Московский Инженерно-Физический Институт
(Государственный Университет)

Текст лекций по курсу

«Функциональное программирование»

Душкин Роман Викторович
darkus@yandex.ru

Москва, 2001

Лекция 1. «Вводная лекция»

Функциональное программирование ставит своей целью придать каждой программе простую математическую интерпретацию. Эта интерпретация должна быть независима от деталей исполнения и понятна людям, которые не имеют научной степени в предметной области.

Лоренс Паулсон

Прежде чем начать описание собственно функционального программирования, необходимо обратиться к истории программирования вообще. В 40-х годах XX века появились первые цифровые компьютеры, которые, как известно, программировались при помощи переключения различного рода тумблеров, проводков и кнопок. Число таких переключений достигало порядка нескольких сотен и неумолимо росло с ростом сложности программ. Поэтому следующим шагом развития программирования стало создание всевозможных ассемблерных языков с простой мнемоникой.

Однако даже ассемблеры не могли стать тем инструментом, которым смогли бы пользоваться обыкновенные люди, т.к. мнемокоды все еще оставались слишком сложными, тем более что всякий ассемблер был жёстко связан с архитектурой, на которой он исполнялся. Таким образом, следующим шагом после ассемблера стали так называемые императивные языки высокого уровня (BASIC, Pascal, C, Ada и прочие, включая объектно-ориентированные). Императивными такие языки были названы по той простой причине, что главным их свойством является ориентированность, в первую очередь, на последовательное исполнение инструкций оперирующих с памятью (т.е. присваиваний) и итеративные циклы. Вызовы функций и процедур, даже рекурсивные, не избавляли такие языки от явной императивности (предписания).

Возвращаясь к функциональному программированию... Краеугольным камнем в парадигме функционального программирования, как будет показано далее, является функция. Если вспомнить историю математики, то можно оценить возраст понятия «функция». Ему уже около четырёхсот лет, и математика придумала бесчисленное множество теоретических и практических аппаратов для оперирования функциями, начиная от обыкновенных операций дифференцирования и интегрирования, заканчивая заумными функциональными анализами, теориями нечётких множеств и функций комплексных переменных.

Математические функции выражают связь между параметрами (входом) и результатом (выходом) некоторого процесса. Так как вычисление — это тоже процесс, имеющий вход и выход, функция является вполне подходящим и адекватным средством описания вычислений. Именно этот простой принцип положен в основу функциональной парадигмы и функционального стиля программирования. Функциональная программа представляет собой набор определений функций. Функции определяются через другие функции или рекурсивно — через самих себя. В процессе выполнения программы функции получают параметры, вычисляют и возвращают результат, в случае необходимости вычисляя значения других функций. Программируя на функциональном языке, программист не должен опи-

сывать порядок вычислений. Ему необходимо просто описать желаемый результат в виде системы функций.

Обращаясь к задачам курса, необходимо в первую очередь подчеркнуть, что функциональное программирование, равно как и логическое программирование, нашло большое применение в искуственном интеллекте и его приложениях. Поэтому здесь функциональное программирование рассматривается чрезвычайно скрупулёзно и со всеми возможными подробностями. Далее в этой лекции рассматривается история функционального программирования, свойства функциональных языков, решаемые задачи и некоторые справочные данные.

История функционального программирования

Как известно, теоретические основы императивного программирования были заложены ещё в 30-х годах XX века Аланом Тьюрингом и Джоном фон Нейманом. Теория, положенная в основу функционального подхода, также родилась в 20-х — 30-х годах. В числе разработчиков математических основ функционального программирования можно назвать Мозеса Шёнфинкеля (Германия и Россия) и Хаскелла Карри (Англия), разработавших комбинаторную логику, а также Алонзо Чёрча (США), создателя λ-исчисления.

Теория так и оставалась теорией, пока в начале 50-х прошлого века Джон МакКарти не разработал язык Lisp, который стал первым почти функциональным языком программирования и на протяжении многих лет оставался единственным таковым. Хотя Lisp всё ещё используется (как, например, и FORTRAN), он уже не удовлетворяет некоторым современным запросам, которые заставляют разработчиков программ взваливать как можно большую ношу на компилятор, облегчив тем самым свой непосильный труд. Необходимость в этом, конечно же, возникла из-за всё более возрастающей сложности программного обеспечения.

В связи с этим обстоятельством всё большую роль начинает играть типизация. В конце 70-х — начале 80-х годов XX века интенсивно разрабатываются модели типизации, подходящие для функциональных языков. Большинство этих моделей включали в себя поддержку таких мощных механизмов как абстракция данных и полиморфизм. Появляется множество типизированных функциональных языков: ML, Scheme, Hope, Miranda, Clean и многие другие. Вдобавок постоянно увеличивается число диалектов.

В результате вышло так, что практически каждая группа, занимающаяся функциональным программированием, использовала собственный язык. Это препятствовало дальнейшему распространению этих языков и порождало многочисленные более мелкие проблемы. Чтобы исправить ситуацию, объединенная группа ведущих исследователей в области функционального программирования решила воссоздать достоинства различных языков в новом универсальном функциональном языке. Первая реализация этого языка, названного Haskell в честь Хаскелла Карри, была создана в начале 90-х годов. В настоящее время действителен стандарт Haskell-98.

В первую очередь большинство функциональных языков программирования реализуются как интерпретаторы, следуя традициям Lisp’a. Интерпретаторы удобны для быстрой отладки программ, исключая длительную фазу компиляции, тем самым укорачивая обыч-

ный цикл разработки. Однако с другой стороны, интерпретаторы в сравнении с компиляторами обычно проигрывают по скорости выполнения в несколько раз. Поэтому помимо интерпретаторов существуют и компиляторы, генерирующие неплохой машинный код (например, Objective Caml) или код на C/C++ (например, Glasgow Haskell Compiler). Что показательно, практически каждый компилятор с функционального языка реализован на этом же самом языке.

В этом курсе для описания примеров функционального программирования будет использоваться либо некий абстрактный функциональный язык, приближенный к математической нотации, либо Haskell, бесплатные компиляторы которого можно скачать с сайта www.haskell.org.

Свойства функциональных языков

В качестве основных свойств функциональных языков кратко рассмотрим следующие:

- краткость и простота;
- строгая типизация;
- модульность;
- функции — это значения;
- чистота (отсутствие побочных эффектов);
- отложенные (ленивые) вычисления.

Краткость и простота

Программы на функциональных языках обычно намного короче и проще, чем те же самые программы на императивных языках. Сравним программы на C и на абстрактном функциональном языке на примере сортировки списка быстрым методом Хоара (пример, уже ставший классическим при описании преимуществ функциональных языков).

Пример 1. Быстрая сортировка Хоара на C.

```
void quickSort (int a[], int l, int r)
{
      int i = l;
      int j = r;
      int x = a[(l + r) / 2];
      do
      {
            while (a[i] < x) i++;
            while (x < a[j]) j--;
            if (i <= j)
            {
                  int temp = a[i];
                  a[i++] = a[j];
                  a[j--] = temp;
            }
      }
      while (i <= j);
      if (l < j) quickSort (a, l, j);
      if (i < r) quickSort (a, i, r);
}
```

Пример 2. Быстрая сортировка Хоара на абстрактном функциональном языке.

```
quickSort ([]) = []
quickSort ([h : t]) = quickSort (n | n ⊆ t, n <= h) + [h] + quickSort (n | n ⊆ t, n > h)
```

Пример 2 следует читать так:

1. Если список пуст, то результатом также будет пустой список.

2. Иначе (если список не пуст) выделяется голова (первый элемент) и хвост (список из оставшихся элементов, который может быть пустым). В этом случае результатом будет являться конкатенация (сращивание) отсортированного списка из всех элементов хвоста, которые меньше либо равны голове, списка из самой головы и списка из всех элементов хвоста, которые больше головы.

Пример 3. Быстрая сортировка Хоара на языке Haskell.

```
quickSort [] = []
quickSort (h : t) = quickSort [y | y <- t, y < h] ++ [h] ++ quickSort [y | y <- t, y >= h]
```

Как видно, даже на таком простом примере функциональный стиль программирования выигрывает и по количеству написанного кода и по его элегантности.

Кроме того, все операции с памятью выполняются автоматически. При создании какого-либо объекта под него автоматически выделяется память. После того как объект выполнит своё предназначение, он вскоре будет также автоматически уничтожен сборщиком мусора, который является частью любого функционального языка.

Ещё одним полезным свойством позволяющим сократить программу является встроенный механизм сопоставления с образцом. Это позволяет описывать функции как индуктивные определения. Например:

Пример 4. Определение N-ого числа Фибоначчи.

```
fibb (0) = 1
fibb (1) = 1
fibb (N) = fibb (N - 2) + fibb (N - 1)
```

Механизм сопоставления с образцом будет расмотрен в дальнейших лекциях, однако здесь видно, что функциональные языки выходят на более абстрактный уровень, чем традиционые императивные языки (здесь не рассматривается объектно-ориентированная парадигма и её расширения).

Строгая типизация

Практически все современные языки программирования являются строго типизированными языками (возможно, за исключением JavaScript и его диалектов, не существует императивных языков без понятия «тип»). Строгая типизация обеспечивает безопасность. Программа, прошедшая проверку типов просто не может выпасть в операционную систему с сообщением, подобным "access violation", особенно это касается таких языков, как C/C++ и Object Pascal, где применение указателей является типичным способом использования языка. В функциональных языках большая часть ошибок может быть исправлена на стадии компиляции, поэтому стадия отладки и общее время разработки программ сокращаются. Вдобавок к этому строгая типизация позволяет компилятору генерировать более эффективный код и тем самым ускорять выполнение программ.

Рассматривая пример с быстрой сортировкой Хоара, можно увидеть, что помимо уже упомянутых отличий между вариантом на языке C и вариантом на абстрактном функцио-

нальном языке, есть ещё одно важное отличие: функция на C сортирует список значений типа int (целых чисел), а функция на абстрактном функциональном языке — список значений любого типа, который принадлежит к классу упорядоченных величин. Поэтому последняя функция может сортировать и список целых чисел, и список чисел с плавающей точкой, и список строк. Можно описать какой-нибудь новый тип. Определив для этого типа операции сравнения, возможно без перекомпиляции использовать quickSort и со списками значений этого нового типа. Это полезное свойство системы типов называется параметрическим или истинным полиморфизмом, и поддерживается большинством функциональных языков.

Ещё одной разновидностью полиморфизма является перегрузка функций, позволяющая давать различным, но в чём-то схожим функциям одинаковые имена. Типичным примером перегруженной операции является обычная операция сложения. Функции сложения для целых чисел и чисел с плавающей точкой различны, однако для удобства они носят одно и то же имя. Некоторые функциональные языки помимо параметрического полиморфизма, поддерживают и перегрузку операций.

В языке C++ имеется такое понятие, как шаблон, которое позволяет программисту определять полиморфные функции, подобные quickSort. В стандартную библиотеку C++ STL входит такая функция и множество других полиморфных функций. Но шаблоны C++, как и родовые функции Ada, на самом деле порождают множество перегруженных функций, которые, кстати, компилятор должен каждый раз компилировать, что неблагоприятно сказывается на времени компиляции и размере кода. А в функциональных языках полиморфная функция quickSort — это одна единственная функция.

В некоторых языках, например в Ada, строгая типизация вынуждает программиста явно описывать тип всех значений и функций. Чтобы избежать этого, в строго типизированные функциональные языки встроен специальный механизм, позволяющий компилятору определять типы констант, выражений и функций из контекста. Этот механизм называется механизмом вывода типов. Известно несколько таких механизмов, однако большинство из них являются разновидностями модели типизации Хиндли-Милнера, разработанной в начале 80-х годов XX века. Таким образом, в большинстве случаев можно не указывать типы функций.

Модульность

Механизм модульности позволяет разделять программы на несколько сравнительно независимых частей (модулей) с чётко определёнными связями между ними. Тем самым облегчается процесс проектирования и последующей поддержки больших программных систем. Поддержка модульности не является свойством именно функциональных языков программирования, однако поддерживается большинством таких языков. Существуют очень развитые модульные императивные языки. В качестве примеров подобных языков можно привести Modula-2 и Ada-95.

Функции — это значения

В функциональных языках (равно как и вообще в языках программирования и математике) функции могут быть переданы другим функциям в качестве аргумента или возвра-

щены в качестве результата. Функции, принимающие функциональные аргументы, называются функциями высших порядков или функционалами. Самый, пожалуй, известный функционал, это функция map. Эта функция применяет некоторую функцию ко всем элементам списка, формируя из результатов заданной функции другой список. Например, определив функцию возведения целого числа в квадрат как:

```
square (N) = N * N
```

Можно воспользоваться функцией map для возведения в квадрат всех элементов некоторого списка:

```
squareList = map (square, [1, 2, 3, 4])
```

Результатом выполнения этой инструкции будет список [1, 4, 9, 16].

Чистота (отсутствие побочных эффектов)

В императивных языках функция в процессе своего выполнения может читать и модифицировать значения глобальных переменных и осуществлять операции ввода/вывода. Поэтому, если вызвать одну и ту же функцию дважды с одним и тем же аргументом, может случиться так, что в качестве результата вычислятся два различных значения. Такая функция называется функцией с побочными эффектами.

Описывать функции без побочных эффектов позволяет практически любой язык. Однако некоторые языки поощряют или даже требуют от функции побочных эффектов. Например, во многих объектно-ориентированных языках в функцию член класса передаётся скрытый параметр (чаще он называется this или self), который эта функция неявно модифицирует.

В чистом функциональном программировании оператор присваивания отсутствует, объекты нельзя изменять и уничтожать, можно только создавать новые путем декомпозиции и синтеза существующих. О ненужных объектах позаботится встроенный в язык сборщик мусора. Благодаря этому в чистых функциональных языках все функции свободны от побочных эффектов. Однако это не мешает этим языкам имитировать некоторые полезные императивные свойства, такие как исключения и изменяемые массивы. Для этого существуют специальные методы.

Каковы же преимущества чистых функциональных языков? Помимо упрощения анализа программ есть ещё одно весомое преимущество — параллелизм. Раз все функции для вычислений используют только свои параметры, мы можем вычислять независимые функции в произвольном порядке или параллельно, на результат вычислений это не повлияет. Причём параллелизм этот может быть организован не только на уровне компилятора языка, но и на уровне архитектуры. В нескольких научных лабораториях уже разработаны и используются экспериментальные компьютеры, основанные на подобных архитектурах. В качестве примера можно привести Lisp-машину.

Отложенные вычисления

В традиционных языках программирования (например, C++) вызов функции приводит к вычислению всех аргументов. Этот метод вызова функции называется вызов-по-значению. Если какой-либо аргумент не использовался в функции, то результат вычислений

пропадает, следовательно, вычисления были произведены впустую. В каком-то смысле противоположностью вызова-по-значению является вызов-по-необходимости. В этом случае аргумент вычисляется, только если он нужен для вычисления результата. Примером такого поведения можно взять оператор конъюнкции всё из того же C++ (&&), который не вычисляет значение второго аргумента, если первый аргумент имеет ложное значение.

Если функциональный язык не поддерживает отложенные вычисления, то он называется строгим. На самом деле, в таких языках порядок вычисления строго определен. В качестве примера строгих языков можно привести Scheme, Standard ML и Caml.

Языки, использующие отложенные вычисления, называются нестрогими. Haskell — нестрогий язык, так же как, например, Gofer и Miranda. Нестрогие языки зачастую являются чистыми.

Очень часто строгие языки включают в себя средства поддержки некоторых полезных возможностей, присущих нестрогим языкам, например бесконечных списков. В поставке Standard ML присутствует специальный модуль для поддержки отложенных вычислений. A Objective Caml помимо этого поддерживает дополнительное зарезервированное слово lazy и конструкцию для списков значений, вычисляемых по необходимости.

Решаемые задачи

В качестве задач, традиционно рассматриваемых в курсах функционального программирования, можно выделить следующие:

1. **Получение остаточной процедуры.**

   Если даны следующие объекты:

   $P(x_1, x_2, ..., x_n)$ — некоторая процедура.
   $x_1 = a_1, x_2 = a_2$ — известные значения параметров.
   $x_3, ..., x_n$ — неизвестные значения параметров.

   Требуется получить остаточную процедуру $P_1(x_3, ..., x_n)$. Эта задача решается только на узком классе программ.

2. **Построение математического описания функций.**

   Пусть имеется программа P. Для неё определены входные значения $<x_1, ..., x_n>$ и выходные значения $<y_1, ..., y_m>$. Требуется построить математичекое описание функции $f : D_{x1} \times ... \times D_{xn} \rightarrow D_{y1} \times ... \times D_{ym}$.

