Функциональное программирование

Введение

Функциональное программирование – это ветвь программирования, при котором программирование ведется с помощью определения функций. Вы можете сказать, что любой программист давно уже программирует с помощью функций, да и не только функций, а еще и процедур, циклов, модулей, объектов,... Что же нового в функциональном программировании по сравнению с тем, что мы уже давно знаем?

Все это хорошо, но в функциональном программировании нет ни процедур, ни циклов, нет даже переменных. Почти одни только функции. Но все-таки не спешите заканчивать чтение. Функциональное программирование обладает рядом очень существенных преимуществ, которые не только позволяют ему существовать наряду с традиционным программированием, но и иметь своих поклонников, свою нишу задач и хорошие перспективы на будущее.

Традиционное программирование родилось в 40-х годах 20 века, когда велась разработка первых электронно-вычислительных машин (ЭВМ). Его основой послужила концепция фон Неймана о хранимой программе автоматических вычислений по заданному алгоритму. Существенными чертами такой программы служили, во-первых, строгая последовательность в выполнении ее элементарных шагов и, во-вторых, возможность хранения и изменения программы наряду с данными для вычислений в общей с ними памяти. Таким образом, программа сама становилась объектом обработки вместе с арифметическими значениями, над которыми, собственно, и должны были производиться все действия.

Когда появились первые компьютеры, то их устройство следовало принципам, сформулированным фон Нейманом (архитектура фон Неймана). Исполнение программ в первых электронно-вычислительных машинах сводилось к выполнению арифметическим устройством (позже оно стало называться *процессором*) элементарных шагов, называемых *командами*, которые строго последовательно производили определенные действия над арифметическими значениями или другими командами, хранящимися в *оперативной памяти* компьютера. Изменять команды нужно было для того, чтобы организовать циклическое повторение участков программы и своевременное завершение таких повторений.

В конце 50-х годов 20 века появились первые языки программирования высокого уровня, в них уже произошел существенный отход от принципов фон Неймана. Во-первых, программа раз и навсегда была отделена от данных. Во-вторых, во время исполнения программы ее текст оставался неизменным, а организация циклического повторения команд в ходе исполнения программы была возложена на систему

программирования, которая уже и должна была перевести (*транслировать*) текст программы в систему команд компьютера так, чтобы ее исполнение происходило в соответствии с написанным текстом. Единственный принцип, остававшийся неизменным, был принцип последовательного исполнения, в соответствии с которым исполнение программы можно было разложить на строго последовательные элементарные шаги алгоритма. Поэтому до сих пор программирование в традиционном стиле часто называют «фон-Неймановским».

Со временем принцип последовательного исполнения стал серьезным препятствием для развития компьютерной техники. Самым узким местом вычислительных систем уже долгое время остается тот самый процессор, который последовательно исполняет элементарные команды. Конечно, скорость работы современных процессоров не сравнить со скоростью работы арифметических устройств первых ЭВМ, однако, производительность компьютеров сейчас, как и раньше, ограничена, в основном, именно этим центральным устройством. Именно скорость работы центрального процессора имеет решающее значение при определении общей производительности компьютера.

Скорость работы процессора стала зависеть уже не столько от его архитектуры и технологических элементов, сколько просто от его размеров, потому что на скорость работы решающее влияние стала оказывать скорость прохождения сигналов по цепям процессора, которая, как известно, не может превысить скорости света. Чем меньше процессор, тем быстрее смогут внутри него проходить сигналы, и тем больше оказывается конечная производительность процессора. Размеры процессора уменьшились многократно, однако, все труднее стало отводить от такого миниатюрного устройства вырабатываемое при работе его элементов тепло. Перед производителями вычислительной аппаратуры встал очень серьезный вопрос: дальнейшее повышение производительности стало почти невозможным без изменения основополагающего принципа всего современного программирования – последовательного исполнения команд.

Конечно, можно так спроектировать вычислительную систему, чтобы в ней могли одновременно работать несколько процессоров, но, к сожалению, это почти не дает увеличения производительности, потому что все программы, написанные на традиционных языках программирования, предполагают последовательное выполнение элементарных шагов алгоритма почти так же, как это было во времена фон Неймана. Время от времени предпринимаются попытки ввести в современные языки программирования конструкции для параллельного выполнения фрагментов кода, однако языки "сопротивляются". Оказывается, думать об организации параллельного выполнения фрагментов программы – это

довольно сложная задача, которая успешно решается человеком только для весьма ограниченных случаев.

Проблему можно решать различными способами. Во-первых, можно попробовать написать специальную программу, которая могла бы проанализировать имеющийся программный текст и автоматически выделить в ней фрагменты, которые можно выполнять параллельно. К сожалению, такой анализ произвольного программного кода очень труден. Последовательность выполнения шагов алгоритма очень трудно предсказать по внешнему виду программы, даже если программа «хорошо структурирована». Второй способ перейти к параллельным вычислениям – это создать такой язык программирования, в котором сам алгоритм имел бы не последовательную структуру, а допускал бы независимое исполнение отдельных частей алгоритма. Но против этого восстает весь накопленный программистами опыт написания программ.

Тем не менее, оказалось, что опыт написания программ, не имеющих строго последовательной структуры, на самом деле есть. Почти одновременно с первым "традиционным" языком программирования – *Фортраном* появился еще один совершенно непохожий на него язык программирования – *Лисп*, для которого последовательность выполнения отдельных частей написанной программы была несущественной. Ветвь программирования, начатая созданием Лиспа, понемногу развивалась с начала 60-х годов 20 века и привела к появлению целой плеяды очень своеобразных языков программирования, которые удовлетворяли всем требованиям, необходимым для исполнения программ несколькими параллельными процессорами. Во-первых, алгоритмы, записанные с помощью этих языков, допускают сравнительно простой анализ и формальные преобразования программ, а во-вторых, отдельные части программ могут исполняться независимо друг от друга. Языки, обладающие такими замечательными свойствами – это и есть языки функционального программирования.

Помимо своей хорошей приспособленности к параллельным вычислениям языки функционального программирования обладают еще рядом приятных особенностей. Программы на этих языках записываются коротко, часто много короче, чем в любом другом традиционном (императивном) языке. Описание алгоритмов в функциональном стиле сосредоточено не на том, ***как*** достичь нужного результата (в какой последовательности выполнять шаги алгоритма), а больше на том, ***что*** должен представлять собой этот результат.

Пожалуй, единственный серьезный недостаток функционального стиля программирования состоит в том, что этот стиль не универсальный. Многие действительно последовательные процессы, такие как поведение программных моделей в реальном времени, игровые и другие программы, организующие взаимодействие компьютера с человеком, не выразимы в

функциональном стиле. Тем не менее, функциональное программирование заслуживает изучения хотя бы еще и потому, что позволяет несколько по-иному взглянуть вообще на процесс программирования, а некоторые приемы программирования, которые, вообще говоря, предназначены для написания программ в чисто функциональном стиле, могут с успехом использоваться и в традиционном программировании.

Мы будем знакомиться с миром функционального программирования, начав с изучения одного из самых распространенных в настоящее время языков функционального программирования – языка *Haskell*.

Глава 1. Элементы функционального программирования на языке *Haskell*

1.1. Функциональный стиль программирования

Часто процесс исполнения программы можно представить схемой, показанной на рисунке 1.1.

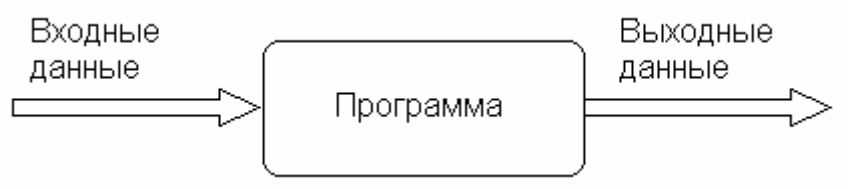

Рис. 1.1. Схема простой программы

Конечно, не любую программу можно представить в виде такой схемы, а только такую, которая, получив некоторые входные данные в начале своей работы, затем обрабатывает их, не общаясь с внешней средой, и в конце работы выдает некоторый результат. Часто такую схему работы программы называют «черным ящиком», подразумевая, что в этой схеме нас не интересует, как и в какой последовательности происходят те или иные действия в программе, а интересует только то, какой результат получается в зависимости от входных данных. Можно сказать, что при такой схеме работы результат находится в функциональной зависимости от исходных данных.

Можно привести много примеров простых и сложных программ, имеющих смысл и работающих по этой схеме. Например, программа аналитических преобразований выражений – входными и выходными данными здесь будут выражения, представленные в разной форме. Еще пример: компилятор с некоторого языка программирования – входными данными программы будет текст, написанный на этом языке программирования, а выходными данными – объектный код программы. Третий пример: программа составления расписания учебных занятий – исходные данные для такой программы – набор «пожеланий» к расписанию, а выходные данные – таблица, представляющая расписание занятий. В то же время многие программы выполняются не по схеме «черного ящика». Например, любая программа, организующая активное взаимодействие (диалог) с пользователем, не является преобразователем набора входных данных в выходные, если только не считать, что входными и выходными данными в этом случае является «среда», включающая в том числе и пользователя.

Всякая программа, написанная в функциональном стиле – это программа, представляющая собой функцию, аргументом которой служат входные данные из некоторого допустимого набора, и выдающую определенный результат. Обычно при этом подразумевается, что функция,

определяющая поведение программы, на одних и тех же входных данных всегда выдает один и тот же результат, то есть функция детерминированная. Можно заметить, что любая программа, содержащая запросы к пользователю (взаимодействующая с пользователем) – не детерминированная, поскольку ее поведение может зависеть от «поведения пользователя», то есть конкретных данных, введенных пользователем. В этой главе мы изучим основы языка функционального программирования *Haskell*, который используется для написания программ в виде функций. Отличительными особенностями функций, определяемых любым языком функционального программирования, являются следующие черты:

Каждая функция в программе выдает один и тот же результат на одном и том же наборе входных данных (аргументов функции), то есть результат работы функции является «повторяемым».

Вычисление функции не может повлиять на результат работы других функций, то есть функции являются «чистыми».

ЗАМЕЧАНИЕ

> Требование детерминированности функций, вообще говоря, не является обязательным. Иногда рассматривают и функции с недетерминированным результатом, при этом можно на основе таких функций строить полезные программы. Мы, однако, вопросы, касающиеся недетерминированных функций, рассматривать не будем.

Если программа состоит только из чистых функций, то порядок вычисления аргументов этих функций будет несущественным. Вообще, любые выражения, записанные в такой программе по отдельности, независимо друг от друга (вычисление значений отдельных элементов списка, полей кортежа и т.п.) могут вычисляться в любой последовательности или даже параллельно. При наличии нескольких независимых процессоров, работающих в общей памяти, вычисления, происходящие по программе, составленной только из вызовов чистых функций, легко распараллелить. Уже это свойство функционального стиля программирования привлекает к нему значительный интерес.

Можно ли писать программы в функциональном стиле на традиционном языке программирования? Конечно, да. Если программа представляет из себя набор чистых детерминированных функций, то она будет "функциональной" независимо от того, написана ли она на специальном языке функционального программирования *Haskell* или на традиционной *Java*. Рассмотрим, например, задачу вычисления числа вещественных корней заданного квадратного уравнения. Функция, решающая эту задачу должна, получив в качестве исходных данных коэффициенты квадратного уравнения, вычислить его дискриминант, а затем сформировать нужный результат в зависимости от знака

вычисленного дискриминанта. На языке *Java* функция может выглядеть следующим образом (листинг 1.1).

Листинг 1.1. Функция вычисления числа корней квадратного уравнения

```
int roots(double a, double b, double c) {
  double discr = b * b - 4 * a * c;
  if (discr < 0) return 0; else
  if (discr == 0) return 1; else
  return 2;
}
```

Эта функция, конечно же, детерминированная и «чистая», однако все же с точки зрения функционального стиля она имеет одну «неправильность». Дело в том, что в функции определяется и используется локальная переменная `discr`, которая используется для запоминания значения дискриминанта квадратного уравнения. В данном случае это никак не влияет на «чистоту» написанной функции, но если бы внутри функции были бы определены другие функции, то результат их работы мог бы быть различным в зависимости от того, когда они вызываются – до присваивания переменной `discr` нового значения или после него. Поэтому чисто функциональный стиль программирования предполагает полное отсутствие присваиваний в программах.

Кроме того, в чисто функциональных программах нет понятия последовательного вычисления. Каждая функция должна представлять собой суперпозицию обращений к другим функциям. В нашем же примере порядок вычислений задается строго, в нем предписывается, что сначала требуется вычислить значение дискриминанта и присвоить вычисленное значение переменной `discr`, потом проверить, верно ли, что полученная переменная имеет значение, меньшее нуля, и т.д.

Впрочем, в нашем случае исправить программу, приведя ее в соответствие с принципами функционального стиля программирования, довольно просто. Для этого определим две вспомогательные функции для вычисления дискриминанта квадратного уравнения и для вычисления знака вещественного числа (аналогичную стандартной `Math.signum`, но выдающую целый результат). В результате получится следующая программа (листинг 1.2).

Листинг 1.2. Функция вычисления корней квадратного уравнения

```
double discr (double a, double b, double c) {
  return b * b - 4 * a * c;
}
int sign (double x) {
  return x < 0 ? -1 : x == 0 ? 0 : 1;
}
int roots(double a, double b, double c) {
  return sign(discr(a, b, c)) + 1;
```

```
}
```

Эта программа уже действительно чисто функциональная. Обратите внимание также и на то, что вместо условных операторов мы в этой программе используем условные выражения, которые соединяют условиями не отдельные части последовательно выполняющейся программы, а отдельные подвыражения. Это тоже является характерной особенностью функционального стиля программирования.

Однако, убрав возможность присваивания значений переменным, мы тем самым сделали бессмысленным использование и одного из самых мощных средств традиционного программирования – циклов. Действительно, циклы имеют смысл только в том случае, если при каждом исполнении тела цикла внешняя среда (совокупность значений доступных в данный момент переменных) хотя бы немного, но меняется. В противном случае цикл либо не исполняется ни разу, либо будет продолжать выполняться бесконечно. Как же в отсутствие циклов написать хотя бы такую простую функцию, как вычисление факториала заданного натурального числа?

Альтернативой циклам являются рекурсивные функции, так что задачу написания функции для вычисления факториала числа все же можно решить на *Java* в чисто функциональном стиле (листинг 1.3):

Листинг 1.3. Функция вычисления факториала натурального числа

```
int factorial (int n) {
  return n == 0 ? 1 : n * factorial(n-1);
}
```

Тогда, может быть, для того, чтобы писать в функциональном стиле, нужно просто взять привычный язык, ну, скажем, ту же *Java*, и «выкинуть» из него переменные, присваивания и циклы, вообще любые «последовательные» операторы? Конечно, в результате действительно получится язык, на котором программы будут всегда написаны «в функциональном стиле», но получившийся язык будет слишком бедным для того, чтобы на нем можно было писать действительно полезные программы. В «настоящих» языках функционального программирования с функциями можно работать значительно более гибко, чем позволяется в традиционных языках программирования. Рассмотрим следующую простую задачу.

