражением тут разбивается стандартной функцией words на список слов — строк между пробельными символами — который обрабатывается функцией левосторонней свёртки (foldl) слева направо по одному слову с помощью функции f, которая поддерживает рабочий список прочитываемых чисел и промежуточных значений (поначалу [] — пустой список) и интерпретирует каждое входное слово как обозначение арифметической функции или как число, в ходе вычисления ею окончательного значения выражения (которое будет первым оставшимся значением в рабочем списке по окончании обработки списка слов входного выражения, так что его можно достать оттуда с помощью стандартной функции head).

Здесь (.) есть оператор композиции функций, (f . g) x = f (g x). Например,

\*Main> calc "1 2 3 + 4 \* - ABS"

19.0

## Числа Фибоначчи

Другой пример показывает способ вычисления бесконечного списка чисел Фибоначчи за линейное время:

fibs **=** 0 : 1 : zipWith (+) fibs (tail fibs)

Бесконечный список здесь определен при помощи механизма корекурсии — последующие значения списка здесь задаются на основе предыдущих, с начальными 0 и 1 в качестве первых двух элементов списка, и выражением-генератором zipWith (+) fibs (tail fibs), вычисляющим все элементы начиная с третьего на основании предшествующих двух, через стандартную функцию zipWith (+) которая суммирует попарно элементы двух своих входных списков.

Это определение является примером применения механизма ленивых вычислений, который является важнейшей частью языка Haskell. Для понимания того, как это определение работает, можно рассмотреть вычисление первых семи чисел Фибоначчи с его помощью:

fibs = 0 : 1 : 1 : 2 : 3 : 5 : 8 : ...

+ + + + + +

tail fibs = 1 : 1 : 2 : 3 : 5 : 8 : ...

= = = = = =

zipWith (+) = 1 : 2 : 3 : 5 : 8 : ...

fibs = 0 : 1 : 1 : 2 : 3 : 5 : 8 : ...

То же самое может быть записано также при использовании определителей списков,

fibs **=** 0 : 1 : [a + b | (a,b) **<-** zip fibs (tail fibs)]

или расширения языка Haskell, реализованного в компиляторе GHC (параллельных определителей списков, англ. *parallel list comprehensions*):

fibs **=** 0 : 1 : [a + b | a **<-** fibs

| b **<-** tail fibs]

или с помощью напрямую самореферентной генерирующей функции:

fibs **=** 0 : 1 : next fibs

**where**

next (a: t@(b:**\_**)) **=** (a+b) : next t

## Простые числа

В этих примерах показано, как можно использовать списочные выражения (генераторы списков). Реализация нахождения всех простых чисел обычным путём (проверка каждого числа на простоту):

*-- общее определение (все натуральные числа > 1, которые являются простыми)*

primeNums **=** 2 : [n | n **<-** [3..], isPrime n]

*-- Число простое, если у него нет (простых) делителей*

isPrime n **=** foldr (\p r**->** p\*p>n || (rem n p /= 0 && r)) True primeNums

или с помощью решета Эратосфена, в прототипичном, неэффективном варианте,

primes **=** (map head . scanl minus [2..] . map (\p **->** [p, p+p..])) primes

или эффективно, с предварительным каскадным объединением потоков составных чисел:

primes **=** 2 : \_Y ((3:) . minus [5,7..] . unionAll . map (\p **->** [p\*p, p\*p+2\*p..]))

**where**

\_Y g **=** g (\_Y g)

unionAll ((x:xs):t) **=** x : union xs (unionAll (pairs t))

pairs ((x:xs):ys:t) **=** (x : union xs ys) : pairs t

или посегментно, массивами,

**import** **Data.Array**

**import** **Data.List** (tails, inits)

ps **=** 2 : [n | (r:q:**\_**, px) **<-** (zip . tails . (2:) . map (^2)) ps (inits ps),

(n,True) **<-** assocs (

accumArray (\**\_** **\_** **->** False) True (r+1,q-1)

[(m,()) | p **<-** px,

**let** s **=** div (r+p) p \* p, m **<-** [s,s+p..q-1]] )]

с использованием канонических функций minus, union (из модуля Data.List.Ordered):

union (x:xs) (y:ys) **=** **case** compare x y **of**

LT **->** x : union xs (y:ys)

EQ **->** x : union xs ys

GT **->** y : union (x:xs) ys

union a b **=** a ++ b

minus (x:xs) (y:ys) **=** **case** compare x y **of**

LT **->** x : minus xs (y:ys)

EQ **->** minus xs ys

GT **->** minus (x:xs) ys

minus a b **=** a

# Приложения, написанные на языке Haskell

Мозаичный оконный менеджер Xmonad для X Window System целиком написан на языке Haskell. Darcs — распределённая система управления версиями с рядом уникальных возможностей — написана на Haskell. Первая реализация компилятора и интерпретатора языка Perl 6, Pugs, была написана на Haskell за несколько месяцев. Компилятор GHC часто выступает экспериментальной площадкой для проверки новых возможностей функционального программирования и оптимизации.

**Коммерческие приложения**

Haskell всё чаще используется в коммерческой среде. Этому способствует и принятая в сообществе традиция выпускать библиотеки под либеральными лицензиями (более 70 % свободно доступных библиотек распространяются на условиях лицензий BSD, MIT или являются общественным достоянием).

Вот примеры некоторых коммерческих приложений, написанных на Haskell: Bluespec SystemVerilog, встраиваемый язык проектирования и верификации полупроводниковых схем, является расширением языка Haskell. Cryptol, коммерческий язык для разработки и проверки криптографических алгоритмов, реализован на Haskell. Примечательно, что первое формально верифицированное микроядро seL4 было тоже написано на Haskell.

Haskell активно применяется в области финансового программирования, анализа рисков, в системах поддержки принятия решений. Haskell применяют разработчики генератора городских ландшафтов для игр и моделирования Gamr7. Есть примеры успешного применения языка для разработки частных информационных систем в коммерческих организациях (в том числе, в странах СНГ).

Это лишь не многие приложения, которые существуют на рынке.

# Вывод

Функциональное программирование позволяет выполнять многие задачи. Но для этого нужно владеть хорошей математической базой, ведь функциональное программирование тесно связанно с математикой, именно поэтому многим людям сложно вникнуть в синтаксис языков ФП.

Но умение писать на языке функционального стиля, позволяет выполнять некоторые задачи намного лучше и быстрее, чем, если бы мы их писали в классическом императивном стиле.

Язык же Haskell является одним из лучших языков поддерживающие основные концепции функционального программирования. Так же этот язык позволяет писать и в аппликативном, и в комбинаторном, и в бесточечном стилях.

# Используемая литература

* *Bryan O’Sullivan, John Goerzen, Don Stewart*. Real World Haskell — O’Reilly, 2008
* *Душкин Роман Викторович.* Функциональное программирование на языке Haskell.
* *А. Филд, П. Харрисон. Функциональное программирование. – М. : Мир, 1993.*
* *Kees Doets, Jan van Eijck*. «The Haskell Road to Logic, Maths and Programming».