3. **Определение формальной семантики языка программирования.**

4. **Описание динамических структур данных.**

5. **Автоматическое построение «значительной» части программы по описанию структур данных, которые обрабатываются создаваемой программой.**

6. **Доказательство наличия некоторого свойства программы.**

7. **Эквивалентная трансформация программ.**

Все эти задачи достаточно легко решаются средствами функционального программирования, но практически неразрешимы в императивных языках.

Справочный материал

- **Языки функционального программирования**

В этом разделе приведено краткое описание некоторых языков функционального программирования (очень немногих). Дополнительную информацию можно почерпнуть, просмотрев ресурсы, перечисленные в следующем разделе.

- **Lisp** (List processor). Считается первым функциональным языком программирования. Нетипизирован. Содержит массу императивных свойств, однако в общем поощряет именно функциональный стиль программирования. При вычислениях использует вызов-по-значению. Существует объектно-ориентированный диалект языка — CLOS.
- **ISWIM** (If you See What I Mean). Функциональный язык-прототип. Разработан Ландиным в 60-х годах XX века для демонстрации того, каким может быть язык функционального программирования. Вместе с языком Ландин разработал и специальную виртуальную машину для исполнения программ на ISWIM'е. Эта виртуальная машина, основанная на вызове-по-значению, получила название SECD-машины. На синтаксисе языка ISWIM базируется синтаксис многих функциональных языков. На синтаксис ISWIM похож синтаксис ML, особенно Caml.
- **Scheme**. Диалект Lisp'а, предназначенный для научных исследований в области computer science. При разработке Scheme был сделан упор на элегантность и простоту языка. Благодаря этому язык получился намного меньше, чем Common Lisp.
- **ML** (Meta Language). Семейство строгих языков с развитой полиморфной системой типов и параметризуемыми модулями. ML преподается во многих западных университетах (в некоторых даже как первый язык программирования).
- **Standard ML**. Один из первых типизированных языков функционального программирования. Содержит некоторые императивные свойства, такие как ссылки на изменяемые значения и поэтому не является чистым. При вычислениях использует вызов-по-значению. Очень интересная реализация модульности. Мощная полиморфная система типов. Последний стандарт языка — Standard ML-97, для которого существует формальные математические определения синтаксиса, а также статической и динамической семантик языка.
- **Caml Light** и **Objective Caml**. Как и Standard ML принадлежит к семейству ML. Objective Caml отличается от Caml Light в основном поддержкой классического объектно-ориентированного программирования. Также как и Standard ML строгий, но имеет некоторую встроенную поддержку отложенных вычислений.
- **Miranda**. Разработан Дэвидом Тернером, в качестве стандартного функционального языка, использовавшего отложенные вычисления. Имеет строгую полиморфную систему типов. Как и ML преподаётся во многих университетах. Оказал большое влияние на разработчиков языка Haskell.

- **Haskell**. Один из самых распространённых нестрогих языков. Имеет очень развитую систему типизации. Несколько хуже разработана система модулей. Последний стандарт языка — Haskell-98.
- **Gofer** (GOod For Equational Reasoning). Упрощённый диалект Haskell'а. Предназначен для обучения функциональному программированию.
- **Clean**. Специально предназначен для параллельного и распределённого программирования. По синтаксису напоминает Haskell. Чистый. Использует отложенные вычисления. С компилятором поставляется набор библиотек (I/O libraries), позволяющих программировать графический пользовательский интерфейс под Win32 или MacOS.

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

- **Список литературы**

  - Хювёнен Э., Сеппенен И. Мир Lisp'а. В 2-х томах. М.: Мир, 1990.
  - Бердж В. Методы рекурсивного программирования. М.: Машиностроение, 1983.
  - Филд А., Харрисон П. Функциональное программирование. М.: Мир, 1993.

- Хендерсон П. Функциональное программирование. Применение и реализация. М.: Мир, 1983.
- Джонс С., Лестер Д. Реализация функциональных языков. М.: Мир, 1991.
- Henson M. Elements of functional languages. Dept. of CS. University of Sassex, 1990.
- Fokker J. Functional programming. Dept. of CS. Utrecht University, 1995.
- Thompson S. Haskell: The Craft of Functional Programming. 2-nd edition, Addison-Wesley, 1999.
- Bird R. Introduction to Functional Programming using Haskell. 2-nd edition, Prentice Hall Press, 1998.

Благодарности

Особую благодарность хочется выразить Сергиевскому Георгию Максимовичу, который в своё время обучил меня основам функционального программирования и помог с организацией этого курса лекций.

Также хочется выразить благодарность Клочкову Андрею, моему коллеге, который с удовольствием помогал мне в создании этого курса.

Лекция 2. «Структуры данных и базисные операции»

Как уже говорилось в первой лекции, основой функциональной парадигмы программирования в большей мере являются такие направления развития математической мысли, как комбинаторная логика и λ-исчисление. В свою очередь последнее более тесно связано с функциональным программированием, и именно λ-исчисление называют теоретическими основами функционального программирования.

Для того, чтобы рассматривать теоретические основы функционального программирования, необходимо в первую очередь ввести некоторые соглашения, описать обозначения и построить некоторую формальную систему.

Пусть заданы объекты некоторого первичного типа A. Сейчас совершенно не важно, что именно представляют собой эти выделенные объекты. Обычно считается, что на этих объектах определён набор базисных операций и предикатов. По традиции, которая пошла ещё от МакКарти (автор Lisp'а), такие объекты называются атомами. В теории фактический способ реализации базисных операций и предикатов совершенно не важен, их существование просто постулируется. Однако каждый конкретный функциональный язык реализует базисный набор по-своему.

В качестве базисных операций традиционно (и в первую очередь это объясняется теоретической необходимостью) выделяются следующие три:

1. Операция создания пары — **prefix** (x, y) ≡ x : y ≡ [x | y]. Эта операция также называется конструктором или составителем.
2. Операция отсечения головы — **head** (x) ≡ **h** (x). Это первая селективная операция.
3. Операция отсечения хвоста — **tail** (x) ≡ **t** (x). Это вторая селективная операция.

Селективные операции отсечения головы и хвоста также называют просто селекторами. Выделенные операции связаны друг с другом следующими тремя аксиомами:

1. **head** (x : y) = x
2. **tail** (x : y) = y
3. **prefix** (**head** (x : y), **tail** (x : y)) = (x : y)

Всё множество объектов, которые можно сконструировать из объектов первичного типа в результате произвольного применения базисных операций, носит название множество S-выражений (обозначение — Sexpr (A)). Например:

$a_1 : (a_2 : a_3) \in \text{Sexpr}$

Для дальнейших исследований вводится фиксированный атом, который также принадлежит первичному типу A. Этот атом в дальнейшем будет называться «пустым списком» и обозначаться символами [] (хотя в разных языках функционального програмирования могут существовать свои обозначения для пустого списка). Теперь можно определить то, чем собственно занимается функциональное программирование — собственное подмножество List (A) ⊂ Sexpr (A), которое называется «список над A».

Определение:
1°. Пустой список [] ∈ List (A)
2°. x ∈ A & y ∈ List (A) → x : y ∈ List (A)

Главное свойство списка: x ∈ List (A) & x ≠ [] → **head** (x) ∈ A; **tail** (x) ∈ List (A).

Для обозначения списка из n элементов можно употреблять множество различных нотаций, однако здесь будет использоваться только такая: $[a_1, a_2, ..., a_n]$. Применяя к такому списку определенным образом операции **head** и **tail** можно добраться до любого элемента списка, т.к.:

head $([a_1, a_2, ..., a_n]) = a_1$
tail $([a_1, a_2, ..., a_n]) = [a_2, ..., a_n]$ (при $n > 0$).

Кроме списков вводится еще один тип данных, который носит название «списочная структура над A» (обозначение — List_str (A)), при этом можно построить следующую структуру отношений: List (A) ⊂ List_str (A) ⊂ Sexpr (A). Определение списочной структуры выглядит следующим образом:

Определение:
1°. a ∈ A → a ∈ List_str (A)
2°. List (List_str (A)) ⊂ List_str (A)

Т.е. видно, что списочная структура — это список, элементами которого могут быть как атомы, так и другие списочные структуру, в том числе и обыкновенные списки. Примером списочной структуры, которая в тоже время не является простым списком, может служить следующее выражение: $[a_1, [a_2, a_3, [a_4]], a_5]$. Для списочных структур вводится такое понятие, как уровень вложенности.

Несколько слов о программной реализации

Пришло время уделить некоторое внимание рассмотрению программной реализации списков и списочных структур. Это необходимо для более тонкого понимания того, что происходит во время работы функциональной программы, как на каком-либо реализованном функциональном языке, так и на абстрактном языке.

Каждый объект занимает в памяти машины какое-то место. Однако атомы представляют собой указатели (адреса) на ячейки, в которых содержатся объекты. В этом случае пара z = x : y графически может быть представлена так, как показано на следующем рисунке.

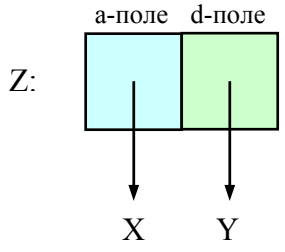

Рисунок 1. Представление пары в памяти компьютера

Адрес ячейки, которая содержит указатели на x и y, и есть объект z. Как видно на рисунке, пара представлена двумя адресами — указатель на голову и указатель на хвост. Традиционно первый указатель (на рисунке выделен голубым цветом) называется a-поле, а второй указатель (на рисунке — зеленоватый) называется d-поле.

Для удобства представления объекты, на которые указывают a- и d-поля, в дальнейшем будут записываться непосредственно в сами поля. Пустой список будет обозначаться перечеркнутым квадратом (указатель ни на что не указывает).

Таким образом, списочная структура, которая рассмотрена несколькими параграфами ранее ([a_1, [a_2, a_3, [a_4]], a_5]) может быть представлена так, как показано на следующем рисунке:

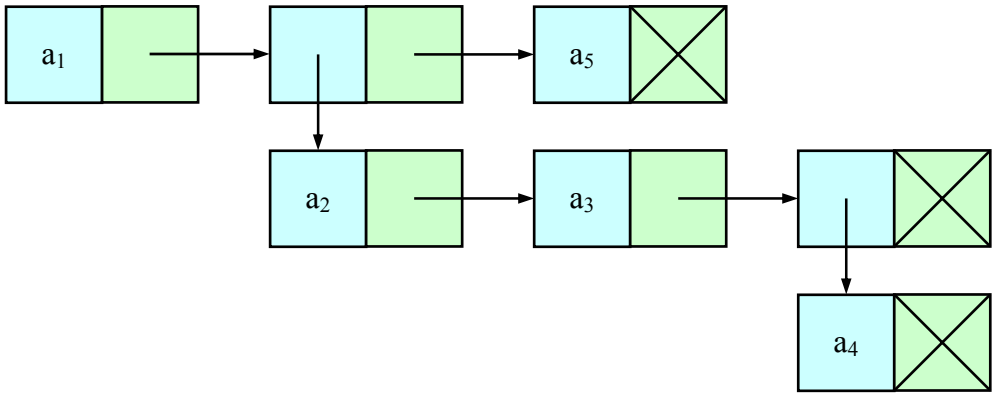

Рисунок 2. Графическое представление списочной структуры [a1, [a2, a3, [a4]], a5]

На этом рисунке также хорошо проиллюстрировано понятие уровня вложенности — атомы a_1 и a_5 имеют уровень вложенности 1, атомы a_2 и a_3 — 2, а атом a_4 — 3 соответственно.

Остается отметить, что операция **prefix** требует расхода памяти, ибо при конструировании пары выделяется память под указатели. С другой стороны обе операции **head** и **tail** не требуют памяти, они просто возвращают адрес, который содержится соответственно в a- или d-поле.

Примеры

Пример 5. Операция prefix.

Для начала необходимо рассмотреть более подробно работу операции **prefix**. Пояснение работы будет проведено на трёх более или менее общих примерах:

1°. **prefix** (a_1, a_2) = a_1 : a_2 (при этом результат не является элементом List_str (A)).

2°. **prefix** (a_1, [b_1, b_2]) = [a_1, b_1, b_2]

3°. **prefix** ([a_1, a_2], [b_1, b_2]) = [[a_1, a_2], b_1, b_2]

Пример 6. Функция определения длины списка Length.

Перед тем, как собственно начать реализовывать функцию Length, необходимо понять, что она должна возвращать. Понятийное определение результата функции Length может звучать как «количество элементов в списке, который передан функции в качестве пара-

метра». Здесь возникает два случая — функции передан пустой список и функции передан непустой список. С первым случаем все ясно — результат должен быть нулевым. Во втором случае задачу можно разбить на две подзадачи, путем разделения переданного списка на голову и хвост при помощи операций **head** и **tail**.

Осмысленно, что операция **head** возвращает первый элемент списка, а операция **tail** возвращает список из оставшихся элементов. Пусть известна длина списка, полученного при помощи операции **tail**, тогда длина исходного списка будет равна известной длина, увеличенной на единицу. В этом случае можно легко записать определение самой функции Length:

```
Length ([]) = 0
Length (L) = 1 + Length (tail (L))
```

Пример 7. Функция слияния двух списков Append.

Реализовать функцию слияния (или сцепления) списков можно многими способами. Первое, что приходит в голову — деструктивное присваивание. Т.е. заменить указатель на [] в конце первого списка на указатель на голову второго списка и тем самым получить результат в первом списке. Однако здесь изменяется сам первый список. Такие приёмы запрещены в функциональном программировании (хотя, в очередной раз необходимо заметить, что в некоторых функциональных языках всё-таки есть такая возможность).

Второй способ состоит в копировании верхнего уровня первого списка и помещении в последний указатель копии ссылку на первый элемент второго списка. Этот способ хорош с точки зрения деструктивности (не выполняет деструктивных и побочных действий), однако требудет дополнительных затрат памяти и времени.

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

Последний пример показывает, как при помощи постепенного конструирования можно построить новый список, который равен сцепке двух заданных.

Упражнения

1. Построить функции, вычисляющие N-ый элемент следующих рядов:
   a. $a_n = x^n$
   b. $a_n = \Sigma_{i=1,n}\ i$
   c. $a_n = \Sigma_{j=1,n}\ (\Sigma_{i=1,j}\ i)$
   d. $a_n = \Sigma_{i=1,p}\ n^{-i}$
   e. $a_n = e^n = \Sigma_{i=0,\infty}\ (n^i / i!)$
2. Объяснить результаты операции prefix, показанные в примере 5. Для объяснения можно воспользоваться графическим методом.
3. Объяснить результат работы функции Append (пример 7). Пояснить, почему функция не является деструктивной.
4. Построить функции, работающие со списками:
   a. GetN — функция вычленения N-ого элемента из заданного списка.

b. ListSumm — функция сложения элементов двух списков. Возвращает список, составленный из сумм элементов списков-параметров. Учесть, что переданные списки могут быть разной длины.
c. OddEven — функция перестановки местами соседних чётных и нечётных элементов в заданном списке.
d. Reverse — функция, обращающая список (первый элемент списка становится последним, второй — предпоследним, и так далее до последнего элемента).
e. Map — функция применения другой переданной в качестве параметра функции ко всем элементам заданного списка.

Ответы для самопроверки

Большинство ответов для самопроверки представляют собой лишь одни из возможных вариантов (в большинстве случаев наиболее интуитивные).

1. Функции, вычисляющие N-ый элемент рядов:

   a. Power:

   ```
   Power (X, 0) = 1
   Power (X, N) = X * Power (X, N – 1)
   ```

   b. Summ_T:

   ```
   Summ_T (1) = 1
   Summ_T (N) = N + Summ_T (N – 1)
   ```

   c. Summ_P:

   ```
   Summ_P (1) = 1
   Summ_P (N) = Summ_T (N) + Summ_P (N – 1)
   ```

   d. Summ_Power:

   ```
   Summ_Power (N, 0) = 1
   Summ_Power (N, P) = (1 / Power (N, P)) + Summ_Power (N, P – 1)
   ```

   e. Exponent:

   ```
   Exponent (N, 0) = 1
   Exponent (N, P) = (Power (N, P) / Factorial (P)) + Exponent (N, P – 1)

   Factorial (0) = 1
   Factorial (N) = N * Factorial (N – 1)
   ```

2. Объяснение работы операции **prefix** можно легко провести в три приёма (равно так же, как и приведено в примере). Для того чтобы не загромождать объяснения, здесь наряду с функциональной записью операции **prefix** также используется инфиксная запись посредством символа двоеточия.

   a. Первый пример работы операции — определение самой операции. Рассматривать его нет смысла, ибо операция **prefix** определяется именно таким образом.

   b. **prefix** (a_1, [b_1, b_2]) = **prefix** (a_1, b_1 : (b_2 : [])) = a_1 : (b_1 : (b_2 : [])) = [a_1, b_1, b_2]
   (Эти преобразование проведены по определению списка).

   c. **prefix** ([a_1, a_2], [b_1, b_2]) = **prefix** ([a_1, a_2], b_1 : (b_2 : [])) = ([a_1, a_2]) : (b_1 : (b_2 : [])) =
   = [[a_1, a_2], b_1, b_2].