Пусть требуется определить функцию, с помощью которой можно создавать суперпозицию двух вещественных функций одного вещественного аргумента. Другими словами, требуется описать инструмент, с помощью которого из двух произвольных функций `f(x)` и `g(x)` можно было бы получить новую функцию `fg(x)`, результат работы которой определялся бы уравнением `fg(x) = f(g(x))`. Конечно, решение этой задачи было бы очень полезным в ситуации, когда практически единственным инструментом программирования является

определение и вызов различных функций. На первый взгляд кажется, что задачу легко запрограммировать на «чуть-чуть расширенной» *Java*, в которой достаточно лишь разрешить описывать тип функциональных значений и возвращать функции в качестве результата работы других функций (возможно, эта возможность появится в JDK 7, однако, на момент написания книги эта информация еще не подтверждена окончательно). Еще одна «мелочь», которую мы будем использовать в расширенной *Java* – это возможность определения нового идентификатора типа с помощью `typedef`. Поставленная задача может быть решена на расширенной *Java* следующим образом (листинг 1.4).

Листинг 1.4. Реализация суперпозиции на Java

```
typedef Func = #double(double);

Func comp(Func f, Func g) {
  final Func result = #double(double x) { return f(g(x)); };
  return result;
}
```

К сожалению, не очень понятно, сможет ли такая программа работать даже в расширенном варианте языка *Java*. Причина этого заключается в том, что функция `result`, определенная внутри функции `comp`, использует глобальные для нее значения функциональных аргументов `f` и `g`. После выхода из функции `comp` связь этих аргументов с фактическими значениями может быть потеряна. Попробуем, скажем, выполнить вызов следующей функции, полученной с помощью `comp`

```
comp(sin, cos)(3.14)
```

При вызове `comp(sin, cos)` будет образовано новое функциональное значение, представленное функцией `result` в определении функции `comp`. Однако функция `result` ссылается на аргументы `f` и `g`, а их связь с функциями `sin` и `cos` может быть потеряна после выхода из функции `comp`. В этом случае вызов новой функции с аргументом `3.14` не сможет проработать правильно. Для того, чтобы эта программа работала правильно, необходимо кардинально пересмотреть подход к образованию функциональных значений.

Описанное поведение заложено в структуре традиционных языков, рассчитанных на последовательное исполнение шагов алгоритма, очень глубоко, и для того, чтобы можно было писать программы для решения задач вроде только что описанной, требуется определять языки с совершенно другими свойствами. Этой цели и служат специализированные языки функционального программирования. В частности, в этих языках функции являются значениями «первого класса», то есть с ними можно проделывать все то же, что и с другими «обычными» значениями: над ними можно выполнять операции и применять к ним другие функции как к

аргументам; можно написать функцию, результатом работы которой будет также функция; функции могут быть элементами сложных структур данных и т.п.

Кстати, программу, содержащую функцию, похожую на `comp`, все же можно написать и на чистом языке *Java*. Для этого надо использовать вместо представлений в виде функции представления в виде реализаций некоторого специального интерфейса. Приведем текст такой программы в листинге 1.5 полностью, хотя и отметим при этом, что программа не отличается ясностью.

Листинг 1.5. Реализация суперпозиции на чистом языке Java

```
// Интерфейс для представления функциональных значений
public interface Func {
  double func(double x);
}
// Основной класс
public class Super {
  /** Функция композиции получает в качестве аргументов две
   * «функции» и выдает в качестве результата их композицию.
   * @param f первая из исходных функций
   * @param g вторая функция
   * @return композиция двух функций
   */
  static public Func comp(final Func f, final Func g) {
    return new Func() {
      public double func(double x) {return f.func(g.func(x));}
    };
  }

  public static void main(String args[]) {
    // Объект, представляющий функцию «синус»
    Func Sin = new Func() {
      public double func(double x) { return Math.sin(x); }
    };
    // Объект, представляющий функцию «косинус»
    Func Cos = new Func() {
      public double func(double x) { return Math.cos(x); }
    };
    // Образуем суперпозицию двух функций
    // и используем ее для вычислений.
    Func SinCos = comp(Sin, Cos);
    System.out.println(SinCos.func(Math.PI));
  }
}
```

Примеры решения задач

Задание 1. Написать программу для вычисления приближенного значения числа e по формуле для разложения e^x в ряд Тейлора.

Предложить программы на языке *Java*, написанные в традиционном императивном и функциональном стилях.

Решение. Данная задача является примером задачи вычисления значения по заданной функциональной зависимости, в данном случае, вычисления значения e^x по заданному аргументу x. Фактически, поскольку вычисления осуществляются приближенно, необходимо ввести дополнительный аргумент, определяющий точность вычислений. Будем считать, что задано также вещественное значение *eps*, причем суммирование ряда Тейлора будет прекращено, когда очередной член ряда станет по абсолютной величине меньше *eps*. Тогда задача сводится к вычислению некоторой функции *f(x, eps)*. Программа, написанная на языке *Java* в традиционном стиле, может выглядеть так, как показано в листинге У1.1.

Листинг У1.1. Вычисление числа *e* в традиционном стиле.

```
public class E {
  public static void main(String[] args) {
    double x, eps; // исходные данные
    double ex = 0; // результат вычислений, исходная сумма = 0
    double u = 1;  // промежуточное значение члена ряда
    int n = 1;     // номер очередного члена ряда Тейлора
    // чтение исходных данных
    try {
      x = Double.parseDouble(args[0]);
      eps = Double.parseDouble(args[1]);
    } catch (NumberFormatException e) {
      System.out.println(e.getMessage());
      return;
    } catch (IndexOutOfBoundsException e) {
      System.out.println(e.getMessage());
      return;
    }
    while (u >= eps) {
      // вычисление очередного члена ряда, суммирование
      ex += u;
      u = u * x / n;
      n++;
    }
    System.out.println(ex);     // вывод результата работы
  }
}
```

Для написания той же программы в функциональном стиле следует написать функцию, определяющую зависимость e^x от x и *eps*, явно. Такая зависимость может быть выражена с помощью рекуррентного соотношения, которое определяет способ вычисления частичных сумм ряда. Если уже имеется вычисленная частичная сумма ряда S_n и очередной член ряда U_n, то вычисления можно заканчивать, если значение U_n меньше

заданного *eps*. Если же это не так, то требуется добавить очередной член ряда к сумме, вычислить значение очередного члена ряда, и вызвать ту же функцию рекурсивно.

Если к тому же считать, что исходные значения могут быть получены с помощью функции чтения `read`, а результат должен быть передан функции вывода результата `write`, то набор функций, решающих задачу, может выглядеть так, как показано в листинге У1.2.

Листинг У1.2. Вычисление числа *e* в функциональном стиле.

```
public class E {
  /** Основная функция вычисления e в степени x.
   * @param x показатель степени,
   * @param eps точность вычислений.
   * @return e в степени x с заданной точностью.
   */
  static double ex (double x, double eps) {
    // вычисление происходит с помощью вызова
    // вспомогательной рекурсивной функции.
    return exRec(0, 1, 1);
  }

  /** Вспомогательная рекурсивная функция,
   * осуществляющая вычисления.
   * @param sum накопленная частичная сумма ряда,
   * @param u очередной член ряда,
   * @param n номер следующего члена ряда.
   * @return сумма ряда с заданной точностью.
   */
  static double exRec(double sum, double u, int n) {
    return u < eps ? sum : exRec(sum + u, u * x / n, n + 1);
  }

  public static void main(String[] args) {
    // ввод аргументов, вычисление и вывод результата
    write(ex(read, read));
  }
}
```

Задание 2. Написать программу для вычисления приближенного значения корня уравнения $\cos x = x$. Вычисления проводить методом бисекции (последовательного деления промежутка, содержащего корень функции, пополам). Предложить программы на языке *Java*, написанные в традиционном императивном и функциональном стилях.

Решение. Задача нахождения корня функции методом бисекции может быть решена в более общем виде, чем для решения одного конкретного уравнения. Необходимым условием для применимости метода служит наличие интервала, где непрерывная функция имеет единственный корень. В нашем случае непрерывная на всей вещественной оси функция

cos x – x = 0 имеет единственный корень на промежутке [0, π/2] (это можно увидеть, например, из графика функции). Программа, написанная на языке *Java* в традиционном стиле, может выглядеть так, как показано в листинге У1.3.

Листинг У1.3. Нахождение корня функции методом бисекции

```
public class Bisection {
  /** функция, корень которой находится в задаче.
   * @param x аргумент функции
   * @return результат функции
   */
  static double f(double x) {
    return Math.cos(x) - x;
  }

  public static void main(String[] args) {
    double eps;    // точность вычислений.
    double a = 0,  // концы интервала, содержащего корень.
           b = Math.PI / 2;
    double fa = f(a),
           fb = f(b);  // значения функции на концах.
    double m, fm;      // вспомогательные переменные.
    // чтение исходных данных (точности вычмслений)
    try {
      eps = Double.parseDouble(args[0]);
    } catch (NumberFormatException e) {
      System.out.println(e.getMessage());
      return;
    } catch (IndexOutOfBoundsException e) {
      System.out.println(e.getMessage());
      return;
    }
    // цикл последовательного деления промежутка,
    // содержащего корень функции, пополам.
    while (b - a > eps) {
      m = (a + b) / 2; fm = f(m);
      if (Math.signum(fa) == Math.signum(fm)) {
        // функция имеет значения разных знаков
        // на концах интервала [m,b]
        a = m; fa = fm;
      } else {
        // функция имеет значения разных знаков
        // на концах интервала [a,m]
        b = m; fb = fm;
      }
    }
    // вывод результата
    System.out.println(a);
  }
```

```
}
```

Если писать программу в чисто функциональном стиле, то функция для решения поставленной задачи должна иметь в качестве аргумента точность вычислений, а в качестве результата – корень функции. Метод решения состоит в том, что на основе имеющихся данных об интервале, содержащем корень функции, строятся уточненные данные о таком интервале методом деления интервала пополам и выбора из двух половин той, которая содержит корень (функция на концах такого интервала принимает значения разных знаков). Поэтому основная функция, реализующая метод бисекции, может иметь в качестве аргументов помимо точности вычислений значения концов интервала и знаки значений функции на концах интервала. Тогда функциональное решение задачи может иметь такой вид, как показано в листинге У1.4.

Листинг У1.4. Поиск корня функции методом бисекции в функциональном стиле.

```
// Интерфейс для представления функциональных значений
public interface Func {
  double func(double x);
}
// Основной класс для решения задачи
public class Bisection {
  // опишем вещественную функцию, корень которой мы ищем.
  Final static Func f = new Func() {
    public double func(double x) {
      return Math.cos(x) - x;
    }
  };
  /** Основная функция вычисления корня функции
   * на заданном промежутке.
   * @param f функция, корень которой ищем.
   * @param a левый конец промежутка.
   * @param b правый конец промежутка.
   * @param fa знак функции на левом конце промежутка.
   * @param fb знак функции на правом конце промежутка.
   * @param eps точность вычислений.
   * @return e в степени x с заданной точностью
   */
  static double root(Func f, double a, double b,
                     boolean fa, boolean fb, double eps) {
    final double  m  = (a + b) / 2;
    final boolean fm = f.func(m) > 0;
    return b - a < eps ?    // требуемая точность достигнута.
           a : fa == fm ?   // осуществляем рекурсивный вызов.
           root(f, m, b, fm, fb, eps) :
           root(f, a, m, fa, fm, eps);
  }
  // Тестовая функция
```

```
  public static void main(String[] args) {
    // ввод аргументов, вычисление и вывод результата
    System.out.println(
        root(f, 0, Math.PI/2, f.func(0) > 0,
             f.func(Math.PI/2) > 0, read));
  }
}
```

По данному решению можно сделать одно замечание. В основной функции `root` одним из аргументов является функция, представленная в нашем решении интерфейсом `Func`. Это не является необходимым, поскольку задача решается для одной конкретной функции, однако добавляет общности.

Задание 3. Написать функцию для нахождения заданного значения в упорядоченном массиве целых чисел методом двоичного поиска. Предложить программы на языке *Java*, написанные в традиционном императивном и функциональном стилях.

Решение. Традиционное решение предполагает использование индексов в массиве для поиска заданного элемента, при этом строится цикл, в процессе работы которого область поиска постоянно сужается до тех пор, пока не будет найдено искомое значение, либо пока область поиска не станет пустой. Одно из возможных решений представлено в листинге У1.5.

Листинг У1.5. Двоичный поиск в массиве целых в традиционном стиле

```
/**
 * Поиск элемента в упорядоченном массиве
 * методом двоичного поиска.
 * @param array исходный упорядоченный массив.
 * @param elem искомый элемент.
 * @return индекс найденного элемента или -1,
 *         если элемент не найден.
 */
public static int search(int[] array, int elem) {
  int low = 0,                  // нижняя граница поиска
      high = array.length - 1; // верхняя граница поиска
  while (low < high) {
    // Inv: искомый элемент в области array[low..high],
    // если только он вообще есть в массиве.
    int middle = (low + high)/2;  // low <= middle < high
    if (array[middle] < elem) {
      low = middle + 1;
    } else {
      high = middle;
    }
  }
  return array[low] == elem ? low : -1;
}
```

Функциональное решение вместо цикла использует рекурсивную функцию, которая сужает область поиска, если искомый элемент еще не найден, и обращается к себе с новой областью поиска. Вариант такого решения представлен в листинге У1.6.

Листинг У1.6. Двоичный поиск в массиве целых в функциональном стиле

```
/**
 * Поиск элемента в упорядоченном массиве
 * методом двоичного поиска.
 * @param array исходный упорядоченный массив.
 * @param elem искомый элемент.
 * @return индекс найденного элемента или -1,
 *         если элемент не найден.
 */
public static int search(int[] array, int elem) {
  return searchRec(array, elem, 0, array.length - 1);
}

/**
 * Вспомогательная рекурсивная функция поиска элемента
 * в заданном фрагменте упорядоченного массива
 * методом двоичного поиска.
 * @param array исходный упорядоченный массив.
 * @param elem искомый элемент.
 * @param low нижний индекс области поиска.
 * @param high верхний индекс области поиска.
 * @return индекс найденного элемента или -1,
 * если элемент не найден.
 */
public static int searchRec(int[] array, int elem,
                            int low, int high) {
  final int middle = (low + high)/2;  // low <= middle < high
  final int am = array[middle];
  return am == elem ?   // элемент найден.
           middle :
         low == high ?  // элемента нет в заданной области.
           -1 :
         am < elem ?    // элемент в верхней части массива.
           searchRec(array, elem, middle+1, high) :
                        // элемент в нижней части массива.
           searchRec(array, elem, low, middle);
}
```

Задания для самостоятельной работы

Во всех нижеследующих заданиях требуется написать программу для решения поставленной задачи с использованием двух различных стилей программирования – традиционного императивного стиля и

функционального стиля. Для оформления программ можно использовать любой из традиционных языков программирования (Паскаль, C, C++, Java и т.п.).

Задание 1. Написать программу для нахождения площади заданного выпуклого многоугольника, если заданы координаты всех его вершин. Можно считать, что вершины упорядочены таким образом, что каждая следующая вершина связана с предыдущей ребром многоугольника.

Задание 2. Написать программу для вычисления n-го члена числовой последовательности, заданной следующим рекуррентным соотношением: $a_0 = a_1 = 1$, $a_{n+2} = 3a_n - 2a_{n-1} + 1$ при $n>1$.

Задание 3. Написать программу для вычисления приближенного значения функции $\cos x$ при заданном значении x. Для вычислений использовать разложение косинуса в ряд Тейлора в окрестности нуля. Использовать периодичность тригонометрических функций для приведения аргумента к окрестности нуля для повышения точности вычислений.

Задание 4. Написать программу для вычисления максимального значения в заданном числовом массиве.

Задание 5. Написать программу для вычисления суммы элементов заданного числового массива.

Задание 6. Написать программу для нахождения минимального из чисел, являющихся максимальными в каждой из строк заданной прямоугольной матрицы.

Задание 7. Написать программу для нахождения длины максимальной возрастающей подпоследовательности элементов в заданном числовом массиве. Рассмотреть два варианта этой задачи: случай, когда искомая подпоследовательность состоит из элементов, расположенных подряд, и случай, когда элементы подпоследовательности могут занимать произвольные места в массиве.

Задание 8. Описать функцию, которая проверяет, можно ли на квадратной шахматной доске размером $n \times n$ клеток расставить n ферзей таким образом, чтобы никакие два ферзя не находились ни на одной вертикали, ни на одной горизонтали и ни на одной диагонали. Разработать два варианта функции, в первом из которых расстановка ферзей на шахматной доске представлена числовым или логическим массивом, а во втором – знание о расстановке представлено функцией, которая по заданным координатам поля выдает информацию о том, свободно ли это поле для установки очередного ферзя.

1.2. Элементы языка *Haskell*

Первая версия языка *Haskell* появилась в 1990-м году. К концу 1990-х годов он был довольно существенно переработан и упорядочен, а в 1998-м году появилась современная версия этого языка - *Haskell'98*. Язык

опирается на богатую историю создания функциональных языков программирования, начатую первым таким языком – *Лиспом*, описанным в 1960-м году МакКарти. Сильное влияние на язык оказали следующие работы в области функционального программирования: система комбинаторного программирования *FP* Бэкуса, появившаяся в конце 1970-х годов, язык *ML* ("Meta-language", созданный в Эдинбургском университете примерно в то же время) и созданный в 1985-1986 под сильным влиянием *ML* язык *Miranda*. Названными языками и системами мир функционального программирования далеко не исчерпывается. В частности, названные языки и системы носят, скорее, «академический» или учебный характер, однако есть и, по крайней мере, один коммерческий язык функционального программирования – *Euler*.

Мы изучим элементы языка *Haskell* для того, чтобы иметь в своем распоряжении инструмент функционального программирования и чтобы понять, какие средства и возможности есть в функциональном программировании. Конечно, для изучения языка необходимо самому писать программы на этом языке, отлаживать и исполнять их. В приложении 1 приведены рекомендации по установке и использованию системы программирования на языке Haskell – WinHugs. Это некоммерческая система программирования, информацию о которой и о многих других системах программирования на языке Haskell можно найти на сайте http://www.haskell.org/.

В процессе дальнейшего изложения мы будем приводить примеры и задачи для самостоятельного решения, которые можно программировать и отлаживать с использованием этой или других систем функционального программирования на языке *Haskell*.

Haskell - строго типизированный язык. Это означает, что любая конструкция в языке, задающая некоторый объект, имеет определенный тип, который можно «вычислить» еще до начала выполнения программы, то есть *статически*. Таким же свойством обладают большинство современных языков программирования, такие как *Паскаль*, *Java*, *Си++* и многие другие. Труднее привести пример ***не*** строго типизированного языка (если, конечно, не считать некоторых отступлений от строгой типизации в упомянутых выше языках, сделанных для удобства практического программирования, таких, например, как тип `Pointer` в языке *Паскаль*). Пожалуй, самым известным из языков, изначально не имеющих строгой типизации, является первый язык функционального программирования *Лисп*.

Основу системы типов языка *Haskell* составляют элементарные встроенные типы данных: целые, представленные двумя подтипами с идентификаторами `Integer` и `Int`, вещественные, также с двумя подтипами `Float` и `Double`, логические, имеющие идентификатор типа `Bool` и символьные (`Char`). Заметим сразу же, что все идентификаторы в

Haskell чувствительны к регистру букв, так что `integer`, `Integer` и `INTEGER` – это три разных идентификатора. Регистр первой буквы идентификатора определяет также, к какому из двух «миров» относится идентификатор: если идентификатор начинается с заглавной буквы, то он обозначает встроенный или определенный программистом *тип* или *класс*. Идентификаторы объектов – значений простых и сложных типов, в том числе функций – начинаются со строчной буквы или символа подчеркивания. Идентификаторы, как и в других языках программирования, строятся из букв, подчеркиваний и цифр, однако в *Haskell* допустимо использовать для построения идентификаторов также символ `"'"` («апостроф»). Ни апостроф, ни цифра не могут быть первым символом идентификатора. Заметим, что апостроф традиционно используется для построения имен функций, вспомогательных по отношению к функции, заданной тем же именем, только без апострофов. Например, если мы определяем функцию с именем `factorial` и в качестве вспомогательных для нее хотим определить еще две функции, не имеющие самостоятельного значения, то имена этих функций по традиции, скорее всего, будут `factorial'` и `factorial''`.

Четыре базовых типа - это, конечно, типы, определяющие наборы хорошо известных значений целого, вещественного, логического и символьного типов. Отметим несколько особенностей базовых типов *Haskell*, непривычных по другим языкам программирования.

Тип `Integer` определяет потенциально бесконечный набор целых чисел произвольной длины. В операциях над целыми типа `Integer` никогда не происходит «переполнения» (конечно, в пределах выделенной для работы программы памяти). Для повышения эффективности работы программ можно использовать и вполне традиционные целые ограниченной длины. Для таких «ограниченных» целых используется идентификатор типа `Int`. Последовательность цифр, возможно, предваренная знаком '–', представляет в программе целое число (как говорят, целые числа имеют литеральные обозначения в виде последовательности цифр).

Вещественные числа типов `Float` и `Double` определяются вполне традиционно и имеют также традиционные литеральные обозначения, такие как `3.14`, `-2.71828` или `0.12e-10`.

Символьный тип также имеет литеральные обозначения для своих значений, и они тоже вполне традиционны. Так, обозначение `'a'` представляет символ `a`.

Логический тип представлен двумя значениями – истина и ложь. Литеральных обозначений для логических значений в языке нет, однако, имеются два стандартных идентификатора – `True` и `False`, обозначающие истинное и ложное значения соответственно.

В программе можно вводить свои собственные идентификаторы для имеющихся значений. Такая возможность совершенно эквивалентна средствам описания констант в различных языках программирования. Например, идентификатор `school` можно ввести для обозначения константы `239` с помощью следующего фрагмента программы:

```
school = 239
```

Если мы хотим явно указать тип для нового идентификатора объекта, то это можно сделать с помощью конструкции определения типа. Например, если мы хотим указать, что введенный идентификатор `school` должен представлять значение типа `Integer`, то перед определением значения следует написать

```
school :: Integer
```

Такое уточнение типа не обязательно, потому что система программирования на языке *Haskell* обязана сама устанавливать (*выводить*) типы всех встречающихся объектов и выражений, но иногда это может все же оказаться существенным. Так, например, литерал `239` в предыдущем примере может обозначать как целое типа `Int`, так и целое типа `Integer` в зависимости от контекста. Явное указание типа `Integer` устраняет неоднозначность.

Значения произвольных типов можно объединять в *кортежи*. Аналогом этого понятия в языке *Паскаль* будет определение типа записи. Объединяемые в кортеж значения просто перечисляются в скобках через запятую. Например, обозначение `(2, 'a')` представляет кортеж из двух значений - целого числа `2` и символа `'a'`. Заметим, что типом этого кортежа будет `(Int, Char)`, так что можно написать следующий фрагмент программы, в котором вводится новое обозначение для этого кортежа и явно указывается его тип:

```
pair :: (Int, Char)
pair = (2, 'a')
```

Разумеется, кортежи сами могут входить в состав других кортежей. Например, следующий фрагмент программы определяет идентификатор для кортежа, в состав которого входит другой кортеж.

```
myValue :: (Int, (Char, Char), Bool)
myValue = (366, ('A', 'K'), True)
```

Над значениями можно выполнять определенные в языке операции и применять стандартные функции. Полный набор стандартных функций можно посмотреть, например, в официальном пересмотренном сообщении о языке *Haskell'98*, или в других подробных документах по языку. Мы будем постепенно вводить новые операции и стандартные функции по мере необходимости, а сейчас пока отметим, что в языке имеется достаточно обычный набор арифметических и логических операций, а также операций сравнения и функций преобразования значений из одних

типов в другие. Запись выражений также достаточно традиционна, так что не вызывает никакого удивления, что, скажем, выражение `2+3` записано правильно и при вычислении выдает значение `5`, а выражение `3<(2+2)` выдает значение *истина*. Наиболее необычными являются две особенности записи выражений.

Во-первых, при вызове функций аргументы отделяются от идентификатора функции не привычными скобками, а пробелом. Если аргументов несколько, то сами аргументы тоже отделяются друг от друга пробелами. Так, например, функцию вычисления синуса, определенную для вещественных аргументов, можно вызвать с помощью конструкции

```
sin 0.5
```

а функцию определения максимального из двух любых упорядоченных значений можно вызвать (в предположении, что идентификатор `x` обозначает некоторое целое значение), скажем, так:

```
max 3 (x+1)
```

Вторая существенная особенность состоит в том, что операторы, задающие арифметические и другие операции, вообще говоря, синтаксически неотличимы от обычных двуместных функций. Я имею в виду, что вызов операции, подобной сложению двух целых, можно записывать не только в виде бинарной операции, расположенной между аргументами, но и так, как это принято для обычных двуместных функций. Для превращения знака бинарной операции в двуместную функцию надо лишь заключить знак операции в скобки, так что запись

```
(+) x y
```

полностью эквивалентна записи

```
x + y
```

Обратное тоже верно. Для того, чтобы превратить обычный идентификатор функции двух аргументов в бинарный оператор достаточно заключить идентификатор функции в обратные апострофы, так что приведенное выше обращение к функции вычисления максимального из двух упорядоченных значений можно было бы записать в виде применения бинарной операции:

```
3 `max` (x+1)
```

Конечно, для того, чтобы написать хоть сколько-то полезную программу, необходимо помимо новых идентификаторов для объектов и написания выражений уметь самому определять новые функции. Каждая функция в *Haskell* – это тоже некоторое значение, для которого определен тип. В типе функции указывают типы ее аргументов и тип значения функции, при этом типы аргументов и значения функции отделяются друг от друга символом `'->'`, так что, например, тип функции одного вещественного аргумента с вещественным результатом (такой, как, например, функция вычисления синуса) можно записать в виде

```
Double -> Double
```

а тип функции с двумя целыми аргументами и одним логическим результатом (такой, как, например, операция сравнения двух целых значений по величине) можно записать в виде

```
Int -> Int -> Bool
```

Саму функцию можно определить с помощью «уравнения», в котором определяется, как функция должна себя вести, если ей задать значение аргумента. Давайте, например, определим функцию удвоения, которая удваивает значение своего вещественного аргумента и выдает получившееся значение. Для этого зададим идентификатор функции `twice`, зададим ее тип и напишем уравнение, в котором покажем, что вызов этой функции с заданным значением аргумента эквивалентен выражению, в котором это значение умножается на два:

```
twice :: Double -> Double
twice x = 2 * x
```

В уравнении, определяющем поведение функции `twice`, можно выделить левую часть, задающую форму вызова функции, и правую часть, в которой записывается выражение, заменяющее вызов функции в тот момент, когда будет задано значение ее аргумента. Если функция `twice` определена, то вычисление выражения `twice 5.5` можно представить себе следующим образом. Сначала происходит сопоставление фактического значения аргумента `5.5` с формальным аргументом функции, заданном в уравнении, определяющем поведение функции. Потом вызов функции заменяется правой частью уравнения, в которой вместо формального аргумента используется сопоставленное с ним значение фактического аргумента. Таким образом, после сопоставления и замены вместо выражения `twice 5.5` получаем выражение `2 * 5.5`, которое после вычисления (применения операции умножения) дает значение `11`. Процесс преобразования (вычисления) выражения можно записать следующим образом:

```
twice 5.5  ⇒  2 * 5.5  ⇒  11
```

Преобразование выражений, подобное только что приведенному, называют еще редукциями. Таким образом, можно сказать, что вычисление выражений в *Haskell* (исполнение программы) осуществляется с помощью последовательных редукций исходного выражения.

Определение функции может быть рекурсивным, то есть в правой части уравнения может быть вызов определяемой функции. В этом случае в процессе преобразования выражения, содержащего вызов рекурсивной функции, может получаться выражение, также содержащее вызов той же самой функции. Для того, чтобы процесс вычисления мог закончиться, необходимо использовать условные выражения, которые приводят к

выбору одной из двух альтернатив при вычислении сложных выражений. Условное выражение имеет вид

```
if <условие> then <выражение-"то"> else <выражение-"иначе">
```

При вычислении условного выражения прежде всего вычисляется условие. Тип вычисленного выражения-условия должен быть логическим (`Bool`). Если вычисленное значение оказывается истинным, то вместо всего условного выражения подставляется *выражение-"то"*, а *выражение-"иначе"* отбрасывается. Если же наоборот, условие оказалось ложным, то отбрасывается *выражение-"то"*, а вместо всего условного выражения остается только часть, определенная *выражением-"иначе"*.

Зададим определение простой рекурсивной функции, предназначенной для вычисления факториала заданного целого числа. Простое и естественное определение этой функции может выглядеть следующим образом:

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

Проследим за тем, как происходит вычисление выражения `factorial 3`, записывая последовательно все этапы преобразования выражения в соответствии с определением функции `factorial` (листинг 1.6).

Листинг 1.6. Процесс преобразования выражения при вычислении факториала числа

```
factorial 3                                          ⇒
if 3 == 0 then 1 else 3 * factorial 2                ⇒
if False then 1 else 3 * factorial 2                 ⇒
3 * factorial 2                                      ⇒
3 * if 2 == 0 then 1 else 2 * factorial 1            ⇒
3 * 2 * factorial 1                                  ⇒
3 * 2 * if 1 == 0 then 1 else 1 * factorial 0        ⇒
3 * 2 * 1 * factorial 0                              ⇒
3 * 2 * 1 * if 0 == 0 then 1 else 0 * factorial (-1) ⇒
3 * 2 * 1 * 1                                        ⇒
6
```

Вместо использования условного выражения можно записать несколько уравнений, определяющих поведение функции при различных значениях аргумента. Так, например, можно переопределить функцию вычисления факториала, задав для нее два уравнения, обусловленных выражениями-«сторожами», определяющими, какое из двух уравнений на самом деле следует применять.

```
factorial :: Integer -> Integer
factorial n | n == 0 = 1
factorial n | n > 0  = n * factorial (n-1)
```

или короче

```
factorial :: Integer -> Integer
factorial n | n == 0 = 1
            | n > 0  = n * factorial (n-1)
```

Обратите внимание, что в новом варианте функции сторожа не покрывают всех возможных значений аргумента. Если написать бессмысленный вызов функции `factorial -3`, то при нашем первоначальном определении вычисление выражения в программе уйдет в бесконечный цикл; преобразование выражения никогда не сможет нормально закончиться. Во втором случае система программирования сразу же выдаст сообщение об ошибке, поскольку ей не удастся найти уравнение со сторожем, который при вычислении выдаст истинное значение.

Еще один способ определения той же функции состоит в том, что в уравнениях, определяющих функцию, можно сразу же задать случай, когда аргументом вызова будет нулевое значение. Определение функции теперь может выглядеть так:

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

В этом определении тоже происходит выбор нужного уравнения из двух возможных вариантов. Порядок задания уравнения здесь важен. Сначала рассматривается первое уравнение, в котором фактический аргумент сопоставляется с нулевой константой. Сопоставление возможно только в том случае, когда сам этот аргумент равен нулю. Если же аргумент больше нуля, то будет выбрано второе уравнение, в котором сопоставление всегда произойдет успешно, поскольку с идентификатором `n` может сопоставиться любое значение. После замены выражения вызова на правую часть будет произведено новое сопоставление и выбор уравнения и т.д. Вычисление факториала будет, конечно, происходить по тому же самому алгоритму. Заметим, что в этом последнем способе задания функции опять возможным становится задание отрицательного значения аргумента, при этом программа будет выполняться бесконечно, а никакого сообщения об ошибке выдано не будет.