3. В качестве примера работы функции Append рассмотрим сцепку двух списков, каждый из которых состоит из двух элементов: [a, b] и [c, d]. Опять же для того, чтобы не загромождать объяснение, для записи операции **prefix** используется инфиксная форма. Для более полного понимания приведенного объяснения необходимо помнить определение списка.

```
Append ([a, b], [c, d]) = a : Append ([b], [c, d]) = a : (b : Append ([], [c, d])) =
      = a : (b : ([c, d])) = a : (b : (c : (d : []))) = [a, b, c, d].
```

4. Функции, работающие со списками:

   a. GetN:

```
GetN (N, []) = _
GetN (1, H:T) = H
GetN (N, H:T) = GetN (N - 1, T)
```

   b. ListSumm:

```
ListSumm ([], L) = L
ListSumm (L, []) = L
ListSumm (H1:T1, H2:T2) = prefix ((H1 + H2), ListSumm (T1, T2))
```

   c. OddEven:

```
OddEven ([]) = []
OddEven ([X]) = [X]
OddEven (H1:[H2:T]) = prefix (prefix (H2, H1), OddEven (T))
```

   d. Reverse:

```
Reverse ([]) = []
Reverse (H:T) = Append (Reverse (T), [H])
```

   e. Map:

```
Map (F, []) = []
Map (F, H:T) = prefix (F (H), Map (F, T))
```

Лекция 3. «Структуры данных и базисные операции - 2»

В этой лекции продолжается описание структур данных и базовых операций, начатое в лекции 2. Более или менее подробно рассматриваются другие аспекты функциональной парадигмы программирования.

Типы в функциональных языках

Как известно, аргументами функций могут быть не только переменные базовых типов, но и другие функции. В этом случае появляется понятие функций высшего порядка. Но для рассмотрения функций высшего порядка необходимо ввести понятие функционального типа (или тип, возвращаемый функцией). Пусть некоторая функция f — это функция одной переменной из множества A, принимающая значение из множества B. Тогда по определению:

$\#(f) : A \to B$

Знак #(f) обозначает «тип функции f». Таким образом, типы, в которых есть символ стрелки $\to$, носят название функциональных типов. Иногда для их обозначения используется более изощренная запись: B^A (далее будет использоваться только стрелочная запись, т.к. для некоторых функций их типы чрезвычайно сложно представить при помощи степеней).

Например: $\#(sin) : Real \to Real$

$\#(Length) : List\ (A) \to Integer$

Для функций многих аргументов определение типа можно выводить при помощи операции декартового произведения (например, $\#(add(x, y)) : Real \times Real \to Real$). Однако в функциональном программировании такой способ определения типов функций многих переменных не прижился.

В 1924 году М. Шёнфинкель предложил представлять функции многих аргументов как последовательность функций одного аргумента. В этом случае тип функции, которая складывает два действительных числа, выглядит так: $Real \to (Real \to Real)$. Т.е. тип таких функций получается последовательным применением символа стрелки $\to$. Пояснить этот процесс можно на следующем примере:

Пример 8. Тип функции add (x, y).

Предположительно, каждый из аргументов функции add уже означен, пусть x = 5, y = 7. В этом случае из функции add путем удаления первого аргумента получается новая функция — add5, которая прибавляет к своему единственному аргументу число 5. Ее тип получается легко, он по определению таков: $Real \to Real$. Теперь, возвращаясь назад, можно понять, почему тип функции add равен $Real \to (Real \to Real)$.

Для того чтобы не изощряться с написанием функций типа add5 (как в предыдущем примере), была придумана специальная аппликативная форма записи в виде «оператор – операнд». Предпосылкой для этого послужило новое видение на функции в функциональном программировании. Ведь если традиционно считалось, что выражение f (5)

обозначает «применение функции f к значению аргумента, равному 5» (т.е. вычисляется только аргумент), то в функциональном программировании считается, что значение функции также вычисляется. Так, возвращаясь к примеру 8, функцию add можно записать как (add (x)) y, а когда аргументы получают конкретные значения (например, (add (5)) 7), сначала вычисляются все функции, пока не появится функция одного аргумента, которая применяется к последнему.

Таким образом, если функция f имеет тип $A_1 \to (A_2 \to (\ldots (A_n \to B) \ldots))$, то чтобы полностью вычислить значение $f(a_1, a_2, \ldots, a_n)$ необходимо последовательно провести вычисление $(\ldots (f(a_1)\ a_2) \ldots)\ a_n$. И результатом вычисления будет объект типа B.

Соответственно выражение, в котором все функции рассматриваются как функции одного аргумента, а единственной операцией является аппликация (применение), называются выражениями в форме «оператор – операнд». Такие функции получили название «каррированные», а сам процесс сведения типа функции к виду, приведенному в предыдущем абзаце — каррированием (по имени Карри Хаскелла).

Если вспомнить λ-исчисление, то обнаружится, что в нем уже есть математическая абстракция для аппликативных форм записей. Например:

$f(x) = x^2 + 5 \quad \Leftrightarrow \quad \lambda x.(x^2 + 5)$

$f(x, y) = x + y \quad \Leftrightarrow \quad \lambda y.\lambda x.(x + y)$

$f(x, y, z) = x^2 + y^2 + z^2 \quad \Leftrightarrow \quad \lambda z.\lambda y.\lambda x.(x^2 + y^2 + z^2)$

И так далее...

Несколько слов о нотации абстрактного языка

Образцы и клозы

Необходимо отметить, что в нотации абстрактного функционального языка, который использовался до сих пор для написаия примеров функций, можно было бы использовать такую конструкцию, как if-then-else. Например, при описании функции Append (см. пример 7), её тело можно было бы записать следующим образом:

```
Append (L₁, L₂) = if (L₁ == [])      then L₂
                                     else head (L₁) : Append (tail (L₁), L₂)
```

Однако данная запись чревата непониманием и трудным разбором. Поэтому даже в примере 7 была использована нотация, которая поддерживает так называемые «образцы».

Определение:

Образцом называется выражение, построенное с помощью операций конструирования данных, которое используется для сопоставления с данными. Переменные обозначаются прописными буквами, константы — строчными.

Примеры образцов:

5 — просто числовая константа
X — просто переменная
X : (Y : Z) — пара
[X, Y] — список

К образцам предъявляется одно требование, которое должно выполняться беспрекословно, иначе сопоставление с образцами будет выполнено неверно. Требование это звучит следующим образом: при сопоставлении образца с данными означивание переменных должно происходить единственным образом. Т.е., например, выражение ($1 + X \Rightarrow 5$) можно использовать как образец, т.к. означивание переменной X происходит единственным образом ($X = 4$), а выражение ($X + Y \Rightarrow 5$) использовать в качестве образца нельзя, ибо означить переменные X и Y можно различным образом.

Кроме образцов в функциональном программировании вводится такое понятие, как «клоз» (от англ. «clause»). По определению клозы выглядят так:

def f p_1, ..., p_n = expr

где:

def и = — константы абстрактного языка
f — имя определяемой функции
p_i — последовательность образцов (при этом $n \geq 0$)
expr — выражение

Таким образом, определение функции в функциональном программировании есть просто последовательность клозов (возможно состоящая из одного элемента). Для того, чтобы упростить запись определений функций, далее слово **def** будет опускаться.

Пример 9. Образцы и клозы в функции Length.

```
Length ([]) = 0
Length (H:T) = 1 + Length (T)
```

Пусть вызов функции Length произведен с параметром [a, b, c]. В этом случае начнет свою работу механизм сопоставления с образцом. Последовательно перебираются все клозы и делаются попытки сопоставления. В данном случае удачное сопоставление будет только во втором клозе (т.к. список [a, b, c] не пуст).

Интерпретация вызова функции заключается в нахождении первого по порядку сверху вниз образца, успешно сопоставимого с фактическим параметром. Значение переменных образца, означиваемые в результате сопоставления, подставляются в правую часть клоза (выражение expr), вычисленное значение которой в данном контексте объявляется значением вызова функции.

Охрана

При написании функций в абстрактной нотации допустимо использовать так называемую охрану, т.е. ограничения на значения переменных образца. Например, при использовании охраны функция Length будет выглядеть примерно следующим образом:

```
Length (L) = 0 when L == []
Length (L) = 1 + Length (tail (L)) otherwise
```

В рассмотренном коде слова **when** (когда) и **otherwise** (в противном случае) являются зарезервированными словами языка. Однако использование этих слов не является необходимым условием для организации охраны. Охрану можно организовывать различными способами, в том числе и с помощью λ-исчисления:

```
Append = λ[].(λL.L)
Append = λ(H:T).(λL.H : Append (T, L))
```

Представленная запись не очень читабельна, поэтому использоваться она будет только в крайних случаях по необходимости.

Локальные переменные

Как уже говорилось, использование локальных переменных представляет собой побочный эффект, поэтому оно недопустимо в функциональных языках. Однако в некоторых случаях использование локальных переменных носит оптимизирующий характер, что позволяет сэкономить время и ресурсы во время вычислений.

Пусть f и h — функции, и необходимо вычислить выражение h (f (X), f(X)). Если в языке нет оптимизирующих методов, то в этом случае произойдет двойное вычисление выражения f (X). Чтобы этого не произошло, можно прибегнуть к такому изощренному способу: (λv.h (v, v))(f (X)). Естественно, что в этом случает выражение f (X) вычислится первым и один раз. Для того, чтобы минимизировать использование λ-исчисления, далее будет применяться следующая форма записи:

let v = f (X) **in** h (v, v)

(слова **let**, = и **in** — зарезервированы в языке). В этом случае v будет называться локальной переменной.

Элементы программирования

Накапливающий параметр — аккумулятор

Бывает так, что при выполнении функции исключительно серьезно встает проблема расхода памяти. Эту проблему можно пояснить на примере функции, вычисляющей факториал числа:

```
Factorial (0) = 1
Factorial (N) = N * Factorial (N - 1)
```

Если провести пример вычисления этой функции с аргументом 3, то можно будет увидеть следующую последовательность:

```
Factorial (3)
3 * Factorial (2)
3 * 2 * Factorial (1)
3 * 2 * 1 * Factorial (0)
3 * 2 * 1 * 1
3 * 2 * 1
3 * 2
6
```

На примере этого вычисления наглядно видно, что при рекурсивных вызовах функций очень сильно используется память. В данном случае количество памяти пропорционально значению аргумента, но аргументов может быть большее число, к примеру. Возникает резонный вопрос: можно ли так написать функцию вычисления факториала (и ей подобные), чтобы память использовалась минимально?

Чтобы ответить на данный вопрос положительно, необходимо рассмотреть понятие аккумулятора (накопителя). Для этого можно рассмотреть следующий пример:

Пример 10. Функция вычисления факториала с аккумулятором.

```
Factorial_A (N) = F (N, 1)

F (0, A) = A
F (N, A) = F ((N - 1), (N * A))
```

В этом примере второй параметр функции F выполняет роль аккумулирующей переменной, именно в ней содержится результат, который возвращается по окончании рекурсии. Сама же рекурсия в этом случае принимает вид «хвостовой», память при этом расходуется только на хранение адресов возврата значения функции.

Хвостовая рекурсия представляет собой специальный вид рекурсии, в которой имеется единственный вызов рекурсивной функции и при этом этот вызов выполняется после всех вычислений.

При реализации вычисления хвостовой рекурсии могут выполняться при помощи итераций в постоянном объеме памяти. На практике это обозначает, что «хороший» транслятор функционального языка должен «уметь» распознавать хвостовую рекурсию и реализовывать её в виде цикла. В свою очередь, метод накапливающего параметра не всегда приводит к хвостовой рекурсии, однако он однозначно помогает уменьшить общий объём памяти.

Принципы построения определений с накапливающим параметром

1. Вводится новая функция с дополнительным аргументом (аккумулятором), в котром накапливаются результаты вычислений.
2. Начальное значение аккумулирующего аргумента задается в равенстве, связывающем старую и новую функции.
3. Те равенства исходной функции, которые соответствуют выходу из рекурсии, заменяются возвращением аккумулятора.
4. Равенства, соответствующие рекурсивному определению, выглядят как обращения к новой функции, в котором аккумулятор получает то значение, которое возвращается исходной функцией.

Возникает вопрос: любую ли функцию можно преобразовать для вычисления с аккумулятором? Очевидно, что ответ на этот вопрос отрицателен. Построение функций с накапливающим параметром — приём не универсальный, и он не гарантирует получения хвостовой рекурсии. С другой стороны, построение определений с накапливающим параметром является делом творческим. В этом процессе необходимы некоторые эвристики.

Определение:

Общий вид рекурсивных определений, позволяющих при трансляции обеспечить вычисления в постоянном объёме памяти через итерацию, называется равенствами в итеративной форме.

Общий вид равенств в итеративной форме может быть описан следующим образом:

$f_i (p_{ij}) = e_{ij}$

При этом на выражения e_{ij} накладываются следующие ограничения:

1. e_{ij} — «простое» выражение, т.е. оно не содержит рекурсивных вызовов, а только операции над данными.
2. e_{ij} имеет вид f_k (v_k), при этом v_k — последовательность простых выражений. Это и есть хвостовая рекурсия.
3. e_{ij} — условное выражение с простым выражением в условии, ветви которого определяются этими же тремя пунктами.

Упражнения

1. Построить функции, работающие со списками. При необходимости воспользоваться дополнительными функциями и теми, что определены в лекции 2.
   a. Reverse_all — функция, получающая на вход списочную структуру и обращающая все её элементы, а также её саму.
   b. Position — функция, возвращающая номер первого вхождения заданного атома в список.
   c. Set — функция, возвращающая список, содержащий единичные вхождения атомов заданного списка.
   d. Frequency — функция, возвращающая список пар (символ, частота). Каждая пара определяет атом из заданного списка и частоту его вхождения в этот список.
2. Написать (если это возможно) функции с накапливающим параметром для заданий из упражнения 1 лекции 2.

Ответы для самопроверки

1. Следующие описания реализуют заявленные функции. В некоторых пунктах реализованы дополнительные функции, обойтись без которых представляется сложным.

   a. Reverse_all:

```
Reverse_all (L) = L when atom (L)
Reverse_all ([]) = []
Reverse_all (H:T) = Append (Reverse_all (T), Reverse_all (H)) otherwise
```

   b. Position:

```
Position (A, L) = Position_N (A, L, 0)

Position_N (A, (A:L), N) = N + 1
Position_N (A, (B:L), N) = Position_N (A, L, N + 1)
Position_N (A, [], N) = 0
```

   c. Set:

```
Set ([]) = []
Set (H:T) = Include (H, Set (T))

Include (A, []) = [A]
Include (A, A:L) = A:L
Include (A, B:L) = prefix (B, Include (A, L))
```

   d. Frequency:

```
Frequency L = F ([], L)

F (L, []) = L
```

```
F (L, H:T) = F (Corrector (H, L), T)

Corrector (A, []) = [A:1]
Corrector (A, (A:N):T) = prefix ((A:N + 1), T)
Corrector (A, H:T) = prefix (H, Corrector (A, T))
```

2. Для всех функций из упражнения 1 лекции 2 можно построить определения с накапливающим параметром. С другой стороны, возможно некоторые из вновь построенных функций не будут оптимизированы.

   a. Power_A:

```
Power_A (X, N) = P (X, N, 1)

P (X, 0, A) = A
P (X, N, A) = P (X, N - 1, X * A)
```

   b. Summ_T_A:

```
Summ_T_A (N) = ST (N, 0)

ST (0, A) = A
ST (N, A) = ST (N - 1, N + A)
```

   c. Summ_P_A:

```
Summ_P_A (N) = SP (N, 0)

SP (0, A) = A
SP (N, A) = SP (N - 1, Summ_T_A (N) + A)
```

   d. Summ_Power_A:

```
Summ_Power_A (N, P) = SPA (N, P, 0)

SPA (N, 0, A) = A
SPA (N, P, A) = SPA (N, P - 1, (1 / Power_A (N, P)) + A)
```

   e. Exponent_A:

```
Exponent_A (N, P) = E (N, P, 0)

E (N, 0, A) = A
E (N, P - 1, (Power_A (N, P) / Factorial_A (P)) + A)
```

Лекция 4. «Основы языка Haskell»

Настоящая лекция будет полностью посвящена синтаксису языка Haskell. Будут рассмотрены все важнейшие понятия языка, их соотношение с уже изученными понятиями (на основе абстрактного функционального языка). Также по возможности будут приводиться примеры на Lisp'е, чтобы не отрываться от основ и традиции.

Структуры данных и их типы

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

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

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

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

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

```
5 :: Integer
```

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

Однако Haskell поддерживает такую незаурядную вещь, как полиморфные типы, или шаблоны типов. Если, например, записать [a], то это будет обозначать тип «список из ато-

мов любого типа», причем тип атомов должен быть одинаковым на протяжении всего списка. Т.е. списки [1, 2, 3] и [‘a’, ‘b’, ‘c’] будут иметь тип [a], а список [1, ‘a’] будет другого типа. В этом случае в записи [a] символ a имеет значение типовой переменной.

Соглашения по именованию

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

Определители списков и математические последовательности

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

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

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

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

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

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

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

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

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

Пример 11. Определение типа для представления двоичных деревьев.

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

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

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

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

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

Вызовы функций

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

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

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

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

```
add 5 7
```

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

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

Можно видеть, что последняя запись — это функция с одним аргументом в строгой нотации Haskell’а. С другой стороны для каррированных функций вполне возможно делать частичное применение. Т.е. при вызове функции двух аргументов передать ей только

один. Как показано в предыдущей лекции результатом такого вызова будет также функция. Более чётко этот процесс можно поиллюстрировать на примере функции inc, которая прибавляет единицу к заданному аргументу:

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

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

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

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

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

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

Использование λ-исчисления

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

Пример 12. Функции add и inc, определённые через λ-абстракции.

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

Пример 13. Вызов аномимной функции.

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

Пример 13 показывает вызов анонимной функции, возводящей в куб переданный параметр. Результатом выполнения этой инструкции будет бесконечный список кубов целых чисел, начиная с нуля. Необходимо отметить, что в Haskell'е используется упрощенный способ записи λ-выражений, т.к. в точной нотации функцию add правильней было бы написать как:

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

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

Инфиксный способ записи функций

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

Пример 14. Инфиксная операция конкатенации списков.

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

Пример 15. Инфиксная операция композиции функций.

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

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

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

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

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

Для вновь определённых инфиксных операций возможно определение порядка вычисления. Для этого в Haskell'е есть зарезервированное слово infixr, которое назначает заданной операции степень её значимости (порядок выполнения) в интервале от 0 до 9, при этом 9 объявляется самой сильной степенью значимости (число 10 также входит в этот интервал, именно эту степень имеет операция применения). Вот так определяются степени для определенных в примерах 14 и 15 операций:

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

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

```
bot   = bot
```

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

```
constant_1 x      = 1
```

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

Упражнения

1. Сконструировать следующие конечные списки (N — количество элементов в конструируемом списке). Для этого воспользоваться либо генераторами списков, либо конструирующими функциями.
   a. Список натуральных чисел. N = 20.
   b. Список нечётных натуральных чисел. N = 20.
   c. Список чётных натуральных чисел. N = 20.
   d. Список степеней двойки. N = 25.
   e. Список степеней тройки. N = 25.
   f. Список треугольных чисел Ферма. N = 50.
   g. Список пирамидальных чисел Ферма. N = 50.
2. Сконструировать следующие бесконечные списки. Для этого воспользоваться либо генераторами списков, либо конструирующими функциями.
   a. Список факториалов.
   b. Список квадратов натуральных чисел.
   c. Список кубов натуральных чисел.
   d. Список степеней пятёрки.
   e. Список вторых суперстепеней натуральных чисел.

Ответы для самопроверки

1. Конечные списки конструируются либо при помощи ограничений, вставляемых в генератор списка, либо при помощи дополнительных ограничивающих параметров.
   a. [1 .. 20]
   b. [1, 3 .. 40] или [1, 3 .. 39]
   c. [2, 4 .. 40]
   d. Список степеней двойки проще всего сконструировать при помощи функции (здесь reverse — функция обращения списка):

```
powerTwo 0 = []
powerTwo n = (2 ^ n) : powerTwo (n - 1)

reverse (powerTwo 25)
```

   e. Список степеней тройки проще всего сконструировать при помощи функции (здесь reverse — функция обращения списка):

```
powerThree 0 = []
powerThree n = (2 ^ n) : powerThree (n - 1)

reverse (powerThree 25)
```

   f. В отличие от предыдущих двух упражнений, здесь можно воспользоваться функцией map, применяющей заданную функцию ко всем элементам списка:

```
t_Fermat 1 = 1
t_Fermat n = n + t_Fermat (n - 1)
```

```
map t_Fermat [1 .. 50]
```

g. Конструирование списка из 50 пирамидальных чисел Ферма также основывается на использовании функции map:

```
p_Fermat 1 = 1
p_Fermat n = t_Fermat n + p_Fermat (n - 1)

map p_Fermat [1 .. 50]
```

2. Бесконечные списки конструируются либо при помощи неограниченных генераторов, либо при помощи конструирующих функций без ограничивающих параметров.
   a. Бесконечный список факториалов:

```
numbersFrom n = n : numbersFrom (n + 1)

factorial n = f_a n 1

f_a 1 m = m
f_a n m = f_a (n - 1) (n * m)

map factorial (numbersFrom 1)
```

   b. Бесконечный список квадратов натуральных чисел:

```
square n = n * n

map square (numbersFrom 1)
```

   c. Бесконечный список кубов натуральных чисел:

```
cube n = n ^ 3

map cube (numbersFrom 1)
```

   d. Бесконечный список степеней пятёрки:

```
powerFive n = 5 ^ n

map powerFive (numbersFrom 1)
```

   e. Бесконечный список вторых суперстепеней натуральных чисел:

```
superPower n 0 = n
superPower n p = (superPower n (p - 1)) ^ n

secondSuperPower n = superPower n 2

map secondSuperPower (numbersFrom 1)
```

Лекция 5. «Служебные слова и синтаксис Haskell'а»

Уже неоднократно подчёркивалось, что синтаксис языка Haskell чрезвычайно похож на синтаксис абстрактного функционального языка. При разработке Haskell'а создатели попытались собрать всё лучшее из имеющихся к этому времени функциональных языков, а также отринуть всё самое плохое. В принципе, это получилось...

Ниже рассматриваются оставшиеся служебные слова, которые ещё не были рассмотрены в предыдущей лекции. Кроме того, по возможности будут показаны различные нововведения в парадигму функционального программирования, которые были реализованы в Haskell'е.

Охрана и локальные переменные

Как было показано в третьей лекции, охрана и локальные переменные используются в функциональном программировании исключительно для простоты записи и понимания текстов программ. Haskell не обошёл своим вниманием и этот аспект, в его синтаксисе имеются специальные средства, которые позволяют организовывать охрану и использовать локальные переменные.

Если возникла необходимость определить какую-либо функцию с использованием механизма охраны, то для этой цели необходимо использовать символ вертикальной черты « | » :

```
sign x      | x > 0     = 1
            | x == 0    = 0
            | x < 0     = -1
```

Функция sign в предыдущем примере использует три охраняющие конструкции, каждая из которых отделена вертикальной чертой от предыдущего определения. В принципе, таких охраняющих конструкций может быть неограниченное количество. Их разбор идёт, естественно, сверху вниз, и если существует непустое пересечение в определении охраны, то сработает та конструкция, которая находится раньше (выше) в записи определения функции.

Для того, чтобы облегчить написание программ, сделать их более читабельными и простыми для понимания в том случае, когда в каком-либо определении функции записано большое количество клозов, в Haskell'е существует ключевое слово «case». При помощи этого слова можно не записывать клозы определения функций так, как это принято в «чистом» функциональном программировании, а несколько сократить запись. Вот общий вид определения функций с ключевым словом «case»:

Function X_1 X_2 ... X_k = **case** (X_1, X_2, ..., X_k) **of**
(P_{11}, P_{21}, ..., P_{k1}) $\rightarrow$ $Expression_1$
...
(P_{1n}, P_{2n}, ..., P_{kn}) $\rightarrow$ $Expression_n$

В приведенном выше описании жирным шрифтом выделены служебные слова и символы языка.

Так функция, которая возвращает список из первых n элементов заданного списка, может быть определена следующим образом при помощи служебного слова «case»:

```
takeFirst n l =   case (n, l) of
                  (0, _) -> []
                  (_, []) -> []
                  (n, (x:xs)) -> (x) : (takeFirst (n - 1) xs)
```

И такая запись будет полностью эквивалентна обычному определению функции:

```
takeFirst 0 _ = []
takeFirst _ [] = []
takeFirst n (x:xs) = (x) : (takeFirst (n - 1) xs)
```

Пришло время объяснить понятие маски подстановки. В Haskell'е маску обозначают символом нижней черты « _ » (абсолютно также, как и в Prolog'е). Этот символ заменяет любой образец и является своего рода анонимной переменной. Если в выражении клоза нет необходимости использования переменной образца, то её можно заменить маской подстановки. При этом, естественно, происходят отложенные вычисления — то выражение, которое может быть подставлено вместо маски, не вычисляется.

Другим способом использования охраняющих конструкций является использование конструкции «if-then-else». В Haskell'е реализована и эта возможность. Формально, эта конструкция может быть лекго трансформирована в выражение с использованием служебного слова «case». Можно даже считать, что выражение:

```
if Exp1 then Exp2 else Exp3
```

Является сокращением выражения:

```
case (Exp1) of
      (True) -> Exp2
      (False) -> Exp3
```

Естественно, что тип Exp1 должен быть Boolean (Bool в Haskell'е), а типы выражений Exp2 и Exp3 совпадать (ведь именно значения этих выражений будет возвращено определяемой через конструкцию «if-then-else» функцией).

Для использования локальных переменных (в смысле функциональной парадигмы программирования) в Haskell'е существует два типа записи. Первый тип полностью соответствует математической нотации, введенной в третьей лекции:

```
let   y = a * b
      f x = (x + y) / y
in f c + f d
```

Другим способом определения локальной переменной является её описание после использования. В этом случае используется служебное слово «where», которое ставится в конце выражения:

```
f x y | y > z     = y - z
      | y == z    = 0
      | y < z     = z - y
      where z = x * x
```

Таким образом видно, что Haskell поддерживает два способа записи определения локальных переменных — префиксный (при помощи служебного слова «let») и постфиксный (при помощи служебного слова «where»). Оба способа являются равнозначными, их

употребление зависит только от наклонностей программиста. Однако обычно постфиксный способ записи используется в выражениях, где также есть охрана, в то время как префиксный способ записи используется в остальных случаях.

Полиморфизм

В первой лекции уже было показано, что функциональная парадигма программирования поддерживает чистый или параметрический полиморфизм. Однако большинство современных языков программирования не обходятся без так называемого полиморфизма ad hoc, или перегрузки. Например, перегрузка практически повсеместно используется для следующих целей:

- Литералы 1, 2, 3 и т.д. (т.е. цифры) используются как для записи целых чисел, так и для записи чисел произвольной точности.
- Арифметические операции (например, сложение — знак « + ») используются для работы с данными различных типов (в том числе и с нечисловыми данными, например — конкатенация для строк).
- Оператор сравнения (в Haskell'е знак двойного равенства — « == ») используется для сравнения данных различных типов.

Перегруженное поведение операций различное для каждого типа данных (зачастую такое поведение вообще может быть неопределено или определено ошибочно), в то время, как в параметрическом полиморфизме тип данных, вообще говоря, не важен. Однако в Haskell'е есть механизм для использования перегрузки.

Рассмотреть возможность использования полиморфизма ad hoc проще всего на примере операции сравнения. Существует большое множество типов, для которых можно и целесообразно переопределить операцию сравнения, но для некоторых типов эта операция бесполезна. Например, сравнение функций бессмысленно, функции необходимо вычислять и сравнивать результаты этих вычислений. Однако, например, может возникнуть необходимость сравнивать списки. Конечно, здесь речь идет о сравнении значений, а не сравнении указателей (как, например, это сделано в Java).

Рассмотрим функцию element, которая возвращает значение истины в зависимости от того, присутствует заданный элемент в заданном списке, или нет (в целях стиля данная функция описана в инфиксной форме):

```
x `element` []          = False
x `element` (y:ys)      = (x == y) || (x `element` ys)
```

Здесь видно, что у функции element тип (a → [a] → Bool), но при этом тип операции « == » должен быть (a → a → Bool). Т.к. переменная типа a может обозначать любой тип (в том числе и список), целесообразно переопределить операцию « == » для работы с любым типом данных.

Классы типов являются решением этой проблемы в Haskell'е. Для того, чтобы рассмотреть этот механизм в действии далее определяется класс типов, содержащий оператор равенства.

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

В этой конструкции использованы служебные слова «class» и «where», а также переменная типов a. Символ «Eq» является именем определяемого класса. Эта запись может быть прочитана следующим образом: «Тип a является экземпляром класса Eq, если для этого типа перегружена операция сравнения « == » соответствующего типа». Необходимо отметить, что операция сравнения должна быть определена на паре объектов одного и того же типа.

Ограничение того факта, что тип a должен быть экземпляром класса Eq записывается как (Eq a). Поэтому выражение (Eq a) не является описанием типа, но оно описывает ограничение, накладываемое на тип a, и это ограничение в Haskell'е называется контекстом. Контексты располагаются перед определением типов и отделяются от них символами « => »:

```
(==)  :: (Eq a) => a -> a -> Bool
```

Эта запись может быть прочитана как «Для каждого типа a, который является экземпляром класса Eq, определена операция « == », которая имеет тип (a → a → Bool)». Эта операция должна быть использована в функции element, поэтому ограничение распространяется и на неё. В этом случае необходимо явно указывать тип функции:

```
element     :: (Eq a) => a -> [a] -> Bool
```

Этой записью декларируется тот необходимый факт, что функция element определена не для всех типов данных, но только для тех, для которых определена соответствующая операция сравнения.

Однако теперь возникает проблема определения того, какие типы являются экземплярами класса Eq. Для этого в Haskell'е есть служебное слово «instance». Например, для того, чтобы предписать, что тип Integer является экземпляром класса Eq, необходимо написать:

```
instance Eq Integer where
      x == y      = x `integerEq` y
```

В этом выражении определение операции « == » называется определением метода (как в объектно-ориентированной парадигме). Функция integerEq может быть любой (и не обязательно инфиксной), главное, чтобы у нее был тип (a → a → Bool). В этом случае скорее всего подойдет примитивная функция сравнения двух натуральных чисел. В свою очередь прочитать написанное выражение можно следующим образом: «Тип Integer является экземпляром класса Eq, и далее приводится определение метода, который производит сравнение двух объектов типа Integer».

Таким же образом можно определить операцию сравнения и для бесконечных рекурсивных типов. Например, для типа Tree (см. лекцию 4) определение будет выглядеть следующим образом:

```
instance (Eq a) => Eq (Tree a) where
      Leaf a == Leaf b                    = a == b
      (Branch l1 r1) == (Branch l2 r2)    = (l1 == l2) && (r1 == r2)
      _ == _                              = False
```

Естественно, что если в языке определено понятие класса, должно быть определена и концепция наследования. Хотя в Haskell'е под классом понимается нечто более абстрактное, чем в обычных объектно-ориентированных языках, в Haskell'е также есть и наследо-

вание. Но в то же время концепция наследования определяется столь же изощренно, что и понятие класса. Например, от определённого выше класса Eq можно унаследовать класс Ord, который будет представлять сравнимые типы данных. Его определение будет выглядеть следующим образом:

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

Все экземпляры класса Ord должны определять кроме операций «меньше», «больше», «меньше или равно», «больше или равно», «минимум» и «максимум» ещё и операцию сравнения « == », т.к. её класс Ord наследует от класса Eq.

Самое удивительно заключается в том, что Haskell поддерживает множественное наследование. В случае использования нескольких базовых классов, всех их просто надо перечислить через запятую в соответствующей секции. При этом экземпляры классов, унаследованных от нескольких базовых классов, должны, естественно, поддерживать и все методы базовых классов.

Сравнение с другими языками

Хотя классы существуют во многих других языках программирования, понятие класса в Haskell’е несколько отличается.