Напишем еще несколько функций, выполняющих те или иные арифметические действия, просто для того, чтобы освоиться с манерой написания функций в языке *Haskell*. Определим функцию, которая по заданным натуральным числам определяет их наибольший общий делитель согласно алгоритму Евклида. Назовем эту функцию `gcd` (*greatest common divider*). Идея Евклида состоит в том, что если ни одно из чисел не равно нулю, то наибольший общий делитель таких двух чисел равен наибольшему общему делителю наименьшего из них и остатка от деления большего на меньшее. Поскольку остаток от деления одного числа на

другое можно вычислить с помощью стандартной бинарной операции `mod`, то функцию `gcd` можно определить следующим образом (листинг 1.7):

Листинг 1.7. Наибольший общий делитель двух чисел

```
gcd :: Integer -> Integer -> Integer
-- в первом уравнении обеспечиваем,
-- чтобы первый аргумент был больше второго.
-- функция определена только для неотрицательных аргументов
gcd m n | m < 0 = error "gcd: Negative argument"
        | n < 0 = error "gcd: Negative argument"
-- собственно алгоритм Евклида:
gcd m 0 = m
gcd m n = gcd n (m `mod` n)
```

В этом примере использованы различные виды уравнений, определяющих одну и ту же функцию. Стандартная функция `error` выводит в стандартный выходной поток сообщение, являющееся ее аргументом и возвращает в качестве результата «неопределенное значение». Если такое неопределенное значение действительно будет получено в результате работы программы, то программа будет завершена аварийно. Как и в только что показанном примере функция `error` обычно используется для вывода диагностических сообщений и аварийного завершения работы программы в случае неверно заданных аргументов функции.

ЗАМЕЧАНИЕ

> Функция `error` является примером функции с побочным эффектом, то есть она не есть «чистая» функция. На самом деле, язык *Haskell* содержит помимо чисто функциональных средств также и набор средств, выходящих за рамки собственно функционального программирования. Функция `error` – это один из примеров применения нефункциональных средств в функциональном языке программирования. В данной книге мы не будем рассматривать и применять такие средства языка, функция `error` – это единственное исключение.

В следующем примере (листинг 1.8) определим функцию для проверки того, является ли заданное натуральное число простым. В этой функции аргумент последовательно проверяется на его делимость на все возрастающие натуральные числа до тех пор, пока не оказывается, что либо делитель найден, либо этот делитель уже превосходит квадратный корень из аргумента. В первом случае исходное число будет составным, а во втором – простым.

Листинг 1.8. Проверка простоты натурального числа

```
prime :: Integer -> Bool
-- вспомогательная функция prime' проверяет простоту числа
```

```
-- при условии, что уже проверена его делимость на все числа,
-- меньшие заданного.
prime' :: Integer -> Integer -> Bool
-- начинаем проверку с числа 2
prime p | p <= 0    = error "prime: Non-positive argument"
        | otherwise = prime' 2 p
-- рассматриваем три случая, заданные тремя уравнениями:
-- не найдено делителей числа, не превышающих его корня;
-- найден делитель – число не простое;
-- делитель пока не найден, продолжаем проверку.
prime' d p | d * d > p       = True
           | p `mod` d == 0  = False
           | otherwise       = prime' (d+1) p
```

В приведенной программе кроме основной функции `prime` описана вспомогательная функция `prime'`. Это сделано потому, что рекурсию по основному аргументу функции – проверяемому числу – организовать очень трудно. В то же время при введении дополнительного аргумента, представляющего возможный делитель числа, рекурсия организуется очень легко. Еще одна особенность приведенной программы – использование идентификатора `otherwise` в качестве условия при написании уравнения для функции `prime'`. На самом деле это тождественно истинное условие, так что можно было бы вместо него просто написать `True`. Специальное слово `otherwise` используется просто для большей наглядности (по крайней мере, для программистов, владеющих английским языком; *otherwise* означает *иначе* или *в противном случае*).

Иногда не удается так же быстро и естественно найти приемлемое по эффективности работы решение. Рассмотрим, например, следующую задачу: по заданному номеру n найти n-ое число в последовательности чисел Фибоначчи. Напомним, что первые два числа в последовательности чисел Фибоначчи равны единице, а каждое из следующих чисел равно сумме двух предыдущих. Другими словами, если обозначить n-ое число Фибоначчи через f_n, то можно выписать следующие формулы:

$$f_1 = f_2 = 1$$
$$f_{n+2} = f_n + f_{n-1}$$

Это простое рекурсивное определение можно положить в основу функции, которая будет решать поставленную задачу, то есть находить число Фибоначчи по его номеру в последовательности (листинг 1.9):

Листинг 1.9. Нахождение чисел Фибоначчи

```
fib :: Integer -> Integer
fib 1 = 1
fib 2 = 1
fib n = fib (n-1) + fib (n-2)
```

В этом решении в главном уравнении, определяющем поведение функции, производятся два рекурсивных обращения к той же самой функции. Это приводит к тому, что число обращений к функции резко возрастает при увеличении значения аргумента. Если 30-е число Фибоначчи еще можно получить сравнительно быстро (современный персональный компьютер справляется с задачей за несколько секунд), то уже нахождение 40-го числа требует нескольких часов работы! С увеличением аргумента время, требующееся для решения задачи, растет экспоненциально. Конечно, это совершенно непроизводительные затраты времени. Причина столь плохого поведения программы состоит в том, что одни и те же значения чисел Фибоначчи вычисляются в ней многократно. В то же время очевидно, что можно найти решение, которое будет выдавать результат за линейное относительно величины аргумента время.

Конечно, для получения эффективного решения надо просто добавить в функцию дополнительную информацию, чтобы ей не приходилось при каждом вызове независимо находить уже найденные ранее значения чисел Фибоначчи. Например, можно легко определить функцию, которая вычисляет *n*-ое число Фибоначчи, зная, что уже вычислены первые *k* чисел Фибоначчи и имея значения хотя бы двух последних вычисленных чисел. Можно определить такую вспомогательную функцию, которая будет иметь четыре аргумента: `n` – номер числа Фибоначчи, которое надо вычислить, `k` – номер последнего вычисленного числа Фибоначчи, `fk` – значение *k*-го числа Фибоначчи и `fk1` – значение предыдущего числа Фибоначчи. Тогда с помощью этой вспомогательной функции легко определить и требуемую нами изначально функцию. Новый вариант программы может выглядеть так, как показано в листинге 1.10.

Листинг 1.10. Эффективное вычисление чисел Фибоначчи

```
-- fib - основная функция вычисления n-го числа Фибоначчи.
fib :: Integer -> Integer
-- fib' - вспомогательная функция.
fib' :: Integer -> Integer -> Integer -> Integer -> Integer
fib' n k fk fk1 | k == n  = fk
                | k < n   = fib' n (k+1) (fk+fk1) fk
fib 1 = 1
-- первоначально известны первые два числа Фибоначчи:
fib n = fib' n 2 1 1
```

Совершенно неудивительно, что описанный вариант программы мгновенно вычисляет не только 40-е, но и 100-е и даже 1000-е число Фибоначчи. Правда, последний результат содержит 209 десятичных цифр.

Дополнительные аргументы вспомогательной функции `fib'` играют в программе ту же роль, что и локальные переменные функций в традиционных языках программирования – они сохраняют информацию о

вычисленных значениях при последовательном исполнении программы. Так, например, аналогичная функция на языке *Java* могла бы быть записана следующим образом:

```
static int fib(int n) {
  int fk = 1, fk1 = 1;
  for (int k = 2; k <= n; ++k) {
    fk = fk + fk1;
    fk1 = fk – fk1;
  }
  fib = fk;
}
```

Эта функция совершенно аналогична приведенной выше программе на языке *Haskell*.

Обычно аргументы рекурсивных функций уменьшаются с течением времени (то есть, при последующих вызовах). Это мы могли уже наблюдать и в некоторых предыдущих примерах, таких как вычисление факториала числа или наибольшего общего делителя двух чисел. Однако, в предыдущем примере аргументы вспомогательной функции `fib'` наоборот, увеличивали значение от вызова к вызову, накапливая результат, пока, наконец, в последнем вызове не выдавали этот накопленный результат. Такие аргументы играют роль обычных переменных в традиционных языках программирования. Часто их называют накапливающими параметрами функции. Мы уже встречали аргумент с похожим поведением в задаче определения простоты заданного натурального числа, где дополнительный аргумент `d` вспомогательной функции `prime'` играл роль переменной, последовательно увеличивающей значение от 2 до корня квадратного из основного аргумента.

Приведем еще один пример функции с накапливающими параметрами. Пусть требуется вычислить сумму числового ряда для получения приближенного значения трансцендентного числа *e*. Аргументом основной функции будет маленькое положительное число, задающее величину последнего суммируемого члена ряда. Очевидно, что здесь требуется иметь несколько накапливающих параметров для хранения значения очередного члена ряда, его номера, накопленной суммы. Окончательный вариант программы может выглядеть как на листинге 1.11.

Листинг 1.11. Приближенное вычисление числа *e*

```
-- Основная функция.
-- Аргумент <eps> - точность вычислений.
-- Результат - вычисленное значение числа e
e :: Double -> Double

-- Вспомогательная функция с дополнительными аргументами
-- Аргументы: eps - точность вычислений,
```

```
--          n - номер последнего вычисленного члена суммы.
--          mem - значение последнего вычисленного члена суммы.
--          sum - накопленное значение суммы.
e' :: Double -> Double -> Double -> Double -> Double

e eps = e' eps 1 1 0
e' eps n mem sum | mem < eps = sum
                 | otherwise = e' eps (n+1) (mem/n) (sum+mem)
```

Конечно, подобным же образом можно решить многие хорошо известные вам задачи, причем решения будут не сильно отличаться от традиционных. Но не в этом сила и привлекательность функционального подхода к программированию. В дальнейшем мы будем писать много программ, совершенно непохожих на традиционные, и алгоритмы их решения также будут выглядеть совершенно по-другому (хотя, идея того или иного алгоритма часто будет вполне привычной).

В традиционных языках программирования для построения объектов, более сложных, чем элементарные числа или логические значения, используют различные встроенные механизмы для построения сложных объектов из более простых. Например, в языке *Паскаль* для этого применяют записи, массивы и списковые структуры, а в языке *Java* основным механизмом построения сложных значений являются классы. Аналогом записи в языке *Haskell* является кортеж, структуру которого мы уже коротко описывали выше. Правда, в языке *Haskell* поля в кортежах чаще всего не имеют имен и различаются лишь по позициям в кортеже, однако на самом деле задавать имена для отдельных полей кортежа и потом использовать эти имена для анализа и построения кортежей можно и в языке *Haskell* (эту возможность языка мы не будем рассматривать; интересующихся отошлем к сообщению о языке *Haskell'98*). Способов построения массивов в языке *Haskell* нет. Причина этого состоит в том, что для привычной работы с массивом требуется, чтобы отдельные компоненты этого массива вели себя как отдельные переменные, но это плохо соответствует функциональному стилю. Зато в языке *Haskell* имеется один способ, позволяющий строить структуры данных произвольной степени сложности из более простых объектов – это списки.

Списки в *Haskell* одновременно напоминают и списковые структуры традиционных языков программирования, и обычные массивы. Список – это последовательность объектов одного и того же типа, состоящая из произвольного числа объектов (возможно, ни одного – соответствующий список называется пустым). Тип списка определяется типом его компонент, и в языке *Haskell* обозначается `[T]`, где `T` – тип компонентов списка. Таким образом, `[Integer]` будет обозначать тип списка, содержащего в качестве компонентов целые числа, а `[([Char], Double)]` – это тип списка, компонентами которого являются кортежи

из двух полей, причем первым полем в каждом кортеже будет список символов, а вторым полем – вещественное число.

Сами объекты-списки в простейшем случае задаются перечислением своих компонентов в квадратных скобках через запятую, так что, например, `[1, 3, 4, 8]` – это четырехкомпонентный список целых чисел (типом этого списка будет `[Int]`), а `[(['p', 'i'], 3.14), (['e'], 2.72)]` – список вышеупомянутого типа `[([Char], Double)]`. Заметим, что запись, в которой компонентами списка будут объекты разных типов, синтаксически неверна, так что, скажем, `[3, 4, '*']` – не может служить изображением списка, поскольку из трех указанных компонент две первые - это целые числа, а третья – символ.

Отметим два полезных сокращения, использующихся для записи списков. Во-первых, список, состоящий из символов, можно записать в виде строки, заключив составляющие его символы в двойные кавычки. Так, например, строка `"Haskell"` обозначает в программе список из семи символьных значений, и такая запись полностью эквивалентна расширенной записи: `['H', 'a', 's', 'k', 'e', 'l', 'l']`. Для списков символов в языке также есть и специальный идентификатор типа – `String`, так что его можно использовать в любом месте программы вместо `[Char]`. Таким образом, один из примеров предыдущего абзаца можно записать так: список `[("pi", 3.14), ("e", 2.72)]` является примером изображения списка типа `[(String, Double)]`.

Второе удобное сокращение или даже, скорее, средство построения списков – это задание списка целых чисел, образующих арифметическую прогрессию. Если шаг прогрессии равен единице, то можно указать только первый и последний элементы списка, разделив их символом из двух точек. Так `[1..10]` обозначает список из десяти последовательных натуральных чисел от `1` до `10`. Если шаг прогрессии не равен единице, то указывают два первых элемента списка и последний элемент. Таким образом, изображение списка `[3, 5..11]` задает список из пяти целых чисел `3`, `5`, `7`, `9` и `11`. Отметим, что эта запись может использоваться и для задания списков с числом элементов, неизвестным в момент написания программы. Например, изображение `[1..n]` может использоваться для создания списка первых натуральных чисел от единицы до текущего значения `n`, которое в разных ситуациях при исполнении программы может быть различным. Комбинировать в изображении списка прогрессию с отдельными элементами, не входящими в прогрессию, нельзя. Так, например, запись `[3, 5..13, 14]` будет некорректной.

Списки могут создаваться в программе путем последовательного присоединения элементов к началу списка с помощью специальной операции – конструктора списков, которая изображается в виде двоеточия. Например, если `x` – список из целых чисел, равный `[2,3,4]`, то

конструкция `1:x` создает новый список, элементами которого будут числа `1`, `2`, `3`, `4`. На самом деле изображение списка в виде последовательности элементов всегда можно записать с помощью конструктора списков. Например, список `[2, 3, 4]` может быть записан в виде `2:(3:(4:[]))`. В этом примере круглые скобки необязательны, так как построение списка происходит «справа налево», то есть от последнего элемента к первому. В этой записи использована также константа `[]`, изображающая пустой список.

Конструктор списков – это не совсем обычная операция. Она отличается от, скажем, операции арифметического сложения целых тем, что не «вычисляет» никакого нового значения, а просто собирает сложный объект из более простых. Эти более простые объекты остаются в составе списка неизменными, их можно извлечь оттуда и обрабатывать отдельно, в то время как после сложения числа 3 с числом 5 получается новый объект – число 8, в состав которого никоим образом не входят ни тройка, ни пятерка. Такие операции, которые лишь соединяют объекты в новый составной объект, вообще в функциональном программировании называются конструкторами, и конструктор списков – это лишь частный случай более общего понятия конструктора. В дальнейшем мы увидим, что изображения константных значений (литералы) также можно рассматривать как конструкторы, создающие элементарные объекты. Таким образом, список, заданный перечислением своих элементов, может быть также задан с помощью применения двух конструкторов – двоеточия и пустого списка.