- Haskell разделяет определения классов и их методов, в то время как такие языки, как C++ и Java вместе определяют структуру данных и методы для её обработки.
- Определения методов в Haskell’е соответствуют виртуальным функциям C++. Каждый конкретный экземпляр класса должен переопределять методы класса.
- Больше всего классы в Haskell’е похожи на интерфейсы Java. Как и определение интерфейса, классы в Haskell’е предоставляют протокол использования объекта, вместо определения самих объектов.
- Haskell не поддерживает стиль перегрузки функции, используемый в C++, когда функции с одним и тем же именем получают данные различных типов для обработки.
- Типы объектов в Haskell’е не могут быть выведены неявно. В Haskell’е не существует базового класса для всех классов (как, например, TObject в Object Pascal’е).
- C++ и Java добавляют в скомпилированный код идентифицирующую информацию (например, таблицы размещения виртуальных функций). В Haskell’е такого нет. Во время интерпретации (компиляции) вся необходимая информация выводится логически.
- Не существует понятия контроля за доступом — нет публичных и защищенных методов. Вместо этого Haskell предоставляет механизм модуляризации программ.

Упражнения

1. В нотации Haskell’а записать функции, работающие со списками (из упражнений лекции 2). По возможности воспользоваться формализмами охраны и локальными переменными.

   a. getN — функция вычленения N-ого элемента из заданного списка.

b. listSumm — функция сложения элементов двух списков. Возвращает список, составленный из сумм элементов списков-параметров. Учесть, что переданные списки могут быть разной длины.
c. oddEven — функция перестановки местами соседних чётных и нечётных элементов в заданном списке.
d. reverse — функция, обращающая список (первый элемент списка становится последним, второй — предпоследним, и так далее до последнего элемента).
e. map — функция применения другой переданной в качестве параметра функции ко всем элементам заданного списка.

2. В нотации Haskell'а записать более сложные функции, работающие со списками (из упражнений лекции 3). При необходимости воспользоваться дополнительными функциями и определёнными в предыдущем упражнении. По возможности воспользоваться формализмами охраны и локальными переменными.
   a. reverseAll — функция, получающая на вход списочную структуру и обращающая все её элементы, а также её саму.
   b. position — функция, возвращающая номер первого вхождения заданного атома в список.
   c. set — функция, возвращающая список, содержащий единичные вхождения атомов заданного списка.
   d. frequency — функция, возвращающая список пар (символ, частота). Каждая пара определяет атом из заданного списка и частоту его вхождения в этот список.

3. Описать следующие классы типов. При необходимости воспользоваться механизмом наследования классов.
   a. Show — класс, объекты экземпляров которого могут быть выведены на экран.
   b. Number — класс, описывающий числа различной природы.
   c. String — класс, описывающий строки (списки символов).

4. Определить типы-экземпляры классов, описанных в предыдущем задании. По возможности для каждого экземпляра класса определить методы, работающие с объектами этого класса.
   a. Integer — тип целых чисел.
   b. Real — тип действительных чисел.
   c. Complex — тип комплексных чисел.
   d. WideString — тип строк, в виде последовательности двухбайтовых символов в кодировке UNICODE.

Ответы для самопроверки

1. Все нижеприведённые описания на Haskell'е являются лишь одними из большого ряда возможных:

   a. getN:

```
getN              :: [a] -> a
getN n []         = _
getN 1 (h:t)      = h
getN n (h:t)      = getN (n - 1) t
```

b. listSumm:

```
listSumm                    :: Ord (a) => [a] -> [a]
listSumm [] l               = l
listSumm l []               = l
listSumm (h1:t1) (h2:t2)    = (h1 + h2) : (listSumm t1 t2)
```

c. oddEven:

```
oddEven                 :: [a] -> [a]
oddEven []              = []
oddEven [x]             = [x]
oddEven (h1:(h2:t))     = (h2:h1) : (oddEven t)
```

d. reverse:

```
append          :: [a] -> [a] -> [a]
append [] l     = l
append (h:t) l2 = h : (append t l2)

reverse         :: [a] -> [a]
reverse []      = []
reverse (h:t)   = append (reverse t [h])
```

e. map:

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

2. Все нижеприведённые описания на Haskell'е являются лишь одними из большого ряда возможных:

a. reverseAll:

```
atom            :: ListStr (a) -> Bool
atom a          = True
atom _          = False

reverseAll      :: ListStr (a) -> ListStr (a)
reverseAll l    = l
reverseAll []   = []
reverseAll (h:t)  = append (reverseAll t) (reverseAll h)
```

b. position:

```
position              :: a -> [a] -> Integer
position a l          = positionN a l 0

positionN             :: a -> [a] -> Integer -> Integer
positionN a (a:t) n   = (n + 1)
positionN a (h:t) n   = positionN a t (n + 1)
positionN a [] n      = 0
```

c. set:

```
set             :: [a] -> [a]
set []          = []
set (h:t)       = include h (set t)

include         :: a -> [a] -> [a]
include a []    = [a]
include a (a:t) = a : t
include a (b:t) = b : (include a t)
```

d. frequency:

```
frequency             :: [a] -> [(a : Integer)]
frequency l           = f [] l

f                     :: [a] -> [a] -> [(a : Integer)]
f l []                = l
f l (h:t)             = f (corrector h l) t

corrector             :: a -> [a] -> [(a : Integer)]
corrector a []        = [(a : 1)]
```

```
corrector a (a:n):t     = (a : (n + 1)) : t
corrector a h:t         = h : (corrector a t)
```

3. Все нижеприведённые описания на Haskell'е являются лишь одними из большого ряда возможных:
   a. Show:

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

   b. Number:

```
class Number a where
      (+)   :: a -> a -> a
      (-)   :: a -> a -> a
      (*)   :: a -> a -> a
      (/)   :: a -> a -> a
```

   c. String:

```
class String a where
      (++)        :: a -> a -> a
      length      :: a -> Integer
```

4. Все нижеприведённые описания на Haskell'е являются лишь одними из большого ряда возможных:
   a. Integer:

```
instance Number Integer where
      x + y = plusInteger x y
      x - y = minusInteger x y
      x * y = multInteger x y
      x / y = divInteger x y

plusInteger       :: Integer -> Integer -> Integer
plusInteger x y   = x + y

...
```

   b. Real:

```
instance Number Real where
      x + y = plusReal x y
...
```

   c. Complex:

```
instance Number Complex where
      x + y = plusComplex x y
...
```

   d. WideString:

```
instance String WideString where
      x ++ y      = wideConcat x y
      length x    = wideLength x

wideConcat x y    = append x y
wideLength x      = length x
```

Лекция 6. «Модули и монады в Haskell'е»

Ни один язык программирования не обходится без понятия модуля, разве что только языки самого низкого уровня, да и те в последнее время приобретают свойства языков более высокого уровня. Модули пришли из давнего прошлого, когда программирование, как искусство (или ремесло), только развивалось. Возникла необходимость разбивать программы на логические части, каждую из которых создавал бы отдельный разработчик или коллектив разработчиков.

В Haskell'е также существует понятие модуля. Однако более интригующим и завораживающим неофитов явлением в языке является понятие «монада». Далее в этой лекции будут подробно рассмотрены оба понятия — «модуль» и «монада».

Модули

В Haskell'е модули несут двоякое назначение — с одной стороны модули необходимы для контроля за пространством имён (как, собственно, и во всех других языках программирования), с другой стороны при помощи модулей можно создавать абстрактные типы данных.

Определение модуля в Haskell'е достаточно просто. Именем модуля может быть любой символ, начинается имя только с заглавной буквы. Дополнительно имя модуля никак не связано с файловой системой (как, например, в Pascal'е и в Java), т.е имя файла, содержащего модуль, может быть не таким же, как и название модуля. На самом деле, в одном файле может быть несколько модулей, т.к. модуль — это всего лишь навсего декларация самого высокого уровня.

Как известно, на верхнем уровне модуля в Haskell'е может быть множество деклараций (описаний и определений) — типы, классы, данные, функции. Однако один вид деклараций должен стоять в модуле на первом месте (если этот вид деклараций вообще используется). Речь идет о включении в модуль других модулей — для этого используется служебное слово import. Остальные определения могут появляться в любой последовательности.

Определение модуля должно начинаться со служебного слова module. Например, ниже приведено определение модуля Tree:

```
module Tree (Tree (Leaf, Branch), fringe) where

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

fringe                          :: Tree a -> [a]
fringe (Leaf x)                 = [x]
fringe (Branch left right)      = fringe left ++ fringe right
```

В этом модуле описан один тип (Tree — ничего страшного, что имя типа совпадает с названием модуля, в данном случае они находятся в различных пространствах имён) и одна функция (fringe). В данном случае модуль Tree явно экспортирует тип Tree (вместе со своими подтипами Leaf и Branch) и функцию fringe — для этого имена типа и функции указаны в скобках после имени модуля. Если наименование какого-либо объекта не ука-

зывать в скобках, то он не будет экспортироваться, т.е. этот объект не будет виден извне текущего модуля.

Использование модуля в другом модуле выглядит еще проще:

```
module Main where

import Tree (Tree(Leaf, Branch), fringe)

main = print (fringe (Branch (Leaf 1) (Leaf 2)))
```

В приведенном примере видно, что модуль Main импортирует модуль Tree, причём в декларации import явно описано, какие именно объекты импортируются из модуля Tree. Если это описание опустить, то импортироваться будут все объекты, которые модуль экспортирует, т.е. в данном случае можно было просто написать: import Tree.

Бывает так, что один модуль импортирует несколько других (надо заметить, что это обычная ситуация), но при этом в импортируемых модулях существуют объекты с одним и тем же именем. Естественно, что в этом случае возникает конфликт имён. Чтобы этого избежать в Haskell'е существует специальное служебное слово qualified, при помощи которого определяются те импортируемые модули, имена объектов в которых приобретают вид: <Имя Модуля>.<Имя Объекта>, т.е. для того, чтобы обратиться к объекту из квалифицированного модуля, перед его именем необходимо написать имя модуля:

```
module Main where

import qualified Tree

main = print (Tree.fringe (Tree.Leaf 'a'))
```

Использование такого синтаксиса полностью лежит на совести программиста. Некоторым нравится полная определённость, которую привносят квалифицированные имена, и они используют их в ущерб размеру программ. Другим нравится использовать короткие мнемонические имена, и они используют квалификаторы (имена модулей) только по необходимости.

Абстрактные типы данных

В Haskell'е модуль является единственным способом создать так называемые абстрактные типы данных, т.е. такие, в которых скрыто представление типа, но открыты только специфические операции над созданным типом, набор которых вполне достаточен для работы с типом. Например, хотя тип Tree является достаточно простым, его все-таки лучше сделать абстрактным типом, т.е. скрыть то, что Tree состоит из Leaf и Branch. Это делается следующим образом:

```
module TreeADT (Tree, leaf, branch, cell, left, right, isLeaf) where

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

leaf                    = Leaf
branch                  = Branch
cell (Leaf a)           = a
left (Branch l r)       = l
right (Branch l r)      = r
isLeaf (Leaf _)         = True
isLeaf _                = False
```

Видно, что к внутренностям типа Tree внешний пользователь (программист) может пробраться только при помощи использования определённых функций. Впоследствии, когда создатель этого модуля захочет изменить представление типа (например, оптимизировать его), ему необходимо будет изменить и функции, которые оперируют полями типа Tree. В свою очередь программист, который использовал в своей программе тип Tree, ничего менять не будет, т.к. его программа все так же останется работоспособной.

Другие аспекты использования модулей

Далее приводятся дополнительные аспекты системы модулей в Haskell'е:

- В декларации импорта (import) можно выборочно спрятать некоторые из экспортируемых объектов (при помощи служебного слова hiding). Это бывает полезным для явного исключения определений некоторых объектов из импортируемого модуля.
- При импорте можно определить псевдоним модуля для квалификации имен экспортируемых из него объектов. Для этого используется служебное слово as. Это может быть полезным для того укорачивания имен модулей.
- Все программы неявно импортируют модуль Prelude. Если сделать явный импорт этого модуля, то в его декларации возможно скрыть некоторые объекты, чтобы впоследствии их переопределить.
- Все декларации instance неявно экспортируются и импортируются всеми модулями.
- Методы классов могут быть так же как и подтипы данных перечислены в скобках после имени соответствующего класса во время декларации экспорта/импорта.

Монады

Многие новички в функциональном програмировании бывают часто озадачены понятием монады в Haskell'е. Но монады очень часто встречаются в языке, так, например, система ввода/вывода основана именно на понятии монады, а стандартные библиотеки содержат целые модули посвященные монадам. Необходимо отметить, что понятие монады в Haskell'е основано на теории категорий, однако для того, чтобы не вдаваться в абстрактную математику, далее будет представлено интуитивное понимание монад.

Монады являются типами, которые представляют собой экземпляры одного из следующих монадических классов: Functor, Monad и MonadPlus. Ни один из этих классов не может являться предком для другого класса, т.е. монадические классы ненаследуемы. В модуле Prelude определены три монады: IO, [] и Maybe, т.е. список также является монадой.

Математически монада определяется через набор правил, которые связывают операции, производимые над монадой. Эти правила дают интуитивное понимание того, как должна использоваться монада и какова ее внутренняя структура. Для конкретизации далее рассматривается класс Monad, в котором определены две базовые операции и одна функция:

```
class Monad m where
      (>>=)       :: m a -> (a -> m b) -> m b
      (>>)        :: m a -> m b -> m b
      return      :: a -> m a
      fail        :: String -> m a

      m >> k      = m >>= \_ -> k
```

Две операции (>>=) и (>>) — это операции связывания. Они комбинируют два монадических значения, тогда как функция return преобразует переданное значение некоторого типа a в монадическое значения типа m a. Сигнатура операции (>>) помогает понять операцию связывания: выражение (m a >>= \v → m b) комбинирует монадическое значение m a, содержащее объект типа a, с функцией, которая оперирует значением v типа a и возвращает результат типа m b. Результатом же комбинации будет монадическое значение типа m b. Операция (>>) используется тогда, когда функция не использует значения, полученного первым монадическим операндом.

Точное значение операция связывания конечно же зависит от конкретной реализации монады. Так, например, тип IO определяет операцию (>>=) как последовательное выполнение двух её операндов, а результат выполнения первого операнда последовательно передается во второй. Для двух других встроенных монадических типов (списки и Maybe) эта операция определена как передача нуля или более параметров из одного вычислительного процесса в следующий.

В Haskell'е определено специальное служебное слово, которое на уровне языка поддерживает использование монад. Это слово do, понимание которого можно увидеть в следующих правилах его применения:

```
do e1 ; e2        = e1 >> e2
do p <- e1 ; e2   = e1 >>= \p -> e2
```

Первое выражение выполняется всегда (переноса значений из первого операнда во второй не производится). Во втором выражении может произойти ошибка, в этом случае производится вызов функции fail, которая также определена в классе Monad. Поэтому более точное определение служебного слова do для второго случая будет выглядеть так:

```
do p <- e1 ; e2   = e1 >>= (\v -> case v of
                                    p -> e2
                                    _ -> fail "s")
```

где s — это строка, которая может определять местоположение оператора do в программе, а может просто нести некоторую семантическую нагрузку. Например, в монаде IO действие ('a' ← getChar) вызовет функцию fail в том случае, если считанный символ не является символом 'a'. Это действие прерывает исполнение программы, т.к. определенная в монаде IO функция fail в свою очередь вызывает системную функию error.

Класс MonadPlus используется для тех монад, в которых есть нулевой элемент и операция «+». Определение этого класса выглядит следующим образом:

```
class (Monad m) => MonadPlus m where
      mzero :: m a
      mplus :: m a -> m a -> m a
```

Нулевой элемент в этом классе монад подчиняется следующим правилам:

```
m >>= \x -> mzero = mzero
mzero >>= m       = mzero
```

Например, для списков нулевым элементом является пустой список [], а операцией «+» — конкатенация списков. Поэтому монада списков является экземпляром класса MonadPlus. С другой стороны монада IO не имеет нулевого элемента, поэтому монада IO является экземпляром только класса Monad.

Встроенные монады

Для конкретизации полученных сведений необходимо рассмотреть что-нибудь более приземленное. Так как списки являются монадами и в то же время списки достаточно скрупулезно изучены, они являются отличным материалом, на котором можно рассмотреть практическое применение механизма монад.

Для списков операция связывания обретает смысл в соединении вместе набора операций, производимых над каждым элементом списка. При использовании со списками сигнатура операции (>>=) приобретает следующий вид:

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

Это обозначает, что дан список значений типа a и функция, которая проецирует значение типа a на список значений типа b. Связывание применяет функцию к каждому элементу данного списка значений типа a и возвращает полученный список значений типа b. Эта операция уже известна — определители списков работают именно таким образом. Т.е. следующие три выражения абсолютно одинаковы:

```
-- Выражение 1 ----------------------------------------------------------------------------
[(x, y) | x <- [1, 2, 3], y <- [1, 2, 3], x /= y]

-- Выражение 2----------------------------------------------------------------------------
do    x <- [1, 2, 3]
      y <- [1, 2, 3]
      True <- return (x /= y)
      return (x, y)

-- Выражение 3----------------------------------------------------------------------------
[1, 2, 3] >>= (\x -> [1, 2, 3] >>= (\y -> return (x /= y) >>=
      (\r -> case r of
            True  -> return (x, y)
            _     -> fail ””)))
```

Какое выражение использовать в написании программ — выбирать программисту.

Упражнения

1. Какое интуитивное понимание можно вложить в понятие «монада»?
2. Какой практический смысл имеет применение монад в функциональном программировании?

Ответы для самопроверки

1. Первое, что приходит в голову это то, что монада — это контейнерный тип. Ведь действительно, список — это контейнерный тип, т.к. внутри списка содержатся элементы другого типа. Именно это и показано в определении монадического типа:

```
class Monad m where
      (>>=)       :: m a -> (a -> m b) -> m b
      (>>)        :: m a -> m b -> m b
      return      :: a -> m a
      fail        :: String -> m a
```

   Запись «m a» как бы показывает, что тип a (необходимо чётко помнить, что при определении классов и других типов данных символы типа a, b и т.д. обозначают переменные типов) обрамлён монадическим типом m. Однако в реальности физическое обрамление доступно только для монадического типа «список», т.к. его обозначение в

виде квадратных скобок пошло традиционно. В строгой нотации Haskell'а нужно было бы писать что-нибудь вроде: List (1 2 3 4 5) — это список [1, 2, 3, 4, 5].

2. Применение монад в функциональных языках — это по существу возвращение к императивности. Ведь операции связывания (>>=) и (>>) предполагают последовательное выполнение связанных выражений с передачей или без результатов вычисления. Т.е. монады — это императивное ядро внутри функциональных языков. С одной стороны это идёт в разрез с теорией функционального програмирования, где отрицается понятие императивности, но с другой стороны некоторые задачи решаются только при помощи императивных принципов. И опять же, Haskell предоставляет удивительную возможность по генерации списков, но это только благодаря тому, что сам тип «список» выполнен в виде монады.

Лекция 7. «Операции ввода/вывода в Haskell'е»

В Haskell'е, как и во всех остальных языках, существует всесторонняя система операций ввода/вывода. Хотя обычно операции ввода/вывода предполагают некоторую последовательность в своем выполнении, т.е. по сути дела императивность, в Haskell'е система операций ввода/вывода полностью поддерживает функциональную парадигму программирования.

Уже говорилось, что все операции ввода/вывода построены при помощи такого понятия языка Haskell, как монада (лекция 6). В то же время для понимания системы операций ввода/вывода в Haskell'е нет особой необходимости понимать теоретические основы понятия монада. Монады можно рассматривать как концептуальные рамки, в которых содержится система ввода/вывода. Можно сказать, что понимание теории категорий так же необходимо для использования системы операций ввода/вывода в Haskell'е, как и понимание теории групп для выполнения арифметических операций.

Операции ввода/вывода в любом языке основаны на понятии действия. При возбуждении действия, оно выполняется. Однако в Haskell'е действия не возбуждаются, а скорее просто декларируются. В свою очередь действия могут быть атомарными или составленными из последовательности других действий. Монада IO содержит операции, которые позволяют создавать сложные действия из атомарных. Т.е. монаду в данном случае можно рассматривать как клей, который связывает действия в программе.

Базовые операции ввода/вывода

Каждое действие ввода/вывода возвращает какое-то значение. Для того, чтобы различать эти значения от базовых, типы этих значений как бы обернуты типом IO (необходимо помнить, что монада является контейнерным типом). Например, тип функции getChar следующий:

```
getChar     :: IO Char
```

В этом примере показано, что функция getChar выполняет некоторое действие, которое возвращает значение типа Char. Действия, которые не возвращают ничего интересного, имеют тип IO (). Т.е. символ () обозначает пустой тип (void в других языках). Так функция putChar имеет тип:

```
putChar     :: Char -> IO ()
```

Друг с другом действия связываются при помощи оператора связывания. Т.е. символы >>= выстраивают последовательность действий. Как известно, вместо этой функции можно использовать служебное слово do. Оно использует такой же двумерный синтексис, как и слова let и where, поэтому можно не использовать для разделения вызова функций символ « ; ». При помощи слова do можно связывать вызовы функций, определение данных (при помощи символа ←) и множество определений локальных переменных (служебное слово let).

Например, так можно определить программу, которая читает символ с клавиатуры и выводит его на экран:

```
main  :: IO ()
main  = do  c <- getChar
            putChar c
```

В этом примере не случайно для имени функции выбрано слово main. В Haskell'е, также, как и в языке C/C++ название функции main используется для обозначения точки входа в программу. Кроме того, в Haskell'е тип функции main должен быть типом монады IO (обычно используется IO ()). Ко всему прочему, точка входа в виде функции main должна быть определена в модуле с именем Main.

Пусть имеется функций ready, которая должна возвращать True, если нажата клавиша «y», и False в остальных случаях. Нельзя просто написать:

```
ready       :: IO Bool
ready       = do  c <- getChar
                  c == 'y'
```

Потому что в этом случае результатом выполнения операции сравнения будет значение типа Bool, а не IO Bool. Для того, чтобы возвращать монадические значения, существует специальная функция return, которая из простого типа данных делает монадический. Т.е. в предыдущем примере последняя строка определения функции ready должна была выглядеть как «return (c == 'y')».

В следующем примере показана более сложная функция, которая считывает строку символов с клавиатуры:

Пример 16. Функция getLine.

```
getLine     :: IO String
getLine     = do  c <- getChar
                  if c == '\n'
                        then return ""
                        else do l <- getLine
                                return (c : l)
```

Необходимо помнить, что в тот момент, когда программист перешёл в мир действий (использовал систему операций ввода/вывода), назад пути нет. Т.е. если функция не использует монадический тип IO, то она не может заниматься вводом/выводом, и наоборот, если функция возвращает монадический тип IO, то она должна подчиняться парадигме действий в Haskell'е.

Программирование при помощи действий

Действия ввода/вывода являются обычными значениями в терминах Haskell'а. Т.е. действия можно передавать в функции в качестве параметров, заключать в структуры данных и вообще использовать там, где можно использовать данные языка Haskell. В этом смысле система операций ввода/вывода является полностью функциональной. Например, можно предположить список действий:

```
todoList    :: [IO ()]
todoList    = [putChar 'a',
               do putChar 'b'
                  putChar 'c',
               do c <- getChar
                  putChar c]
```

Этот список не возбуждает никаких действий, он просто содержит их описания. Для того, чтобы выполнить эту структуру, т.е. возбудить все ее действия, необходима некоторая функция (например, sequence_):

```
sequence_   :: [IO ()] -> IO ()
sequence_ []      = return ()
sequence_ (a:as)  = do  a
                        sequence as
```

Эта функция может быть полезна для написания функции putStr, которая выводит строку на экран:

```
putStr      :: String -> IO ()
putStr s    = sequence_ (map putChar s)
```

На этом примере видно явное отличие системы операций ввода/вывода языка Haskell от систем императивных языков. Если бы в каком-нибудь императивном языке была бы функция map, она бы выполнила кучу действий. Вместо этого в Haskell'е просто создается список действий (одно для каждого символа строки), который потом обрабатывается функцией sequence_ для выполнения.

Обработка исключений

Что делать, если в процессе операций ввода/вывода возникла неординарная ситуация? Например, функция getChar обнаружила конец файла. В этом случае произойдет ошибка. Как и любой продвинутый язык программирования Haskell предлагает для этих целей механизм обработки исключений. Для этого не используется какой-то специальный синтаксис, но есть специальный тип IOError, который содержит описания всех возникаемых в процессе ввода/вывода ошибок.

Обработчик исключений имеет тип (IOError → IO a), при этом функция catch ассоциирует (связывает) обработчик исключений с набором действий:

```
catch :: IO a -> (IOError -> IO a) -> IO a
```

Аргументами этой функции являются действие (первый аргумент) и обработчик исключений (второй аргумент). Если действие выполнено успешно, то просто возвращается результат без возбуждения обработчика исключений. Если же в процессе выполнения действия возникла ошибка, то она передается обработчику исключений в качестве операнда типа IOError, после чего выполняется сам обработчик.

Таким образом, можно написать более сложные функции, которые будут грамотно вести себя в случае выпадения ошибочных ситуаций:

```
getChar'    :: IO Char
getChar'    = getChar `catch` eofHandler
              where eofHandler e = if isEofError e then return \'n\ else ioError e

getLine'    :: IO String
getLine'    = catch getLine'' (\err -> return ("Error: " ++ show err))
              where getLine'' = do c <- getChar'
                                   id c == '\n' then return ""
                                               else do l <-  getLine'
                                                       return (c : l)
```

В этой программе видно, что можно использовать вложенные друг в друга обработчики ошибок. В функции getChar' отлавливается ошибка, которая возникает при обнаружении

символа конца файла. Если ошибка другая, то при помощи функции ioError она отправляется дальше и ловится обработчиком, который «сидит» в функции getLine'. Для определённости в Haskell'е предусмотрен обработчик исключений по умолчанию, который находится на самом верхнем уровне вложенности. Если ошибка не поймана ни одним обработчиком, который написан в программе, то её ловит обработчик по умолчанию, который выводит на экран сообщение об ошибке и останавливает программу.

Файлы, каналы и обработчики

Для работы с файлами Haskell предоставляет все возможности, что и другие языки программирования. Однако большинство этих возможностей определены в модуле IO, а не в Prelude, поэтому для работы с файлами необходимо явно импортировать модуль IO.

Открытие файла порождает обработчик (он имеет тип Handle). Закрытие обработчика инициирует закрытие соответствующего файла. Обработчики могут быть также ассоциированы с каналами, т.е. портами взаимодействия, которые не связаны напрямую с файлами. В Haskell'е предопределены три таких канала — stdin (стандартный канал ввода), stdout (стандартный канал вывода) и stderr (стандартный канал вывода сообщений об ошибках).

Таким образом, для использования файлов можно пользоваться следующими вещами:

```
type FilePath     = String
openFile          :: FilePath -> IOMode -> IO Handle
hClose            :: Handle -> IO ()
data IOMode       = ReadMode | WriteMode | AppendMode | ReadWriteMode
```

Далее приводится пример программы, которая копирует один файл в другой:

```
main = do fromHandle <- getAndOpenFile "Copy from: " ReadMode
          toHandle   <- getAndOpenFile "Copy to: " WriteMode
          contents   <- hGetContents fromHandle
          hPutStr toHandle contents
          hClose toHandle
          hClose fromHandle
          putStr "Done."

getAndOpenFile             :: String -> IOMode -> IO Handle
getAndOpenFile prompt mode =
    do putStr prompt
       name <- getLine
       catch (openFile name mode)
             (\_ -> do putStrLn ("Cannot open "++ name ++ "\n")
                       getAndOpenFile prompt mode)
```

Здесь использована одна интересная и важная функция — hGetContents, которая берёт содержимое переданного ей в качестве аргумента файла и возвращает его в качестве одной длинной строки.

Окончательные замечания

Получается так, что в Haskell'е заново изобретено императивное программирование...

В некотором смысле — да. Монада IO встраивает в Haskell маленький императивный подъязык, при помощи которого можно осуществлять операции ввода/вывода. И написание программ на этом подъязыке выглядит обычно с точки зрения императивных языков. Но есть существенное различие: в Haskell'е нет специального синтаксиса для

ввода в программный код императивных функций, все осуществляется на уровне функциональной парадигмы. В то же время опытные программисты могут минимизировать императивный код, используя монаду IO только на верхних уровнях своих программ, т.к. в Haskell’е императивный и функциональный миры чётко разделены между собой. В отличие от Haskell’а в императивных языках, в которых есть функциональные подъязыки, нет чёткого разделения между обозначенными мирами.

Лекция 8. «Конструирование функций»

Для конструирования функций используются различные формализмы, одним из которых является синтаксически-ориентированное конструирование. Чтобы применять последнюю методику, можно воспользоваться методом, который в свое время предложил Хоар.

Ниже приводится описание метаязыка, используемого для определения структур данных (в абстрактном синтаксисе):

1. **Декартово произведение**: Если C_1, ..., C_n — это типы, а C — это тип, состоящий из множества n-ок вида $<c_1, ..., c_n>$, $c_i \in C_i$, i = 1,n, то говорится, что C — декартово произведение типов C_1, ..., C_n и обозначается как $C = C_1 \times ... \times C_n$. При этом предполагается, что определены селекторы s_1, ..., s_n для типа C, что записывается как s_1, ..., s_n = selectors C.

   Таким же образом записывается конструктор g: g = constructor C. Конструктор — это функция, имеющая тип $(C_1 \to ... (C_n \to C) ...)$, т.е. для $c_i \in C_i$, i = 1,n : $g\ c_1 ... c_n = <c_1, ..., c_n>$.

   Будет считаться, что справедливо равенство:

   $$\forall x \in C : \text{constructor C } (s_1, x) ... (s_n, x) = x$$

   Это равенство называется аксиомой тектоничности. Кроме того, иногда эту аксиому записывают следующим образом:

   $$s_i \text{ (constructor C } c_1 ... c_n) = c_i$$

2. **Размеченное объединение**: Если C_1, ..., C_n — это типы, а C — это тип, состоящий из объединения типов C_1, ..., C_n, при условии выполнения «размеченности», то C называется размеченным объединением типов C_1, ..., C_n. Обозначается этот факт как $C = C_1 + ... + C_n$. Условие размеченности обозначает, что если из C взять какой-нибудь элемент c_i, то однозначно определяется тип этого элемента C_i. Размеченность можно определить при помощи предикатов P_1, ..., P_n таких, что:

   $$(x \in C) \,\&\, (x \in C_i) \Rightarrow (P_i\ x = 1) \,\&\, (\forall j \neq i : P_j\ x = 0)$$

   Размеченное объединение гарантирует наличие таких предикатов. Этот факт указывается записью: P_1, ..., P_n = predicates C. Ещё есть части типа, которые обозначаются так: N_1, ..., N_n = parts C.

Как видно, в представленном метаязыке используется два конструктора типов: × и +. Далее рассматриваются несколько примеров определения новых типов.

Пример 17. Формальное определение типа List (A).

List (A) = NIL + (A × List (A))
null, nonnull = predicates List (A)
NIL, nonNIL = parts List (A)
head, tail = selectors List (A)

prefix = constructor List (A)

Глядя на это описание (скорее — определение) типа, можно описать внешний вид функций, обрабатывающих структуры типа List (A):

Каждая функция должна содержать как минимум два клоза, первый обрабатывает NIL, второй — nonNIL соответственно. Этим двум частям типа List (A) в абстрактной записи соответствуют селекторы [] и (H : T). Два клоза можно объединить в один с использованием охраны. В теле второго клоза (или второго выражения охраны) обработка элемента T (или **tail** (L)) выполняется той же самой функцией.

Пример 18. Формальное определение типа List_str (A).

List_str (A) = A + List (List_str (A))
atom, nonAtom = predicates List_str (A)

Функции над List_str (A) должны иметь по крайней мере следующие клозы:

1°. A → when (atom (A))
2°. [] → when (null (L))
3°. (H : T) → **head** (L), **tail** (L)
3.1°. atom (**head** (L))
3.2°. nonAtom (**head** (L))

Пример 19. Формальное определение деревьев и лесов с помеченными вершинами.

Tree (A) = A × Forest (A)
Forest (A) = List (Tree (A))
root, listing = selectors Tree (A)
ctree = constructor Tree (A)

Пример 20. Формально определение деревьев с помеченными вершинами и дугами.

MTree (A, B) = A × MForest (A, B)
MForest (A, B) = List (Element (A, B))
Element (A, B) = B × MTree (A, B)
mroot, mlist = selectors MTree (A, B)
null, nonNull = predicates MForest (A, B)
arc, mtree = selectors Element (A, B)

Утверждается, что любая функция, работающая с типом MTree (A, B), может быть представлена только через упомянутые шесть операций независимо от того, как она реализована. Это утверждение можно проверить при помощи диаграммы (скорее, это гиперграф), на которой ясно видно, что к любой части типа MTree (A, B) можно «добраться», используя только эти шесть операций.

Для конструирования функций, обрабатывающих структуры данных MTree, необходимо ввести несколько дополнительных понятий и обозначений для них. Это делается для простоты. Начальная вершина, вершина MForest и вершина MTree (выходящая из Element) обозначаются как S_0, S_1 и S_2 соответственно. Для обработки этих вершин необходи-

мы три функции — f_0, f_1 и f_2, причем f_0 — это начальная функция, а две последних — рекурсивные.