Для того, чтобы писать программы обработки списков, надо научиться извлекать отдельные элементы списка. Это можно делать либо с помощью встроенных операций, позволяющих извлечь отдельные составные части списка, либо с помощью аппарата *сопоставления с образцом*. Давайте сначала рассмотрим первый способ.

Имеется много встроенных в язык операций обработки списков. Вот важнейшие из них:

`head, last` – функции, которые по заданному аргументу-списку выдают его первый и последний элементы соответственно. Операции применимы только к непустому списку.

`tail` – функция, выдающая остаток списка, полученный отбрасыванием первого элемента. Аналогично, функция `init` выдает список без последнего элемента. Обе операции также неприменимы к пустому списку.

`!!` – операция, которая по списку и заданному номеру элемента выдает соответствующий элемент списка. Список должен быть непустым, а номер должен лежать в пределах от нуля (первый элемент списка имеет номер ноль) до количества элементов списка без единицы. Например,

вычисление конструкции `[1, 2, 3, 4] !! 2` приведет к выдаче в качестве результата числа `3` – элемента списка, имеющего номер `2`.

`null` – функция проверки пустоты списка. Если в качестве аргумента этой операции будет задан пустой список, то функция выдаст значение `True`, в противном случае – `False`.

`length` – функция, вычисляющая количество элементов в списке (длину списка). Если аргумент – пустой список, то результатом функции будет ноль.

`++` – операция соединения двух списков. В результате применения этой операции к двум спискам из элементов одного и того же типа получается новый список из элементов того же типа, составленный из элементов первого списка, за которыми следуют элементы второго списка. Часто эта операция используется для соединения (катенации) двух строк.

Давайте с помощью некоторых из этих функций запрограммируем вычисление суммы элементов списка. Оформим эту программу в виде функции `sumList`, получающей список из целых значений в качестве аргумента и выдающей их сумму (листинг 1.12).

Листинг 1.12. Суммирование элементов списка

```
-- функция суммирования элементов целочисленного списка.
sumList :: [Integer] -> Integer
sumList s | null s = 0
          | otherwise = head s + sumList (tail s)
```

Во втором уравнении для функции `sumList` аргумент функции `s` «разбирается на части» применением встроенных функций `head` и `tail`. На самом деле нет необходимости делать дважды декомпозицию одного и того же списка. Поскольку любой список «собирается» с помощью конструктора списков, то и его «разборку» достаточно сделать один раз – на первый элемент и остаток списка. Это можно сделать с помощью аппарата сопоставления с образцом, при котором в уравнении для определения функции записывается «образец» – формальное выражение с конструктором, в котором отдельные части списка – его первый элемент и остаток – обозначены отдельными идентификаторами. Так, второе уравнение в определении функции `sumList` могло бы выглядеть так:

```
sumList (x:s) = x + sumList s
```

Если в качестве аргумента функции будет передан непустой список, то при попытке использовать второе уравнение произойдет сопоставление аргумента с образцом `(x:s)`. Поскольку непустой список всегда имеет форму, образованную с помощью конструктора списка (явно или неявно), то сопоставление пройдет успешно, причем в результате сопоставления произойдет «разборка» списка на первый элемент (голову) и остаток (хвост) списка. Идентификатор `x` начнет обозначать значение головы, а `s`

будет обозначать хвост списка. В правой части уравнения эти идентификаторы используются при рекурсивном вызове функции и в операции сложения целых.

Первое уравнение тоже можно переписать с использованием сопоставления с образцом, только в качестве образца надо написать пустой список. Сопоставление с образцом по первому уравнению в этом случае будет успешным, только если в качестве аргумента функции задан пустой список. Теперь полный текст функции будет выглядеть так (листинг 1.13):

Листинг 1.13. Суммирование элементов списка

```
-- функция суммирования элементов целочисленного списка.
sumList :: [Integer] -> Integer
sumList []    = 0
sumList (x:s) = x + sumList s
```

Заметим, что первый вариант функции `sumList` можно легко изменить таким образом, чтобы суммировать элементы, начиная с последнего по направлению к началу списка. Для этого вместо выражения `head s + sumList (tail s)` надо просто использовать выражение `last s + sumList (init s)`. Во втором варианте суммировать элементы с конца не получится, поскольку конструктор списков асимметричный: он присоединяет первый элемент к остатку списка, но не может присоединить последний элемент к началу списка. На самом деле и в первом варианте, несмотря на кажущуюся легкость изменения порядка суммирования, суммирование все же следует проводить от начала списка к концу. Дело в том, что встроенные функции `last` и `init` работают существенно дольше функций `head` и `tail` и требуют для работы больше памяти. Фактически каждая из этих функций при работе просматривает весь список от начала до конца, а функция `init` еще и строит заново список из всех начальных элементов! В этом легко убедиться, если посмотреть, как реализованы эти функции в стандартной библиотеке. Вот как выглядят там, например, основные два уравнения в определении функции `init`:

```
init [x]        =  []
init (x:xs)     =  x : init xs
```

Напротив, функции `head` и `tail` реализованы очень просто и не содержат никаких вспомогательных функций или рекурсивных вызовов.

Скорость работы встроенных функций определения длины списка и соединения списков также линейно зависит от размера списка-аргумента. Рассмотрим, например, реализацию уравнений для функции соединения списков. Обратите внимание на форму записи уравнений: функция представлена инфиксной операцией, поэтому в левых частях уравнений формальные вызовы также удобно записать в инфиксной форме.

```
[] ++ s         = s
```

```
(x:s1) ++ s2    = x : (s1 ++ s2)
```

Мы не указывали для описываемых встроенных функций их типов. Фактически аргументами этих функций могут быть любые списки независимо от типов их элементов; надо только, чтобы типы этих списков «соответствовали» друг другу. Например, в операции соединения списков "++" аргументы должны быть списками из элементов одного и того же типа. Результатом, конечно, будет список с элементами того же самого типа. Для таких функций используют *переменные типа*, указывая для тех типов, которые должны совпадать, одинаковые идентификаторы. Вот как, например, можно описать тип функции `length`:

```
length  ::  [a] -> Int
```

В этом описании использована переменная типа `a`, фактически это описание говорит о том, что аргументом функции `length` может быть произвольный список, а результатом работы функции будет короткое целое число. Сходным образом тип операции соединения списков будет описан как

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

то есть два операнда этой операции должны быть списками одного и того же типа (обозначенного идентификатором `a`), при этом результат будет списком с элементами того же типа, что и у операндов. Функции, в определении типа которых участвуют переменные типа, называются полиморфными функциями. Полиморфная функция может иметь аргументы типа, который не определен в момент описания функции, а определяется только в момент применения этой функции к конкретному аргументу.

Опишем еще одну функцию обработки списков, которая, получая список в качестве своего аргумента, выдает список с теми же элементами, но расположенными в этом списке в обратном порядке (обращение списка). В этой функции можно также с помощью сопоставления с образцом «разобрать» исходный список на элементы, однако, собирать его придется уже не с помощью конструктора списков, а с помощью операции соединения списков. Вот как может выглядеть текст функции (листинг 1.14):

Листинг 1.14. Обращение списка

```
-- обращение списка
reverse       :: [a] -> [a]
reverse []    =  []
reverse (x:s) =  reverse s ++ [x]
```

Если внимательно проанализировать текст функции, то видно, что время ее работы пропорционально квадрату длины списка-аргумента. Конечно, это слишком много; интуитивно ясно, что задачу можно решить за линейное время. Проблему легко решить с использованием

накапливающих параметров. Для этого придется ввести вспомогательную рекурсивную функцию, которая будет по сравнению с основной версией иметь лишний аргумент. Новая версия функции будет иметь следующий вид (листинг 1.15):

Листинг 1.15. Более эффективный вариант функции обращения списка

```
-- обращение списка
reverse            :: [a] -> [a]
-- дополнительный аргумент вспомогательной функции reverse'
-- служит для накопления результата
reverse'           :: [a] -> [a] -> [a]
reverse s          =  reverse' [] s
reverse' s []      =  s
reverse' s1 (x:s2) =  reverse' (x:s1) s2
```

Описание функции `reverse` также есть в стандартной библиотеке *Haskell*, но если вы посмотрите на то, как там определена эта функция, то вы не увидите ничего похожего ни на первый, ни на второй вариант. Позже мы научимся писать такие функции в том стиле, который продемонстрирован в стандартной библиотеке.

Ну, и в заключение раздела еще одна задача на обработку списков. В этой задаче рассматривается чуть более сложная структура данных – список строк, то есть список, элементами которого также служат списки – списки символов. Пусть требуется определить, имеются ли в заданном списке строк «белые» строки. Мы будем называть «белой» строку, которая либо совсем не содержит символов, либо содержит только пустые символы, то есть символы пробелов, переводов строк и т.п. Для проверки того, является ли данный конкретный символ пустым, мы будем использовать библиотечную функцию *Haskell* `isSpace`, которая описана в модуле `Char`. Для подключения библиотечных модулей в *Haskell* используется предложение `import`, которое мы и добавим в начало текста нашей программы.

Для решения задачи сначала определим функцию, которая будет проверять, является ли строка «белой». Это очень простая функция, которую легко определить с помощью простой рекурсии. После этого основная задача решается уже также просто: надо просто последовательно перебрать все строки, проверяя «цвет» каждой из них. Если будет обнаружена «белая» строка, то функция тут же завершит работу с результатом `True`. Если ни одной «белой» строки не обнаружится, то результатом работы функции будет `False`. В листинге 1.16 представлено полное решение задачи.

Листинг 1.16. Функция проверки наличия в списке «белых» строк

```
import Char
-- функция определения "цвета" строки
```

```
white                       :: String -> Bool
white []                    = True
white (x:s) | isSpace x     = white s
            | otherwise     = False

-- функция, проверяющая, имеются ли в списке строк "белые"
hasWhites                   :: [String] -> Bool
hasWhites []                = False
hasWhites (x:s) | white x   = True
                | otherwise = hasWhites s
```

Отметим, что решение задачи записано очень просто и интуитивно понятно. Применение функционального стиля программирования помогает выразить не столько то, *как* надо решать задачу, сколько *что* именно нам надо получить.

Примеры решения задач

Задание 1. Написать функцию, определяющую количество строк в списке, содержащих хотя бы одну букву (буквой будем называть символ, для которого функция `isAlpha :: Char -> Bool` выдает значение `True`).

Решение. Прежде всего определим тип для нашей функции. Аргументом функции является список строк, а результатом – целое число, не превышающее длины списка. Таким образом, тип функции может быть задан следующим образом:

```
hasLetters :: [String] -> Int
```

Теперь определим метод решения задачи. Очевидно, что базовым случаем для рекурсии может быть случай пустого исходного списка с очевидным результатом работы функции – 0. Теперь постараемся свести решение задачи для непустого списка к решению той же задачи для более короткого списка. Очевидно, что для этого нам понадомится вспомогательная функция, определяющая, есть ли в заданной строке хотя бы одна буква. Тип этой вспомогательной функции может быть задан следующим образом:

```
hasLetter :: String -> Bool
```

Если предположить, что вспомогательная функция уже написана и работает, то наша основная функция может быть определена с помощью следующих двух уравнений:

```
hasLetters [] = 0
hasLetters (s:ls) =
               (if hasLetter s then 1 else 0) + hasLetters ls
```

Аналогично можно проанализировать и написать функцию для решения вспомогательной задачи: в пустой строке букв нет, в непустой строке, начинающейся с буквы, буквы, очевидно, есть, а в непустой строке, начинающейся с символа, не являющегося буквой, результат определяется

наличием букв в «хвосте» строки, получающемся из исходной строки отбрасыванием первой буквы. Отсюда следуют три уравнения для вспомогательной функции.

```
hasLetter [] = False
hasLetter (x:s) | isAlpha x = True
                | otherwise = hasLetters' s
```

Итак, искомая программа для решения поставленной задачи может выглядеть так, как показано в листинге У1.7. В этом же листинге показан пример оформления задания.

Листинг У1.7. Количество строк, содержащих буквы.

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

import Char(isAlpha)

-- Функция для решения основной задачи.
-- Аргумент: исходный список строк;
-- Результат: число строк, содержащих хотя бы одну букву
hasLetters :: [String] -> Int
hasLetters [] = 0
hasLetters (s:ls) =
             (if hasLetter s then 1 else 0) + hasLetters ls

-- Вспомогательная функция для определения наличия букв
-- в заданной строке.
-- Аргумент: исходная строка;
-- Результат: True, если в строке имеется буква,
--            False в противном случае.
hasLetters' :: String -> Bool
hasLetters' [] = False
hasLetters' (x:s) | isAlpha x = True
                  | otherwise = hasLetters' s

-- Тестовые примеры
test1 = ["Hello, world!", "", "12345", "1-a"]
test2 = ["*", "1+2", "", "17"]
test3 = []
test4 = ["Volga", "Neva", "Neman", "X17"]

-- Тестовая функция
-- Ожидаемый результат: [2, 0, 0, 4]
main = [hasLetters test1, hasLetters test2,
        hasLetters test3, hasLetters test4]
```

В начале программы указано, что для решения задачи используется функция `isAlpha` из библиотечного модуля `Char`. Это сделано с

помощью предложения `import`. Подробнее об импорте модулей можно посмотреть в приложении к данной книге.

Обратите внимание на оформление программы. Каждая функция требует отдельного комментария, в котором указывается, что функция делает, какие имеет аргументы и каков их смысл, каков результат работы функции. В конце текста указаны несколько характерных тестовых примеров, проверяющих правильность работы функции для типичных исходных данных, а также для «крайних» случаев (пустые списки, пустые строки и т.п.). Тестовая функция представляет собой выражение, в котором комбинируется вызов целевой функции для тестовых примеров.

Подобное оформление будет использоваться и в дальнейшем без дополнительных комментариев.

Задание 2. Написать функцию, вычисляющую число сочетаний из *n* предметов по *m*.

Решение. Формула для числа сочетаний `C(n, m)` может быть записана несколькими разными способами. Наиболее употребительным является способ записи с помощью функции вычисления факториала натурального числа:

```
C(n, m) = n! / (m! * (n-m)!)
```

Эту формулу можно использовать для непосредственного программирования искомой функции, а для этого сначала определим функцию вычисления факториала заданного натурального числа. Получающаяся при этом программа может выглядеть так, как представлено в листинге У1.8.

Листинг У1.8. Число сочетаний из *n* по *m* (первый вариант)

```
-- Функция вычисления факториала числа
fact :: Integer -> Integer
fact 0 = 1
fact n = n * fact (n-1)

-- Функция вычисления числа сочетаний
comb :: Integer -> Integer -> Integer
comb n m | (n < 0) || (m < 0) || (n < m) = 0
         | otherwise = fact n `div` fact m `div` fact (n-m)
```

Разумеется, это не самый эффективный способ вычисления функции. Если сократить дробь, представленную в формуле, то можно получить несколько более эффективную формулу вычисления того же самого значения:

```
C(n, m) = (n * (n-1) *... *(n-m+1)) / m!
```

Программирование этой формулы предоставляется читателям для несложной самостоятельной работы. Заметим только, что можно минимизировать число сомножителей, воспользовавшись тем, что `C(n, m) = C(n, n-m)`. Мы же воспользуемся прямым рекуррентным

соотношением, непосредственно вытекающим из только что приведенной формулы:

```
C(n, 0) = 1
C(n, m) = C(n-1, m-1) * n / m
```

Это почти готовая программа на языке Haskell. Полностью функция вместе с ее оформлением приведена в листинге У1.9.

Листинг У1.9. Число сочетаний из n по m

```
-- Задание: Написать функцию, вычисляющую число сочетаний
--          из n предметов по m.