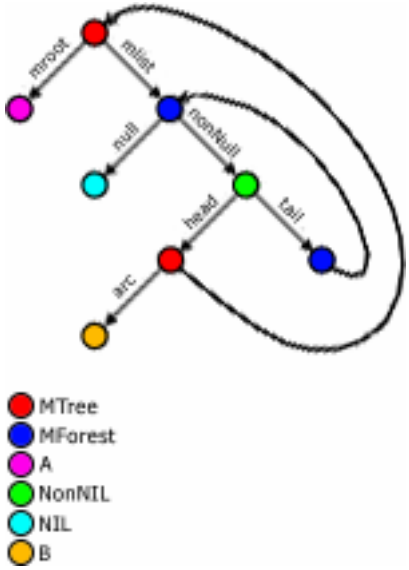

Рисунок 3. Гиперграф для представления структуры MTree

Конструирование функции f_0 выглядит просто — у этой функции один параметр T, который соответствует начальной вершине S_0. Две другие функции сконструировать сложнее.

Функция f_1 получает следующие параметры:

A — метка вершины;
K — параметр, содержащий результат обработки просмотренной части дерева;
L — лес, который необходимо обработать.

```
f1 A K L = g1 A K when null L
f1 A K L = f1 A (g2 (f2 A (arc (head L)) (mtree (tail L)) K) A (arc L) K) (tail L) otherwise
```

Эта функция организует режим просмотра дерева «сначала в глубину».

Функция f_2 получает следующие параметры (и это уже должно быть ясно из её вызова во втором клозе функции f_1):

A — метка вершины;
B — метка дуги;
T — поддерево для обработки;
K — результат обработки просмотренной части дерева.

```
f2 A B T K = f1 (mroot T) (g3 A B K) (mlist T)
```

Необходимо отметить, что это общий вид функций для обработки структур данных MTree. Реализация дополнительных функций g_1, g_2 и g_3 зависит от конкретной задачи. Теперь можно сконструировать и общий вид функции f_0:

```
f0 T = f1 (root T) k (mlist T)
```

где k — это начальное значение параметра K.

Для более глубокого закрепления методики конструирования функций можно рассмотреть конкретную реализацию функций работы с B-деревьями. Пусть для структуры дан-

ных BTree существует набор базисных операций, а сами деревья представляются в виде списков (особой роли представление не играет). Базисные операции следующие:

1°. cbtree A Left Right = [A, Left, Right]
2°. ctree = []
3°. root T = **head** T
4°. left T = **head** (**tail** T)
5°. right T = **head** (**tail** (**tail** T))
6°. empty T = (T == [])

Пример 21. Функция insert для вставки элемента в дерево.

```
insert (A:L) T = cbtree (A:L) ctree ctree when (empty T)
insert (A:L) T = cbtree (root T) (insert (A:L) (left T)) (right T) when (A < head (root T))
insert (A:L) T = cbtree (A:(L:tail (root T))) (left T) (right T) when (A == head (root T))
insert (A:L) T = cbtree (root T) (left T) (insert (A:L) (right T)) otherwise
```

Это реализация на абстрактном уровне.

Пример 22. Функция access для поиска элементов в B-дереве.

```
access A Emptic = []
access A ((A1:L)×Left×Right) = access A Left when (A < A1)
access A ((A1:L)×Left×Right) = access A Right when (A > A1)
access A ((A:L)×Left×Right) = L
access A (Root×Left×Right) = access A Right otherwise
```

В этом примере приведено две новых конструкции — абстрактный элемент Emptic, представляющий собой, по сути, пустое дерево, а также знак ×, при помощи которого абстрагируется декартово произведение, которое используется здесь вместо списочного представления. Но надо помнить, что это только абстрактный функциональный язык.

В представленных двух примерах существует одна проблема. При использовании написанных функций совершается огромное количество лишних копирований из одного места в памяти в другое. По сути дела это воссоздание нового дерева с новыми элементами (речь идет о функции insert). Этого можно избежать при использовании деструктивного присваивания.

Упражнения

1. Сконструировать функцию insert для вставки элемента в B-дерево, использующую деструктивное присваивание.

Ответы для самопроверки

1. Один из возможных вариантов функции insert с деструктивным присваиванием:

```
-- «Псевдо-функции» для деструктивного присваивания. В строгом функциональном языке (Haskell)
-- так делать нельзя. В Lisp’е есть возможность использовать деструктивное присваивание.

replace_root A T – функция добавления элемента в корень дерева
replace_left K (Root×Emptic×Right) => (Root×(K×Emptic×Emptic)×Right)
replace_right K (Root×Left×Emptic) => (Root×Left×(K×Emptic×Emptic))

-- Функция insert

insert K Emptic = cbtree K ctree ctree
```

```
insert (A:L) ((A1:L1)×Left×Right) = insert (A:L) Left when ((A < A1) & nonEmpty Left)
insert (A:L) ((A1:L1)×Emptic×Right) = replace_left (A:L) ((A1:L1)×Emptic×Right) when (A < A1)
insert (A:L) ((A1:L1)×Left×Right) = insert (A:L) Right when ((A > A1) & nonEmpty Right)
insert (A:L) ((A1:L1)×Left×Emptic) = replace_right (A:L) ((A1:L1)×Left×Emptic) when (A > A1)
insert A T = replace_root A T otherwise
```

Лекция 9. «Доказательство свойств функций»

Формальная задача: пусть имеется набор функций $\mathbf{f} = \langle f_1, ..., f_n \rangle$, определённых на областях $\mathbf{D} = \langle D_1, ..., D_n \rangle$. Требуется доказать, что для любого набора значений **d** имеет место некоторое свойство, т.е.:

$$\forall d \in D : P(f(d)) = 1,$$

где P — рассматриваемое свойство. Например:

1. $\forall d \in D : f(d) \geq 0$
2. $\forall d \in D : f(d) = f(f(d))$
3. $\forall d \in D : f(d) = f_1(d)$

Вводится принципиальное ограничение на рассматриваемые свойства — свойства только тотальные, т.е. справедливые для всей области **D**.

Далее рассматриваются некоторые виды областей определения **D**...

1. D — линейно упорядоченное множество.

Полуформально линейно упорядоченное множество можно определить как такое множество, для каждых элементов которого можно сказать, какой меньше (или больше), либо они равны, т.е.:

$$\forall d_1, d_2 \in D : (d_1 \geq d_2) \vee (d_1 \leq d_2).$$

В качестве примера можно привести множество целых чисел. Однако линейно упорядоченные множества встречаются в мире функционального программирования очень редко. Взять хотя бы простейшую структуру, которую очень любят обрабатывать в функциональном программировании — список. Для списков уже довольно сложно определить отношение порядка.

Для доказательства свойств функций на линейно упорядоченных множествах достаточно провести индукцию по данным. Т.е. достаточно доказать два пункта:

1°. $P(f(\varnothing))$ — базис индукции;

2°. $\forall d_1, d_2 \in D, d_1 \leq d_2 : P(f(d_1)) \Rightarrow P(f(d_2))$ — шаг индукции.

В силу того, что структуры данных редко образуют линейно упорядоченные множества, более эффективным способом оказывается применение метода индукции по построению типа **D**.

2. D — определяется как индуктивный класс

Из прошлой лекции известно, что индуктивный класс определяется через ввод базиса класса (это либо набор каких либо констант $d_{i=0,n} \in \mathbf{D}$, либо набор первичных типов $A_{i=0,n} : d \in A_i \Rightarrow d \in \mathbf{D}$. Также индуктивный класс определяется при помощи шага индукции — заданы конструкторы $g_1, ..., g_m$, определённые над A_i и **D**, и справедливо, что:

$$(a_i \in A_i) \wedge (x_j \in D) \Rightarrow g_k(a_i, x_j) \in D,\ k = \overline{1,m}.$$

В этом случае доказательство свойств функций также резонно проводить в виде индукции по даным. Метод индукции по даным в этом случае также очень прост:

1°. P (**f** (d)) необходимо доказать для базиса класса;
2°. Шаг индукции: P (**f** (d)) = P (**f** (g_i (d))).

Например, для доказательства свойств функций для списков (тип List (A)), достаточно доказать рассматриваемое свойство для двух следующих случаев:

1°. P (**f** ([])).
2°. $\forall a \in A, \forall L \in List(A) : P(f(L)) \Rightarrow P(f(a : L))$.

Доказательство свойств функций над S-выражениями (тип S-expr (A)) можно проводить на основе следующей индукции:

1°. $\forall a \in A : P(f(a))$.
2°. $\forall x, y \in S - expr(A) : P(f(x)) \wedge P(f(y)) \Rightarrow P(f(x : y))$.

Пример 23. Доказать, что $\forall L \in List(A) : L * [] = L$.

Для доказательства этого свойства можно использовать только определение типа List (A) и самой функции append (в инфиксной записи используется символ *).

1°. L = [] : [] * [] = [] = L. Базис индукции доказан.
2°. $\forall a \in A, \forall L \in List(A) : L * [] = L$. Теперь пусть применяется конструктор: a : L. Необходимо доказать, что (a : L) * [] = a : L. Это делается при помощи применения второго клоза определения функции append:

$$(a : L) * [] = a : (L * []) = a : (L) = a : L.$$

Пример 24. Доказать ассоциативность функции append.

Т.е. необходимо доказать, что для любых трех списков L_1, L_2 и L_3 имеет место равенство (L_1 * L_2) * L_3 = L_1 * (L_2 * L_3). При доказательстве индукция будет проводиться по первому операнду, т.е. списку L_1:

1°. L_1 = []:
([] * L_2) * L_3 = (L_2) * L_3 = L_2 * L_3.
[] * (L_2 * L_3) = (L_2 * L_3) = L_2 * L_3.

2°. Пусть для списков L_1, L_2 и L_3 ассоциативность функции append доказана. Необходимо доказать для (a : L_1), L_2 и L_3:
((a : L_1) * L_2) * L_3 = (a : (L_1 * L_2)) * L_3 = a : ((L_1 * L_2) * L_3).
(a : L_1) * (L_2 * L_3) = a : (L_1 * (L_2 * L_3)).

Как видно, последние два выведенных выражения равны, т.к. для списков L_1, L_2 и L_3 ассоциативность полагается доказанной.

Пример 25. Доказательство тождества двух определений функции reverse.

Определение 1:

```
reverse [] = []
reverse (H : T) = (reverse T) * [H]
```

Определение 2:

```
reverse' L = rev L []

rev [] L = L
rev (H : T) L = rev T (H : L)
```

Видно, что первое определение функции обращения списков — это обычное рекурсивное определение. Второе же определение использует аккумулятор. Требуется доказать, что:

$$\forall L \in List(A) : reverse\, L = reverse'\, L .$$

1°. Базис — L = []:
reverse [] = [].
reverse’ [] = rev [] [] = [].

2°. Шаг — пусть для списка L тождество функций reverse и reverse’ доказано. Необходимо доказать его для списка (H : L).

reverse (H : L) = (reverse L) * [H] = (reverse’ L) * [H].
reverse’ (H : L) = rev (H : L) [] = rev L (H : []) = rev L [H].

Теперь необходимо доказать равенство двух последних выведенных выражений для любых списков над типом A. Это также делается по индукции:

2.1°. Базис — L = []:
(reverse’ []) * [H] = (rev [] []) * [H] = [] * [H] = [H].
rev [] [H] = [H].

2.2°. Шаг — L = (A : T):
(reverse’ (A : T)) * [H] = (rev (A : T) []) * [H] = (rev T (A : [])) * [H] = (rev T [A]) * [H].
rev (A : T) [H] = rev L (A : H).

Здесь произошло выпадение в дурную бесконечность. Если дальше пытаться проводить доказательство по индукции для новых выведенных выражений, то эти самые выражения будут все усложняться и усложняться. Но это не причина для того, чтобы отчаиваться, ибо доказательство всё равно можно провести. Надо просто придумать некую «индукционную гипотезу», как это было сделано в предыдущем примере.

Индукционная гипотеза: (reverse’ L_1) * L_2 = rev L_1 L_2. Эта индукционная гипотеза является обобщением выражения (reverse’ L) * [H] = rev L [H].

Базис индукции для этой гипотезы очевиден. Шаг индукции в применении к выражению в пункте 2.2 выглядит следующим образом:
(reverse’ (A : T)) * L_2 = (rev (A : T) []) * L_2 = (rev T [A]) * L_2 = ((reverse’ T) * [A]) * L_2 = = (reverse’ T) * ([A] * L_2) = (reverse’ T) * (A : L_2).
rev (A : T) L_2 = rev T (A : L_2) = (reverse’ T) * (A : L_2).

Что и требовалось доказать.

Общий вывод: в общем случае для доказательства свойств функций методом индукции может потребоваться применение некоторых эвристических шагов, а именно введение индукционных гипотез. Эвристический шаг — это формулирование утверждения, которое

ниоткуда не следует. Таким образом, доказательство свойств функций есть своего рода творчество.

Лекция 10. «Формализация Функционального Программирования на основе λ-исчисления»

- Объект изучения: множество определений функций.
- Предположение: будет считаться, что любая функция может быть определена при помощи некоторого λ-выражения.
- Цель исследования: установить по двум определениям функций <f_1> и <f_2> их тождество на всей области определения — $\forall x : f_1(x) = f_2(x)$. (Здесь использована такая нотация: f — некоторая функция, <f> — определения этой функции в терминах λ-исчисления).

Проблема заключается в том, что обычно при описании функций задается интенсионал этих функций, а потом требуется установить экстенсиональное равенство. Под экстенсионалом функции понимается её график (ну или таблица в виде множества пар <аргумент, значение>). Под интенсионалом функции понимаются правила вычисления значения функции по заданному аргументу.

Возникает вопрос: как учесть семантику встроенных функций при сравнении их экстенсионалов (т.к. явно определения этих функций не известны)? Варианты ответов:

1. Можно попытаться выразить семантику встроенных функций при помощи механизма λ-исчисления. Этот процесс можно довести до случая, когда все встроенные функции не содержат непроинтерпретированных операций.
2. Говорят, что <f_1> и <f_2> семантически равны (этот факт обозначается как $\models f_1 = f_2$), если $f_1(x) = f_2(x)$ при любой интерпретации непроинтерпретированных идентификаторов.

Понятие формальной системы

Формальная система представляет собой четверку:

P = <V, Ф, A, R>, где

V — алфавит.
Ф — множество правильно построенных формул.
A — аксиомы (при этом $A \subseteq Ф$).
R — правила вывода.

В рассматриваемой задаче формулы имеют вид ($t_1 = t_2$), где t_1 и t_2 — λ-выражения. Если некоторая формула выводима в формальной системе, то этот факт записывается как ($\vdash t_1 = t_2$).

Говорят, что формальная система корректна, если ($\vdash t_1 = t_2$) $\Rightarrow$ ($\models t_1 = t_2$).

Говорят, что формальная система полна, если ($\models t_1 = t_2$) $\Rightarrow$ ($\vdash t_1 = t_2$).

Семантическое определение понятия «конструкция» (обозначение — Exp):

1°. $v \in Id \Rightarrow v \in Exp$.

2°. $v \in Id, E \in Exp \Rightarrow \lambda v.E \in Exp$.

3°. $E, E' \in Exp \Rightarrow (EE') \in Exp$.

4°. $E \in Exp \Rightarrow (E) \in Exp$.

5°. Никаких других Exp нет.

Примечание: Id — множество идентификаторов.

Говорят, что v свободна в M ∈ Exp, если:

1°. M = v.
2°. M = (M_1M_2), и v свободна в M_1 или в M_2.
3°. M = λv'.M', и v ≠ v', и v свободна в M'.
4°. M = (M'), и v свободна в M'.

Множество идентификаторов v, свободных в M, обозначается как FV (M).

Говорят, что v связана в M ∈ Exp, если:

1°. M = λv'.M', и v = v'.
2°. M = (M_1M_2), и v связана в M_1 или в M_2 (т.е. один и тот же идентификатор может быть свободен и связан в Exp).
3°. M = (M'), и v связана в M'.

Пример 26. Свободные и связанные идентификаторы.

1. M = v. v — свободна.
2. M = λx.xy. x — связана, y — свободна.
3. M = (λv.v)v. v входит в это выражение как свободно, так и связанно.
4. M = VW. V и W — свободны.

Правило подстановки: подстановка в выражение E выражения E' вместо всех свободных вхождений переменной x обозначается как E[x ← E']. Во время подстановки иногда случается так, что получается конфликт имён, т.е. коллизия переменных. Примеры коллизий:

(λx.yx)[y ← λz.z] = λx.(λz.z)x = λx.x — корректная подстановка
(λx.yx)[y ← xx] = λx.(xx)x — коллизия имён переменных
(λz.yz)[y ← xx] = λz.(xx)z — корректная подстановка

Точное определение базисной подстановки:

1°. x[x ← E'] = E'
2°. y[x ← E'] = y
3°. (λx.E)[x ← E'] = λx.E
4°. (λy.E)[x ← E'] = λy.E[x ← E'], при условии, что y ∉ FV (E')
5°. (λy.E)[x ← E'] = (λz.E[y ← z])[x ← E'], при условии, что y ∈ FV (E')
6°. (E_1E_2)[x ← E'] = (E_1[x ← E']E_2[x ← E'])

Построение формальной системы

Таким образом, сейчас уже все готово для того, чтобы перейти к построению формальной системы, определяющей Функциональное Программирование в терминах λ-исчисления.

Правильно построенные формулы выглядят так: Exp = Exp.

Аксиомы:

$\vdash \lambda x.E = \lambda y.E[x \leftarrow y]$ (α)

$\vdash (\lambda x.E)E' = E[x \leftarrow E']$ (β)

$\vdash t = t$, в случае, если t — идентификаторы (ρ)

Правила вывода:

$t_1 = t_2 \Rightarrow t_1t_3 = t_2t_3$ (μ)

$t_1 = t_2 \Rightarrow t_3t_1 = t_3t_2$ (ν)

$t_1 = t_2 \Rightarrow t_2 = t_1$ (σ)

$t_1 = t_2, t_2 = t_3 \Rightarrow t_1 = t_3$ (τ)

$t_1 = t_2 \Rightarrow \lambda x.t_1 = \lambda x.t_2$ (ξ)

Пример 27. Доказать выводимость формулы: (λx.xy)(λz.(λu.zu))v = (λv.yv)v

$(\lambda x.xy)(\lambda z.(\lambda u.zu))v = (\lambda v.yv)v$

(μ): $(\lambda x.xy)(\lambda z.(\lambda u.zu)) = (\lambda v.yv)$

(β): $(\lambda z.(\lambda u.zu))y = (\lambda v.yv)$

(α): $\lambda u.yu = \lambda v.yv$ — выводима.

Во втором варианте формализации Функционального Программирования можно воспользоваться не свойством симметричности отношения «=», а свойством несимметричности отношения «→».

Во второй формальной системе правильно построенные формулы выглядят абсолютно так же, как и в первом варианте. Однако аксиомы принимают несколько иной вид:

$\vdash \lambda x.M \rightarrow \lambda y.M[x \leftarrow y]$ (α')

$\vdash (\lambda x.M)N \rightarrow M[x \leftarrow N]$ (β')

$\vdash M \rightarrow M$ (ρ')

Правило вывода во втором варианте формальной системы одно:

$t_1 \rightarrow t_1', t_2 \rightarrow t_2' \Rightarrow t_1t_2 \rightarrow t_1't_2'$ (π)

По существу это правило вывода гласит, что в любом выражении можно выделить вхождения подвыражения и заменить их все любой редукцией из этого подвыражения.

Определения:

- Выражение вида λx.M называется α-редексом.
- Выражение вида (λx.M)N называется β-редексом.
- Выражения, не содержащие β-редексов, называются выражениями в нормальной форме.

Несколько теорем (без доказательств):

- $\vdash E_1 = E_2 \Rightarrow E_1 \to E_2 \vee E_2 \to E_1$.
- $E \to E_1 \wedge E \to E_2 \Rightarrow \exists F : E_1 \to F \wedge E_2 \to F$. (Теорема Чёрча-Россера).
- Если E имеет нормальные формы E_1 и E_2, то они эквивалентны с точностью до α-конверсии, т.е. различаются только обозначением связанных переменных.

Стратегия редукции

1°. Нормальная редукционная стратегия. На каждом шаге редукции выбирается текстуально самый левый β-редекс. Доказано, что нормальная редукционная стратегия гарантирует получение нормальной формы выражения, если она существует.

2°. Аппликативная редукционная стратегия. На каждом шаге редукции выбирается β-редекс, не содержащий внутри себя других β-редексов. Далее будет показано, что аппликативная редукционная стратегия не всегда позволяет получить нормальную форму выражения.

Пример 28. Редукция выражения M = (λy.x)(EE), где E = λx.xx

1°. НРС: $(\lambda y.x)(EE) = (\lambda y.x)[y \leftarrow EE] = x$.

2°. АРС: $(\lambda y.x)(EE) = (\lambda y.x)((\lambda x.xx)(\lambda x.xx)) = (\lambda y.x)((\lambda x.xx)(\lambda x.xx)) = \ldots$

В этом примере видно, как аппликативная редукционная стратегия может привести к выпадению в дурную бесконечность. Получить нормальную форму выражения M в случае применения аппликативной редукционной стратегии невозможно.

Примечание: красным шрифтом выделен β-редекс, редуцируемый следующим шагом.

Пример 29. Редукция выражения M = (λx.xyxx)((λz.z)w)

1°. НРС: $(\lambda x.xyxx)((\lambda z.z)w) = ((\lambda z.z)w)y((\lambda z.z)w)((\lambda z.z)w) = wy((\lambda z.z)w)((\lambda z.z)w) = wyw((\lambda z.z)w) = wyww$.

2°. АРС: $(\lambda x.xyxx)((\lambda z.z)w) = (\lambda x.xyxx)w = wyww$.

В программировании нормальная редукционная стратегия соответствует вызову по имени. Т.е. аргумент выражения не вычисляется до тех пор, пока к нему не возникнет обращения в теле выражения. Аппликативная редукционная стратегия соответствует вызову по значению.

Соответствие между вычислениями функциональных программ и редукцией

Работа интерпретатора описывается несколькими шагами:

1. В выражении необходимо выделить некоторое обращение к рекурсивной или встроенной функции с полностью означенными аргументами. Если выделенное обращение к встроенной функции существует, то происходит его выполнение и возврат к началу первого шага.

2. Если выделенное на первом шаге обращение к рекурсивной функции, то вместо него подставляется тело функции с фактическими параметрами (т.к. они уже означены). Далее происходит переход на начало первого шага.

3. Если больше обращений нет, то происходит остановка.

В принципе, вычисления в функциональной парадигме повторяют шаги редукции, но дополнительно содержат вычисления встроенных функций.

Представление определений функций в виде λ-выражений

Показано, что любое определение функции можно представить в виде λ-выражения, не содержащего рекурсии. Например:

fact = λn.if n == 0 then 1 else n * fact (n – 1)

То же самое определение можно описать при помощи использования некоторого функционала:

fact = **(λf.λn.if n == 0 then 1 else n * f (n – 1))** fact

Жирным шрифтом в представленном выражении выделен функционал F. Таким образом, функцию вычисления факториала можно записать так: fact = F fact. Можно видеть, что любое рекурсивное определение некоторой функции f можно представить в таком виде:

$$f = F\ f$$

Это выражение можно трактовать как уравнение, в котором рекурсивная функция f является неподвижной точкой функционала F. Соответственно интерпретатор функционального языка можно в таком же ключе рассматривать как численный метод решения этого уравнения.

Можно сделать предположение, что этот численный метод (т.е. интерпретатор) в свою очередь может быть реализован при помощи некоторой функции Y, которая для функционала F находит его неподвижную точку (соответственно определяя искомую функцию) — f = Y F.

Свойства функции Y определяются равенством:

$$Y\ F = F\ (Y\ F)$$

Теорема (без доказательства):

Любой λ-терм имеет неподвижную точку.

λ-исчисление позволяет выразить любую функцию через чистое λ-выражение без использования встроенных функций. Например:

1°. prefix = λxyz.zxy
head = λp.p(λxy.x)
tail = λp.p(λxy.y)

2°. Моделирование условных выражений:

$True \equiv \lambda xy.x$

$False \equiv \lambda xy.y$

if B then M else $N \equiv BNM$, где $B \in \{True, False\}$.

Лекция 11. «Трансформация программ»

Под программой P на некотором языке L понимается некоторый текст на L. В случае функционального языка под программой понимается набор клозов. Семантика же языка L определяется, если задан интерпретатор этого языка. Интерпретатор определяется формулой:

$$\mathrm{Int}\ (P, d) = d'$$

где:

P — программа;
d — исходные данные;
d' — выходные данные.

Если интерпретатор Int представлен в виде каррированной функции f такой, что f P d = d', тогда определение f = M f, а точнее функционал M называется денотационной семантикой языка L. В этом случае имеет смысл λ-выражение: $f\ P = \lambda d.M' : D \to D'$. При этом частичную функцию f P можно рассматривать как функцию f_P одного аргумента, которая есть функция, реализующая программу P. Как было показано в предыдущей лекции, можно построить рекурсивное определение вида: $f_P = M_P\ f_P$. Такой вид изначально имеют все программы на функциональном языке. Но эту запись можно понимать двояко:

- Это определение можно понимать просто как строку символов, подаваемую на вход интерпретатора. Функция, вычисляемая интерпретатором по тексту f = M f, обозначается как f_{int}.
- f = M f — есть чистое математическое определение функции f. Решение этого уравнения обозначается как f_{mat}.

Резонный вопрос: каково соотношение этих двух функций — f_{int} и f_{mat}? Надо полагать, что корректный интерпретатор как раз вычисляет f_{mat}.

Определение: говорят, что функция f_1 менее определена, чем функция f_2 (обозначается как $f_1 \subseteq f_2$), если $\forall x : f_1 x = y \Rightarrow f_2 x = y$. Если для двух функций одновременно выполняется $f_1 \subseteq f_2$ и $f_2 \subseteq f_1$, то имеет место тождество функций.

Как правило, $f_{\mathrm{int}} \subseteq f_{mat}$ — это происходит потому что, обычно интерпретатор реализует аппликативную стратегию редукции. Однако в дальнейших рассуждениях будет полагаться тождество функций f_{int} и f_{mat}, поэтому тексты программ будут рассматриваться как математическое определение функций. Тогда эквивалентное преобразование функциональных программ есть просто эквивалентные преобразования специального вида математических определений функций.

Трансформация программ — это просто синтаксические преобразования, во время которых совершенно не затрагивается семантика программ, т.к. программа воспринимается просто как набор символов. Обозначение того факта, что один текст f_1 получаем при помощи синтаксических трансформаций другого текста f_2, выглядит следующим образом: $f_1 \vdash f_2$.

Говорят, что преобразование корректно, если $f_1 \subseteq f_2$.

Говорят, что преобразование эквивалентно, если $f_1 \equiv f_2$.

Далее вводится еще несколько специальных обозначений:

1. Имеется исходный набор клозов (т.е. объектов преобразования), и этот набор будет обозначаться либо DEF, либо SPEC.
2. Клозы, описывающие функцию, которая содержит отображения из исходных клозов, обозначается как INV.
3. Некоторые равенства, выражающие свойства внутренних (зарезервированных) функций, обозначаются как LOW.

Определение: пусть F (X) — некоторое выражение (равенство), в котором выделены все свободные вхождения терма X. Тогда F [X ← M] называется примером F. При этом считается, что M — это некоторое выражение.

Виды преобразований

1°. Конкретизация (instantiation) — INS.

$$\frac{E_1(X) = E_2(X)}{E_1[X \leftarrow N] = E_2[X \leftarrow N]}$$

2°. Преобразование без названия :)

$$\frac{M(Y) = N(Y), E_1 = E_2[X \leftarrow M'], M' = M[Y \leftarrow G]}{E_1 = E_2[X \leftarrow N'],\ где\ N' = N[Y \leftarrow G]}$$

3°. Развертка (unfolding) — UNF.

$$\frac{M(Y) = N(Y), E_1 = E_2(M'), M' = M[Y \leftarrow G]}{E_1 = E_2(N'),\ где\ N' = N[Y \leftarrow G]}$$

4°. Свертка (folding) — FLD.

$$\frac{M(Y) = N(Y), E_1 = E_2(N'), N' = N[Y \leftarrow G]}{E_1 = E_2(M'),\ где\ M' = M[Y \leftarrow G]}$$

5°. Закон (law) — LAW.

$$\frac{M(Y) = N(Y), E_1 = E_2(M'), M' = M[Y \leftarrow G]}{E_1 = E_2(N'),\ где\ N' = N[Y \leftarrow G]}$$

6°. Абстракция и аппликация (abstraction & application) — ABS.

$$\frac{M[X \leftarrow G] = (\lambda X.M)G, E_1 = E_2(M[X \leftarrow G]}{E_1 = E_2((\lambda X.M)G)}$$

Пример 30. Преобразование функции length.

```
length []  = 0                                          1 (DEF)
length (H:T) = 1 + length T                             2 (DEF)
length_2 L1 L2 = length L1 + length L2                  3 (SPEC)
```

```
length_2 []  L     = length [] + length L                  4 (INS 3)
                   = 0 + length L                          5 (UNF 1)
                   = length L                              6 (LAW +) (*)

length_2 (H:T) L   = length (H:T) + length L               7 (INS 3)
                   = (1 + length T) + length L             8 (UNF 2)
                   = 1 + (length T + length L)             9 (LAW +)
                   = 1 + length_2 T L                      10 (FLD 3) (**)
```

Теперь остаётся взять два полученных клоза (*) и (**) и составить из них новое рекурсивное определение новой функции, не использующее вызова старой функции:

```
length_2 [] L = length L
length_2 (H:T) L = 1 + length_2 T L
```

Однако следует отметить, что выбор новых клозов для формирования нового определения требует дополнительных исследований, а не выполняется автоматически.

Подобная трансформация определений функций часто будет приводить к уменьшению сложности определения функций. Например, для функции, вычисляющей N-ое число Фиббоначи можно построить определение, сложность вычисления которого линейно зависит от N, а не по закону Фиббоначи, как это есть для обычного определения.

Но трансформации следует делать обдуманно, т.к. можно придти к бесконечным циклам шагов FLD и UNF. Чтобы при трансформации не придти к абсурду, необходимо следить за тем, чтобы в процессе преобразования общность получаемых выражений не увеличивалась. Т.е. трансформацию надо осуществлять от общего к частному.

Второй закон информатики

- Существуют неразрешимые задачи.
- Не существует эффективной реализации для декларативных языков, если они универсальны.

Концепция трансформационного синтеза: позволить программисту писать определения функций, не заботясь об их эффективности.

Однако было доказано, что по языку спецификаций невозможно выработать (т.е. трансформировать исходный текст) вариант функции, работающий эффективно. Если в качестве языка спецификаций рассматривать функциональный язык, то становится понятно, что программист сам должен заботиться об эффективности своих программ — концепция трансформационного синтеза здесь не пройдет.

Частичные вычисления

Пусть P и S — два языка, работающих со строками символов (это не нарушает общности рассуждений), а **P** и **S** — множества синтаксически правильных программ на соответствующих языках. D — домен всевозможных символьных последовательностей.

$P :: D \to (D^* \to D)$

Если p — программа из **P**, то:

$P(p) :: D^* \to D$

$P(p)(<d_1, ..., d_n>) = d$, и $d \in D$

P (r) (<y_1, ..., y_n>) = P (p) (<d_1, ..., d_m, y_1, ..., y_n>)

В последнем равенстве переменными y_i обозначены неизвестные данные. Для программы p эти n переменных представляют остаточный код.

Частичным вычислителем MIX называется программа из **P** (хотя, частичный вычислитель может быть реализован на любом языке), такая что:

$$\forall p \in \mathbf{P}, p(x_1,...,x_m,x_{m+1},...,x_n) : P(MIX)(< p,d_1,...,d_m >) = r .$$

Т.е. MIX — это программа, которая, получив некоторую исходную программу и данные для её известных параметров, выдаёт остаточную программу для исходной.

Интерпретатором языка S называется программа INT ∈ **P**, такая что:

$$\forall s \in \mathbf{S}, \langle d_1,...,d_n \rangle \in D^* : P(INT)(< s,d_1,...,d_n >) = S(s)(< d_1,...,d_n >) .$$

Компилятором языка S называется программа COMP ∈ **P**, такая что:

$$P(COMP)(< s >) = TARGET$$
$$P(TARGET)(< d_1,...,d_n >) = S(s)(< d_1,...,d_n >) .$$

Или, что то же:

$$P(P(COMP)(< s >)(< d_1,...,d_n >) = S(s)(< d_1,...,d_n >) .$$

Компилятором компиляторов языка S называется программа COCOM ∈ **P**, такая что:

$$P(COCOM)(< INT >) = COMP$$
$$P(P(P(COCOM)(< INT >))(< s >))(< d_1,...,d_n >) = S(s)(< d_1,...,d_n >) .$$

Проекции Футаморы

- TARGET = P (MIX) (<INT, s>)
- COMP = P (MIX) (<MIX, INT>)
- COCOM = P (MIX) (<MIX, MIX>)

Строго говоря, эти три утверждения являются теоремами, которые, однако, вполне легко доказываются при помощи определений, данных выше.