-- Основная функция вычисления числа сочетаний из n по m
-- Аргументы: n, m – исходные аргументы функции
-- Результат: число сочетаний из n по m.
comb :: Integer -> Integer -> Integer
comb n m | (n < 0) || (m < 0) || (n < m) = 0
         | n < 2 * m = comb' n (n-m)
         | otherwise = comb' n m

-- Вспомогательная рекурсивная функция для вычисления C(n, m)
comb' :: Integer -> Integer -> Integer
comb' n 0 = 1
comb' n m = (comb' (n-1) (m-1)) * n `div` m

-- Тестовая функция
-- Ожидаемый результат: [15, 120, 120, 0, 0]
main = [comb 6 2,
        comb 10 3,
        comb 10 7,
        comb 2 5,
        comb 5 (-3)]
```

Задание 3. Написать функцию, вычисляющую длину самой длинной неубывающей последовательности подряд идущих чисел в заданном списке целых. Например, в списке `[2, 5, 12, 9, 10, 4, 7, 8]` самыми длинными неубывающими последовательностями чисел являются `[2, 5, 12]` и `[4, 7, 8]`, так что для заданного списка функция должна выдавать результат 3.

Решение. Простого способа свести решение задачи к решению той же задачи для более короткого списка найти не удается. Действительно, если известно, что длина максимальной неубывающей подпоследовательности (МНП) в списке, состоящем из n элементов, равна некоторому k, то непосредственно из этого факта невозможно найти длину МНП в списке длиной (n+1) элемент. Расширим постановку задачи. Попробуем найти для данного списка пару чисел: длину его МНП и длину максимальной неубывающей последовательности элементов, начинающуюся с первого элемента (МНП1; возможно, это будет одна и та

же последовательность, МНП = МНП1). Оказывается, рекурсивное решение для этой более общей задачи найти легче.

Пусть список *list* состоит из более, чем одного элемента. Обозначим его первый элемент (голову) через *x*, остаток списка (хвост) через *tail*, и его второй элемент через *y*. Можно сказать, что наш список удовлетворяет образцу `list@(x:tail@(y:_))`. Пусть для хвоста списка задача решена, и мы нашли пару чисел `(n1, nmax)`, означающие длину МНП1 списка `tail` и длину его МНП. Очевидно, что `1 <= n1 <= nmax`. Если `x > y`, то для всего списка `list` значение МНП останется тем же самыми, а длина МНП1 будет равна единице. Если же `x <= y`, то МНП1 для списка `list` будет на единицу больше, чем для списка `tail`, а его МНП окажется максимальным значением из нового значения МНП1 и старого значения МНП для списка `tail`. Отсюда немедленно следует программа, приведенная в листинге У1.10.

Листинг У1.10. Длина максимальной неубывающей последовательности элементов списка

```
-- Задание: Написать функцию, вычисляющую длину наибольшей
--          последовательности подряд идущих элементов
--          (МНП) заданного списка целых.

-- Вспомогательная рекурсивная функция для вычисления пары
-- МНП и МНП1 (длины наибольшей неубывающей последовательности
-- элементов, начинающейся с первого элемента списка).
-- Аргумент: исходный список.
-- Результат: пара значений (МНП1, МНП).
uplength2 :: [Integer] -> (Int, Int)
uplength2 [] = (0, 0)  -- в пустом списке длины равны нулю.
uplength2 [x] = (1, 1)
uplength2 (x:tail@(y:_))
     | x > y     = (1, nmax)
     | otherwise = (n1 + 1, max (n1 + 1) nmax)
         where (n1, nmax) = uplength2 tail

-- Основная функция решения поставленной задачи.
-- Аргумент: исходный список.
-- Результат: длина МНП этого списка.
uplength :: [Integer] -> Int
uplength list = snd (uplength2 list)

-- Тестовая функция
-- Ожидаемый результат: [2, 3, 1, 0]
main = [uplength [2, 3, 2, 1, 0, 5, 4],
        uplength [2, 3, 1, 2, 5, 3, 2],
        uplength [5, 4, 3, 2, 1],
        uplength []]
```

В этой программе для обозначения результатов рекурсивного вызова функции использована конструкция `where`, которая более подробно обсуждается далее. В основной функции `uplength` для извлечения второго элемента кортежа использована стандартная функция `snd`.

Задания для самостоятельной работы

Во всех нижеследующих заданиях требуется написать программу для решения поставленной задачи на языке Haskell.

Задание 1. Написать программу для нахождения *n*-го члена последовательности, заданной следующей рекуррентной формулой.

$a_0 = 1;\ a_1 = 2;$

$a_n = 3 * a_{n-1} - 2 * a_{n-2} + 1$ при $n = 2, 3, \ldots$

Имейте в виду, что прямое программирование данной формулы «как есть» приводит к крайне неэффективной программе!

Задание 2. Совершенным числом называется натуральное число, равное сумме всех своих делителей, включая единицу, но исключая само это число. Так, например, число 28 – совершенное, поскольку 28 = 1 + 2 + 4 + 7 + 14. Написать программу для нахождения первых *n* совершенных чисел.

Задание 3. Близнецами называется пара натуральных чисел, каждое из которых равно сумме делителей другого числа. Так, например, числа 220 и 284 – близнецы (проверьте!). Написать программу для нахождения первых *n* пар близнецов.

Задание 4. В заданном списке строк найти самую длинную строку.

Задание 5. По заданному списку строк построить список строк, в котором содержатся те же строки, что и в исходном списке, но каждая вторая строка выброшена из списка (то есть в списке останутся только строки с нечетными номерами), а в каждой из оставшихся строк каждый второй символ также выброшен. Так, например, если аргументом программы является список строк

```
["Близнецами", "называется", "пара", "натуральных", "чисел"]
```

то результатом работы должен быть список строк

```
["Бинцм", "пр", "чсл"]
```

Задание 6. По заданному списку строк построить список строк, в котором содержатся те же строки, что и в исходном списке, но выброшены строки, содержащие хотя бы одну цифру. Проверить, является ли некоторый символ `c` цифрой, можно с помощью вызова стандартной функции *Haskell* `Char.isDigit`.

Задание 7. Заданный числовой список разбить на подсписки из возрастающих подпоследовательностей максимальной длины рядом стоящих чисел. Так, например, если исходный список состоял из чисел [2, 7, 10, 8, 3, 4, 9, 1, 2, 0, 8, 3, 2, 5], то результатом работы программы должен

быть следующий список списков: [[2, 7, 10], [8], [3, 4, 9], [1, 2], [0, 8], [3], [2, 5]].

Задание 8. Для заданного вещественного числа x найти сумму числового ряда с общим членом $u_n = x^n / (n!+1)$. Суммирование производить, пока очередной член ряда не окажется по абсолютной величине меньше заданного числа.

1.3. Определение новых типов данных

Вообще говоря, любую программу можно написать, используя только встроенные простые типы данных, кортежи и списки, однако для удобства организации данных в современных языках программирования предоставляется аппарат определения новых типов данных. В языке *Haskell* также имеются механизмы, с помощью которых можно определить свои собственные типы данных или ввести свои обозначения для уже имеющихся типов, а также определить допустимый набор операций над объектами (класс типа).

С помощью конструкции введения синонима для типа можно определить новый идентификатор для уже имеющегося типа. Эта конструкция начинается словом `type`. Например, если мы хотим определить идентификатор `Pair` для обозначения пары (кортежа) из двух вещественных чисел, то в программе следует ввести следующее описание:

```
type Pair = (Double, Double)
```

Напомним, что идентификаторы типов в *Haskell* всегда начинаются с заглавной буквы, так что и здесь, определяя новый идентификатор типа, мы начали его с заглавной буквы `P`. Для объектов типа `Pair` теперь можно определять новые функции. Например, если считать, что точка плоскости представляется парой вещественных координат, то функцию определения расстояния между двумя точками плоскости можно описать так, как показано в листинге 1.17.

Листинг 1.17. Вычисление расстояния между двумя точками

```
distance :: Pair -> Pair -> Double
distance (x1, y1) (x2, y2) =
         sqrt ((x2 - x1) * (x2 - x1) + (y2 - y1) * (y2 - y1))
```

В приведенной программе идентификатор функции `sqrt` обозначает стандартную функцию вычисления квадратного корня из вещественного числа.

Понятие пары можно несколько обобщить, если считать, что пара может быть составлена из двух объектов произвольного типа, а не обязательно вещественных чисел. Единственное условие – это то, что тип обеих компонент пары должен быть одним и тем же. Раньше мы определяли полиморфные функции, просто вводя переменную типа в определение типа функции. Аналогично мы можем при задании типа `Pair`

ввести переменную типа для обозначения типа объектов, составляющих пару. Тогда определение пары будет иметь следующий вид:

```
type Pair a = (a, a)
```

Если мы по-прежнему хотим определять функцию, работающую с парами вещественных чисел, то при определении типа такой функции следует задать в качестве фактического параметра тип `Double`. Например, если мы считаем, что пара вещественных чисел представляет координаты двумерного вектора, то функция сложения векторов `addVector` может быть определена следующим образом (листинг 1.18):

Листинг 1.18. Сложение векторов на плоскости

```
addVector :: Pair Double -> Pair Double -> Pair Double
addVector (x1, y1) (x2, y2) = (x1 + x2, y1 + y2)
```

Конечно, можно определять и полиморфные функции, которые будут использовать тип `Pair` независимо от типа составляющих пару значений. Например, опишем функцию, которая просто переставляет местами компоненты пары, выдавая в результате пару тех же самых объектов, но расположенных в другом порядке.

```
swap :: Pair a -> Pair a
swap (x, y) = (y, x)
```

Определенную нами функцию `swap` можно будет применять к парам значений произвольного типа, например `swap(1, 3)`, `swap('$', '€')` или `swap(True, False)`. Однако, будет выдана ошибка при попытке применить функцию к паре, составленной из значений разных типов, например, `swap(1, True)`.

С помощью задания синонима типа в стандарте языка определен тип строк. Можно считать, что любая программа выполняется в контексте следующего описания:

```
type String = [Char]
```

то есть идентификатор `String` введен в качестве синонима описателя типа `[Char]` стандартным для языка *Haskell* образом.

С помощью описателя синонима `type` определяется лишь новый идентификатор типа, но значениями этого типа будут уже имеющиеся в программе объекты. Так, например, мы определили новый идентификатор типа `Pair`, но значениями объектов этого типа являются кортежи, то есть значения, которые уже существовали до определения типа `Pair`. С помощью другой конструкции языка – `data` – можно определить не только новый идентификатор типа, но и новые объекты, которые и будут значениями этого типа. Для этого помимо нового идентификатора в определении `data` описывают конструкторы объектов, с помощью которых можно создавать новые объекты описываемого типа. В простейшем случае определение нового типа будет просто состоять из

перечисления всех конструкторов, разделенных вертикальной чертой. Например, описание, сделанное ниже в листинге 1.19, задает новый тип данных `WeekDay`, значениями которого будут объекты, созданные с помощью конструкторов `Monday`, `Tuesday` и т.д.

Заметим, что имена конструкторов, так же, как и имена новых описываемых типов, всегда начинаются с заглавной буквы. Простенькая функция `weekend`, определяющая, является ли заданный день выходным, может выглядеть так, как это продемонстрировано в листинге 1.19.

Листинг 1.19. Определение и обработка дней недели

```
data WeekDay = Sunday | Monday | Tuesday | Wednesday |
               Thurthday | Friday | Saturday
weekend :: WeekDay -> Bool
weekend Sunday     = True
weekend Saturday   = True
weekend _          = False
```

Эту функцию мы описали с помощью трех уравнений. В первых двух в качестве образца для сопоставления использованы константы, заданные с помощью конструкторов `Sunday` и `Saturday`. В третьем уравнении образцом является безымянная переменная `'_'`. Идентификатор `'_'` можно использовать только для задания образцов, содержащих безымянные переменные. Безымянная переменная ведет себя точно так же, как и обычная переменная, но ее нельзя использовать в правой части уравнения, поэтому она используется там, где значение, сопоставленное с переменной, не содержится в правой части уравнения.

При вызове функции `weekend` ее аргумент будет последовательно сопоставляться сначала с образцом `Sunday`, потом с образцом `Saturday` и, наконец, если ни тот, ни другой образцы не подойдут, то будет произведено последнее сопоставление с образцом `'_'`. Это последнее сопоставление обязательно будет успешным, и в этом последнем случае значением функции окажется `False`.

Можно считать, что стандартный тип данных `Bool` тоже определен так, как это описано выше, при этом определяемыми в этом типе данных конструкторами являются `False` и `True`. Таким образом, любая программа действует в контексте описания типа

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

Вообще говоря, подобным же образом можно описать и другие стандартные типы, определенные в языке, такие как `Int`, `Char` и другие, но в качестве конструкторов данных этих типов выступают соответствующие литеральные обозначения, такие как `25` (типа `Int`) или `'@'` (типа `Char`). Разумеется, литералы использовать в качестве имен конструкторов в языке запрещено – литералы всегда обозначают только ту константу, которую они изображают, так что, говоря о том, что

стандартные типы `Char` и `Int` определяются «точно так же», как и тип `Bool`, мы говорим только о том, что это *принципиально* такое же определение.

Новые типы и конструкторы, создаваемые с помощью описателя `data`, как и те, что создаются с помощью описателя `type`, могут быть параметризованными. Выше мы определили тип данных `Pair`, просто обозначив этим идентификатором кортеж из пары значений. Если мы хотим подобным же образом ввести совершенно новый тип, то следует воспользоваться описателем `data`. Вот как можно определить тип данных `Point` с одним конструктором, с помощью которого можно создавать новые точки вещественной плоскости:

```
data Point = Pt Double Double
```

Здесь для нового типа данных `Point` определен единственный конструктор `Pt` с двумя вещественными аргументами. Фрагмент программы, задающий точку с координатами `(1.5, 2.5)` в виде нового объекта типа `Point`, может выглядеть так:

```
Pt 1.5 2.5
```

Приведем вариант функции, вычисляющей расстояние между двумя точками плоскости, который использует вышеприведенное определение типа точки (листинг 1.20).

Листинг 1.20. Вычисление расстояния между двумя точками

```
dist :: Point -> Point -> Double
dist (Pt x1 y1) (Pt x2 y2) =
     sqrt ((x2 - x1) * (x2 - x1) + (y2 - y1) * (y2 - y1))
```

В этом варианте образцы для аргументов представлены уже не кортежами, а конструкторами, определенными при описании типа `Point`. В остальном определение функции в точности соответствует описанному выше в листинге 1.17 определению функции `distance`.

Более сложной будет ситуация, при которой в определении нового типа данных имеется параметр – переменная типа. Например, вместо определения нового идентификатора типа `Pair` из вышеприведенного примера можно аналогично ввести новый тип данных и новый конструктор для создания пар объектов одного и того же типа:

```
data Pair a = Couple a a
```

В этом примере `Pair` – идентификатор нового типа данных с конструктором `Couple`. Параметром типа является переменная типа `a`. Описанная выше функция сложения двумерных векторов теперь может быть описана как функция двух аргументов этого нового типа (листинг 1.21):

Листинг 1.21. Сложение векторов на плоскости

```
addVector :: Pair Double -> Pair Double -> Pair Double
addVector (Couple x1 y1) (Couple x2 y2) =
          Couple (x1 + x2) (y1 + y2)
```

Обратите внимание, как при новом описании функции используются идентификаторы нового типа `Pair` и конструктора этого типа `Couple`. В левой части уравнения конструктор `Couple` использован для определения образца, с которым будет сопоставляться фактический аргумент функции. В правой части – тот же конструктор используется уже для конструирования новых значений `Couple (x1 + x2) (y1 + y2)`. С помощью конструктора здесь создается новый объект типа `Pair Double`, поскольку аргументы конструктора – это значения типа `Double`.

Идентификаторы типов и идентификаторы конструкторов встречаются в разных ситуациях; там, где допустим идентификатор типа, никогда не может встретиться конструктор, и наоборот, там, где может стоять конструктор объекта, никогда не может быть идентификатора типа. Говорят, что типы и конструкторы объектов принадлежат разным *пространствам имен*. Это позволяет использовать одни и те же идентификаторы как для наименования типов, так и для имен конструкторов. Чтобы вас не запутать, мы не будем злоупотреблять этой возможностью, однако при описании типа с единственным конструктором хорошей традицией является использование одного и того же идентификатора как для наименования самого типа, так и его конструктора. В только что приведенном примере можно было бы использовать идентификатор `Pair` в обеих ситуациях. Определение типа данных и функции сложения векторов тогда могли бы выглядеть следующим образом (листинг 1.22):

Листинг 1.22. Сложение векторов на плоскости

```
data Pair a = Pair a a

addVector :: Pair Double -> Pair Double -> Pair Double
addVector (Pair x1 y1) (Pair x2 y2) = Pair (x1 + x2) (y1 + y2)
```

Здесь в определении типа функции `addVector` используется идентификатор типа `Pair`, а в определяющем уравнении для этой функции тот же идентификатор уже используется как идентификатор конструктора `Pair`.

Определение нового типа может быть рекурсивным, когда идентификатор определяемого типа используется в качестве одного из параметров в правой части определения. Определим, например, альтернативный тип для списков, состоящих из целых чисел. Традиционное определение списка будет содержать два конструктора, один из которых предназначен для создания пустого списка, а другой

позволяет присоединить новый элемент (в данном случае – целое число) к уже имеющемуся списку. Первый конструктор не имеет параметров, однако во втором конструкторе необходимо задать целое число и список, то есть объект определяемого нами типа. Назовем новый тип `IntList`:

```
data IntList = Nil | Cons Integer IntList
```

Объекты этого нового типа списков могут обрабатываться точно так же, как мы делали это раньше со встроенными в язык списками. Вот, например, (листинг 1.23) определение функции, выдающей максимальное значение среди всех элементов списка (функция неприменима к пустому списку, так как в этом случае максимальное значение определить невозможно).

Листинг 1.23. Определение максимального элемента списка

```
maxList :: IntList -> Integer
maxList Nil           = error "maxList: empty agrument"
maxList (Cons x Nil)  = x
maxList (Cons x s)    = max x (maxList s)
```

Конечно, приведенное определение списка очень ограничено, поскольку определяет только списки, состоящие из целых чисел. Более общее определение получится, если ввести параметр – переменную типа, обозначающую тип элементов списка. В этом случае определение более общего типа списка, состоящего из элементов произвольного типа, могло бы выглядеть так:

```
data List a = Nil | Cons a (List a)
```

В этом новом определении типа списка идентификатор `List` – это идентификатор нового типа, а в качестве имен конструкторов пустого и непустого списков как и в предыдущем определении используются идентификаторы `Nil` и `Cons`. Конечно, это более общее определение списка удобнее использовать, чем предыдущее определение списка из целых. Если мы по-прежнему хотим определить функцию, определяющую максимальный элемент из списка целых, то такую функцию можно будет теперь определить следующим образом:

```
maxList :: List Integer -> Integer
maxList Nil           = error "maxList: empty agrument"
maxList (Cons x Nil)  = x
maxList (Cons x s)    = max x (maxList s)
```

Хотелось бы, однако, иметь и более общий вариант этой функции, поскольку поиск максимального значения возможен не только для целочисленных списков, но и вообще для списков, содержащих значения, которые можно сравнивать друг с другом. Если мы попытаемся просто переопределить функцию `maxList` так же, как мы это только что сделали, только сделав ее полиморфной (то есть, введя параметр типа), то увидим, что определение функции становится бессмысленным. Действительно,

если повторить определение, сделав чисто формально замену идентификатора `Integer` на параметр `a`, то мы получим следующий текст:

```
maxList :: List a -> a
maxList Nil           = error "maxList: empty agrument"
maxList (Cons x Nil)  = x
maxList (Cons x s)    = max x (maxList s)
```

Здесь, однако, стандартная функция `max` вычисления максимального из двух значений применяется не к целым числам, а к значениям некоторого, вообще говоря, произвольного типа `a`. Это, разумеется, недопустимо. Функция `max` применима только к таким типам данных, которые позволяют сравнение значений этого типа друг с другом. Подобного рода ограничения могут возникнуть и в других случаях, когда при определении достаточно общих функций потребуется обращаться к другим функциям и операциям, имеющим определенные ограничения на типы аргументов. Для того, чтобы можно было задавать и контролировать подобные ограничения, в языке *Haskell* введено понятие класса, которое мы рассмотрим немного позже. Пока же только скажем, что класс, для которого определена функция `max`, называется `Ord`, так что для того, чтобы определение функции `maxList` было корректным, надо указать, что тип элементов списка, являющегося аргументом этой функции, должен принадлежать классу `Ord`.

Ограничения на используемые в описаниях типы вводятся с помощью конструкции, которую мы будем называть *контекстом* типа. Для того, чтобы определение полиморфной функции `maxList` стало синтаксически правильным, описание типа этой функции надо записать в следующем виде:

```
maxList :: Ord a => List a -> a
```

тогда определение функции, приведенное выше, станет корректным.

Конечно, на самом деле нет смысла вводить новое определение для типа списков, поскольку гораздо удобнее обходиться стандартными списками, определенными в языке. Мы привели это новое определение, только чтобы показать, как подобные структуры могут определяться в языке, а также, чтобы продемонстрировать, что стандартные списки языка *Haskell* на самом деле тоже могут быть определены чисто языковыми средствами. Правда, определить списки так, чтобы синтаксическая форма записи типа и его конструкторов в точности совпадала бы с принятыми в *Haskell* формами, не удастся, однако все-таки приведем определение очень близкое к синтаксически правильному определению стандартных списков:

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

В этой записи в качестве "идентификатора типа" вместо ранее использовавшегося нами имени `List` используется стандартная запись с

квадратными скобками (получается, что параметр этого типа `a` располагается между скобками). В качестве конструктора пустого списка использовано литеральное обозначение `[]`, которое в действительности, как и в случае с другими литералами, нельзя использовать в качестве имени конструктора. Другой конструктор (тот, для которого мы ранее выбрали идентификатор `Cons`) теперь обозначается двуместным оператором `':'`. Надо заметить, что на этот раз синтаксические правила все же соблюдены, потому что как для имен функций, так и для имен конструкторов разрешено использовать двуместные операторы, образованные последовательностями литер из некоторого предопределенного списка символов, в который, в частности, входит и символ двоеточия. Чтобы обеспечить визуальную различимость обозначений функциональных операций и операций-конструкторов, принято соглашение, согласно которому названия операций-конструкторов должны начинаться с двоеточия, а названия функциональных операций, напротив, с символа двоеточия начинаться не могут. Это соглашение аналогично тому, что идентификаторы функций должны начинаться со строчных букв, а идентификаторы конструкторов – с прописных. Кстати, в вышеприведенном определении «стандартных» списков это соглашение выполняется: конструктор для формирования списков действительно имеет обозначение, начинающееся с двоеточия, точнее, состоит только из двоеточия.

Вообще, конструкторы и функции имеют много общих черт. И конструктор, и функция служат для образования новых значений из имеющихся (конструкторы без аргументов, скорее, напоминают не функции, а значения-константы, заданные литерально). Разница состоит в том, что функции для образования нового значения требуются правила, с помощью которых это значение и образуется, а конструктору никаких правил не нужно – аргументы собираются конструктором в единый объект нового типа чисто механически. Позже мы увидим, что такое сходство конструкторов и функций позволяет в ряде случаев использовать и то и другое абсолютно одинаковым образом.

В качестве чуть более сложного примера задачи обработки списков приведем пример функции, которая сортирует заданный список элементов, то есть располагает его элементы в порядке возрастания. Это достаточно традиционная для программирования задача, однако отметим две ее особенности при решении ее в функциональном стиле. Во-первых, задачу можно поставить и решить для любых списков, элементы которых можно сравнивать друг с другом по величине, то есть если тип этих элементов принадлежит классу `Ord`. Получившаяся функция будет похожа по своей общности на *шаблон функции*, который можно описать, например в языке *Си++*. В языке *Haskell* это будет обычной (полиморфной) функцией. Во-вторых, при использовании традиционных языков обычно полагают, что

элементы сортируемой структуры данных – массива или списка – остаются «внутри» этой структуры, то есть элементы перемещаются внутри структуры данных. При функциональном подходе к программированию вместо изменения структур данных каждый раз строится новый объект. Функция сортировки списка при таком подходе – это функция, которая, получив некоторый список в качестве аргумента, строит в качестве результата новый список, содержащий те же самые элементы, но расположенные в порядке возрастания.

В этом примере мы, наряду с определением основной функции, задействуем вспомогательную функцию `insert`, которая вставляет новый элемент в уже упорядоченный фрагмент списка. Именно эта функция и строит фактически новый упорядоченный список на базе уже имеющегося, добавляя к нему новый элемент (листинг 1.24).

Листинг 1.24. Сортировка элементов списка «простыми вставками»

```
-- сортировка элементов списка
simpleSort      :: Ord a => [a] -> [a]
insert          :: Ord a => a -> [a] -> [a]

simpleSort []       = []
simpleSort (x:s)    = insert x (simpleSort s)

insert elem []  = [elem]
insert elem list@(x:s) | elem < x   = elem:list
                       | otherwise  = x:(insert elem s)
```

В данном примере используется еще одна не использовавшаяся нами ранее конструкция. В определении функции `insert` в левой части второго уравнения образцу `(x:s)` предшествует обозначение `list@`. Это просто означает, что на весь второй аргумент функции можно будет ссылаться в правой части уравнения по имени `list`. Если бы такого обозначения не было введено, то в правой части уравнения пришлось бы написать выражение `elem:x:s`. Конечно, функция все равно осталась бы правильной, однако введение обозначения `list` позволило сделать ее чуть более эффективной: теперь в некоторых случаях после выполнения сопоставления с образцом не обязательно будет нужно собирать заново второй аргумент из составляющих его частей `x` и `s`, а можно сразу же использовать значение аргумента целиком.

Давайте теперь определим тип данных для более сложной структуры данных – двоичного (бинарного) дерева – и напишем несколько функций, аргументами и/или результатом работы которых будут такие деревья.

Двоичным деревом называют структуру данных, содержащую элементы, связанные друг с другом таким образом, что каждый элемент (узел дерева) связан с единственным другим элементом (его *непосредственным предком*), причем этот предок является предком для не

более чем двух других элементов, называемых его *непосредственными потомками*. Ровно один узел из всей совокупности (корень дерева) является исключением из общего правила: у него нет непосредственного предка. Узлы, не имеющие потомков, называются листьями дерева. Нетрудно доказать, что если в дереве имеется *n* узлов, то около половины из них будут листьями, при этом хотя бы один лист обязательно имеется в любом дереве. Иногда рассматривают также и *пустое дерево*, не содержащее узлов вообще, тогда двоичное дерево можно определить рекурсивно следующим образом: двоичным деревом называется либо пустое дерево, либо совокупность из одного узла (корня дерева) и двух других деревьев, называемых левым и правым поддеревьями этого узла. Разумеется, каждое из поддеревьев может в свою очередь быть пустым деревом или узлом со своими поддеревьями.

В основу нашего описания структуры двоичного дерева мы положим этот второй рекурсивный вариант определения дерева. Зададим два конструктора, первый из которых будет порождать пустое дерево (конструктор `Null`), а второй (`Tree`) будет иметь в качестве параметров узел и два его поддерева. Соответствующее описание будет выглядеть следующим образом:

```
data Tree a = Null |
              Tree a (Tree a) (Tree a)
```

Обратите внимание, что здесь, как и в некоторых примерах, приведенных нами ранее, идентификатор `Tree` используется в двух разных смыслах: как идентификатор типа данных и как идентификатор конструктора, причем в конструкции `Tree a (Tree a) (Tree a)` первое вхождение идентификатора `Tree` – это определение конструктора, а два других – рекурсивные ссылки на идентификатор определяемого типа данных.

В качестве простых примеров функций обработки двоичных деревьев приведем две функции: вычисление высоты дерева и подсчет количества листьев в дереве. Обе функции легко определяются рекурсивно с помощью нескольких уравнений, в каждом из которых участвуют образцы аргумента-дерева, соответствующие одному из возможных способов конструирования дерева. Вот как будет выглядеть наше определение функции, вычисляющей высоту заданного дерева (листинг 1.25).

Листинг 1.25. Вычисление высоты двоичного дерева

```
-- функция для определения высоты заданного двоичного дерева;
-- данное определение является хорошим определением
-- самого понятия высоты двоичного дерева
height :: Tree a -> Int
-- высоту пустого дерева полагаем равной нулю
height Null            = 0
```

```
height (Tree _ tl tr)  = 1 + max (height tl) (height tr)
```

Лист дерева задается образцом `(Tree _ Null Null)`, то есть лист – это узел с двумя пустыми поддеревьями. Поэтому функция для подсчета количества листьев в дереве будет выглядеть так (листинг 1.26):

Листинг 1.26. Вычисление количества листьев в двоичном дереве

```
-- функция для подсчета количества листьев в дереве;
-- лист - это узел дерева вида (Tree node Null Null)
leafCount :: Tree a -> Int
-- пустое дерево не содержит листьев:
leafCount Null                = 0
-- если дерево состоит только из листа:
leafCount (Tree _ Null Null)  = 1
-- корень не является листом:
leafCount (Tree _ tl tr)      = leafCount tl + leafCount tr
```

Еще одна очень полезная функция – это функция выстраивания элементов дерева в список (или *разглаживание дерева*). Функция производит обход всех узлов дерева и строит список из этих узлов. Часто порядок обхода узлов и помещения их в список важен для решения определенных задач. Например, иногда важно, чтобы любой узел попадал в список только после своего непосредственного предка (обходы сверху вниз), иногда наоборот. Здесь мы представим функцию, которая будет осуществлять обход таким образом, чтобы любой узел попадал в список после узлов левого поддерева и до узлов правого поддерева (левосторонний обход). Назовем такую функцию `flatten` (сделать плоским, «разгладить») и опишем ее рекурсивно (листинг 1.27).

Листинг 1.27. «Разглаживание» двоичного дерева

```
-- функция разглаживания дерева
flatten :: Tree a -> [a]
flatten Null              = []
flatten (Tree node tl tr) = flatten tl ++ [node] ++ flatten tr
```

Если исходное дерево – упорядоченное, то есть величина любого узла больше значений, хранящихся в левом поддереве этого узла и не меньше значений, хранящихся в правом поддереве, то при его разглаживании список получится упорядоченным по возрастанию. Это позволяет написать еще один алгоритм сортировки элементов списка в дополнение к уже приведенному ранее алгоритму сортировки «простыми вставками». В этом новом варианте элементы исходного списка просматриваются и вставляются по очереди в упорядоченное дерево, после чего отсортированный список получается путем разглаживания сформированного дерева. Приведем текст получившейся программы, состоящей из следующих функций: `treeSort` – основная функция сортировки списка с помощью дерева; `insert` – функция вставки элемента в упорядоченное дерево; `formTree` – функция формирования

упорядоченного дерева из исходного списка последовательными вставками элементов и `flatten` – функция разглаживания дерева (листинг 1.28).

Листинг 1.28. Сортировка списка с помощью двоичного дерева

```
-- Описание типов использующихся для решения задачи функций:
treeSort   :: (Ord a)=> [a] -> [a]
insert     :: (Ord a)=> a -> Tree a -> Tree a
formTree   :: (Ord a)=> [a] -> Tree a
flatten    :: Tree a -> [a]

-- Основная функция сортировки описана просто
-- как суперпозиция других функций:
treeSort s    = flatten (formTree s)

-- Функция вставки элемента в упорядоченное дерево
-- похожа на функцию вставки элемента в упорядоченный список
-- из решения задачи о сортировке методом "простых вставок"
insert elem Null                        =  Tree elem Null Null
insert elem (Tree n tl tr) | elem < n  =
                                  Tree n (insert elem tl) tr
                           | otherwise =
                                  Tree n tl (insert elem tr)

-- Функция создания упорядоченного дерева из элементов списка
formTree []    = Null
formTree (x:s) = insert x (formTree s)

-- Функция разглаживания дерева
flatten Null               = []
flatten (Tree node tl tr) = flatten tl ++ [node] ++ flatten tr
```

Эффективность работы функции сортировки, основанной на построении упорядоченного дерева невысока, несмотря на то, что выбранный алгоритм обычно показывает неплохую производительность при сортировке списков. Кстати, можно заметить, что алгоритм сортировки очень похож на алгоритм «быстрой» сортировки – и там, и там элементы исходной последовательности разделяются на две совокупности элементов: элементы, большие выбранного и элементы, меньшие выбранного. Проблема, однако, состоит в скорости работы рекурсивной функции разглаживания дерева `flatten`. Основное уравнение, определяющее работу этой функции, содержит не только два рекурсивных обращения к этой же функции, но и обращение к операции соединения списков, которая будет просматривать заново уже разглаженную часть дерева только для того, чтобы присоединить ее к другим частям получающегося списка. В результате для дерева, содержащего n узлов, скорость разглаживания будет в среднем пропорциональна $n \times \log_2 n$, а в

худшем случае будет достигать n^2, в то время как интуитивно ясно, что скорость работы хорошего алгоритма разглаживания дерева должна зависеть от количества узлов в дереве линейно.

Скорость работы функции разглаживания дерева можно увеличить, если использовать дополнительные накапливающие аргументы функции, как мы это уже делали раньше, программируя функции для вычисления значения члена последовательности Фибоначчи по его номеру и для обращения списка. Дополнительным накапливающим аргументом для функции разглаживания дерева мог бы быть список, к голове которого присоединяется результат разглаживания дерева. Если обозначить вспомогательную рекурсивную функцию `flatten'`, то результат ее работы определяется следующим соотношением:

```
flatten' tree list = (flatten tree) ++ list
```

Определение функции `flatten'` можно легко построить без использования операции соединения списков. После этого достаточно будет выразить результат работы функции `flatten` через `flatten'`. Вот как будет выглядеть часть программы, содержащая уравнения для функций `flatten` и `flatten'` (листинг 1.29):

Листинг 1.29. Более эффективная функция разглаживания дерева

```
flatten' Null list                = list
flatten' (Tree node tl tr) list  =
                        flatten' tl (node : (flatten' tr list))
flatten tree                      = flatten' tree []
```

В следующей главе мы рассмотрим еще одно определение функции разглаживания дерева, в котором результат тоже будет достигаться за линейное время. А сейчас давайте рассмотрим более подробно аппарат определения классов и их реализаций в языке *Haskell*.

Класс – это набор операций (функций), которые могут применяться к объектам определенных типов. В языке *Haskell* классы определяются формально, то есть просто объявляется список описаний типов для некоторого набора функций и бинарных операций. Конкретный тип будет принадлежать объявленному классу, если для него определены все функции, объявленные в классе. В языке *Haskell* изначально определено довольно большое количество классов, которым принадлежат стандартные типы языка. Так, например, имеется класс `Eq`, содержащий операции сравнения `'=='` и `'!='`. Все стандартные примитивные типы, такие как `Int`, `Char`, `Bool` принадлежат этому классу, так как для них определены соответствующие операции. Для того, чтобы объявить, что заданный тип принадлежит некоторому классу, надо предоставить реализацию всех функций, определенных в классе.

Рассмотрим определение класса `Eq` как оно определено в стандартной библиотеке языка.

```
class Eq a where
    (==), (!=)   :: a -> a -> Bool
    x != y       =  not (x == y)
```

В этом определении вводится идентификатор класса `Eq` с одним параметром `a`. Далее в классе объявляются две бинарные операции с обозначениями `'=='` и `'!='`, типы которых содержат вхождения параметра `a`. Далее задается «реализация по умолчанию» для одной из этих операций – операции `'!='`. Это позволит в дальнейшем при определении реализации класса ограничиваться определением только одной операции – операции `'=='`, тогда в качестве реализации для второй операции будет выбрана «умолчательная» реализация.

Для того, чтобы определить реализацию класса `Eq`, нужно указать, что некоторый тип «принадлежит» классу `Eq`, и описать реализацию операций этого класса – в случае класса `Eq` это будет реализация операции `'=='`. Вот как можно, например, определить реализацию операций сравнения на равенство для типа `Bool`:

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

В этом определении идентификатор типа `Bool` занимает позицию фактического параметра класса `Eq`, тем самым объявляется, что тип `Bool` принадлежит классу `Eq`, а далее приведены уравнения, определяющие, что два логических значения равны только в том случае, когда они оба образованы конструкторами `True` или оба – `False`. Аналогичным образом можно было бы привести описания реализаций класса `Eq` и для всех остальных стандартных типов, хотя, конечно, на самом деле такое определение корректными средствами языка фактически можно выполнить только для типа `Bool`.

Описание реализации класса позволяет включить в этот класс новый тип. Например, мы можем включить в класс наш новый тип бинарных деревьев – `Tree`. Для этого нам будет нужно определить реализацию по крайней мере операции сравнения на равенство для объектов типа `Tree`. Мы будем считать равными деревья, имеющие одинаковую структуру, и содержащие в соответствующих узлах равные значения. Такое определение подразумевает, что над значениями узлов также можно выполнять операцию сравнения с помощью оператора `'=='`. Таким образом, мы должны задать определение реализации с контекстом, который накладывает ограничение на параметр типа `Tree`. Выглядеть такое определение будет следующим образом (листинг 1.30).

Листинг 1.30. Описание функции сравнения деревьев на равенство

```
instance Eq a => Eq (Tree a) where
```

```
    Null                    ==  Null                    = True
    (Tree node1 t11 t12)  ==  (Tree node2 t21 t22)  =
              (node1 == node2) && (t11 == t21) && (t12 == t22)
    _                       ==  _                       = False
```

Здесь в заголовке реализации класса `Eq` контекст используется для того, чтобы задать ограничение на параметр типа `Tree`. Операции класса `Eq` будут определены корректно для типа `Tree a`, если параметр `a`, в свою очередь, принадлежит классу `Eq`.

Ранее мы использовали еще один стандартный класс языка *Haskell* - класс `Ord`, задающий набор операций сравнения значений. Этот класс является *расширением* класса `Eq`, поскольку в него, наряду с "новыми" операциями `'<'`, `'>'`, `'<='` и `'>='` входят и операции класса `Eq`. Определение класса `Ord` в стандартной библиотеке выглядит следующим образом:

```
class Eq a => Ord a where
    (<), (>), (<=), (>=)  :: a -> a -> Bool
    x  <=  y  =  not (x > y)
    x  >=  y  =  not (x < y)
```

«Базовый» класс участвует в этом определении в виде контекста, описывающего ограничение на определяемый параметр класса. Приведенное определение класса `Ord` можно читать следующим образом: «при условии, что тип `a` принадлежит классу `Eq`, определение класса `Ord` для такого типа будет содержать набор операций `'<'`, `'>'`, `'<='` и `'>='` типа `a -> a -> Bool`, при этом имеют место следующие реализации операций `'<='` и `'>='` "по умолчанию"... »

Фактически мы и в самом деле, определяя новый класс `Ord`, расширили определение класса `Eq` новыми операциями, поскольку теперь мы должны будем, определяя реализацию класса `Ord` для некоторого конкретного типа данных, определять не только реализации для операций, описания которых приведены в этом классе, но и обеспечить, что этот тип принадлежит и классу `Eq`, то есть задать реализации и его операций тоже.

Следующим образом можно задать реализацию класса `Ord` для стандартного типа `Bool`.

```
instance Ord Bool where
    False < True  = True
    _     < _     = False

    True  > False = True
    _     > _     = False
```

Фактически такая реализация задает упорядоченность логических значений: значение `False` меньше `True`.

Заметим, что понятие класса в языке *Haskell* служит совершенно другим целям, чем аналогичное понятие в традиционных объектно-

ориентированных языках программирования. Некоторое сходство заключается лишь в том, что и в том и в другом случае класс используется как форма объединения в единое целое некоторого набора операций. Однако, если в традиционных объектно-ориентированных языках программирования класс определяет абстрактный или конкретный тип данных с фиксированным набором операций над значениями этого типа, то в языке *Haskell* понятие класса введено просто для того, чтобы можно было одни и те же имена функций и знаки операций использовать с аргументами различных типов, то есть для обеспечения полиморфизма функций. В традиционных языках это, скорее, называется перегрузкой имен функций и операций. Действительно, определение класса задает имена функций и их формальные типы (количество аргументов, возможно, типы некоторых аргументов и/или результата, некоторые соотношения между типами аргументов и т.п.). Это позволяет задавать разные реализации этих функций для аргументов различных типов в реализации класса. Однако это никак не ограничивает программиста в написании других функций, использующих аргументы упомянутых типов, и наоборот, различные типы данных могут «принадлежать» одному и тому же классу.

Заметим еще, что определение функций «по умолчанию», которые делаются непосредственно в определении классов, можно переопределять при описании реализации классов. Таким образом, определение функций «по умолчанию» служит лишь для сокращения текста реализации классов в случае, когда соответствующие реализации функций будут одинаковыми и естественными, однако, это никак не ограничивает программистов в изобретении специальных алгоритмов для реализации тех же операций для конкретных типов данных.

На этом мы заканчиваем наше краткое введение в язык *Haskell* и методы программирования на этом языке. Однако по-настоящему мы еще даже не начали раскрывать все возможности функционального программирования. Программы, которые мы писали до сих пор, не слишком отличаются от программ на традиционных языках программирования, разве что форма записи этих программ не совсем обычная, да вместо локальных переменных и циклов постоянно используются накапливающие аргументы функций и рекурсивные вызовы. В следующей главе мы рассмотрим гораздо более глубокие методы, применяющиеся в функциональном программировании, позволяющие писать необычайно короткие и очень выразительные программы, а также решать задачи, которые с трудом поддаются адекватному решению в традиционных языках.

Глава 2. Что еще есть в функциональном программировании

2.1. Концевая рекурсия

Часто приходится слышать следующее возражение против применения рекурсивных функций в программировании: они, мол, значительно снижают эффективность работы программы как в смысле времени, так и в смысле затрат памяти. Действительно, компьютеры, к которым мы привыкли, обычно не очень хорошо приспособлены для исполнения программ, содержащих большое количество рекурсивных вызовов. Это связано с тем, что обычно при каждом вызове функции или процедуры производятся довольно сложные действия, включающие отведение памяти под аргументы функции, установление нового контекста глобальных переменных, запоминание точки возврата и т.п. Все это приводит к тому, что несмотря на лучшие возможности для распараллеливания вычислений, на компьютерах традиционной архитектуры программы, написанные в функциональном стиле, показывают заметно меньшую производительность, чем программы, решающие те же задачи с помощью традиционных средств.

Это действительно так, и до тех пор, пока для исполнения функциональных программ будут использоваться компьютеры традиционной архитектуры, трудно ожидать, что область применения функционального программирования будет очень широкой. Правда, в таких областях как системы искусственного интеллекта, базы знаний, текстовая обработка и некоторых других функциональное программирование все же занимает свою нишу, предоставляя удобные средства программирования задач в этих областях, но все же во многих случаях придется обращать особое внимание на вопросы эффективности работы программ, для того, чтобы их можно было исполнять не только на машинах со специализированной архитектурой, но и на традиционных компьютерах.

В некоторых случаях рекурсивные вызовы процедур могут быть эффективно реализованы с помощью обычных циклических вычислений, которые очень эффективно исполняются на традиционных компьютерах. Рекурсивный вызов легко можно заменить на циклическое вычисление, если в теле функции рекурсивное обращение к ней является последним исполняемым вызовом. Рассмотрим, например, два определения функции для вычисления факториала натурального числа. Первое из них, представленное ниже функцией `fact`, уже рассматривалось нами в первой главе.

```
fact :: Integer -> Integer
fact 0 = 1
```

```
fact n = n * fact (n-1)
```

В этом определении рекурсивный вызов в правой части уравнения, определяющего работу функции, не является последним вызовом: после него происходит еще обращение к операции умножения. Однако, рассмотрим другое определение этой функции, в которой вычисления производятся с помощью другой (вспомогательной) функции `fact'`. В этой вспомогательной функции вместо одного аргумента используются два, дополнительный накапливающий аргумент добавлен таким образом, чтобы значение факториала первого аргумента, умноженное на второй аргумент, давало бы в результате искомое значение (листинг 2.1).

Листинг 2.1. Вычисление факториала числа с помощью концевой рекурсии

```
fact  :: Integer -> Integer
fact' :: Integer -> Integer -> Integer
fact' 0 f = f
fact' k f = fact' (k-1) (k*f)
fact  n   = fact' n 1
```

Нетрудно убедиться в том, что приведенное определение функции корректно. Однако, в отличие от предыдущего варианта, рекурсивный вызов в определении функции `fact'` является последним вызовом, исполняющимся в правой части уравнения. Такая рекурсия носит название *концевой* рекурсии. Компилятор, анализирующий программу, может обнаружить все случаи использования концевой рекурсии и сгенерировать для таких функций более быструю программу, поскольку концевая рекурсия может быть легко представлена в виде циклического вычисления. Действительно, рассмотрим процесс вычисления значения `fact 4` согласно первому определению функции. Последовательность выражений, получающаяся в результате подстановок правых частей уравнений, определяющих поведение функции `fact`, будет выглядеть так:

```
fact 4                          ⇒
4 * fact 3                      ⇒
4 * (3 * fact 2)                ⇒
4 * (3 * (2 * fact1))           ⇒
4 * (3 * (2 * (1 * fact 0)))    ⇒
4 * (3 * (2 * (1 * 1)))         ⇒
4 * (3 * (2 * 1))               ⇒
4 * (3 * 2)                     ⇒
4 * 6                           ⇒
24
```

Видно, как при этом вычислении сначала в результате последовательности рекурсивных вызовов функции выстраивается длинное выражение, в котором запоминаются все "отложенные" операции умножения, а потом эти умножения выполняются, формируя

окончательный результат. При втором способе определения функции последовательность вычислений будет выглядеть совершенно по-другому:

```
fact 4                  ⇒
fact' 4 1               ⇒
fact' (4-1) (4*1)       ⇒
fact' 3 4               ⇒
fact' (3-1) (3*4)       ⇒
fact' 2 12              ⇒
fact' (2-1) (2*12)      ⇒
fact' 1 24              ⇒
fact' (1-1) (1*24)      ⇒
fact' 0 24              ⇒
24
```

Теперь уже умножения не "откладываются" до момента окончания работы рекурсии, а исполняются непосредственно во время работы рекурсии. Вызов функции `fact'` все время находится на самом внешнем уровне выражения, откуда и видно, что фактически вычисления можно оформить в виде цикла с двумя переменными, роль которых исполняют меняющиеся значения аргументов функции. Заметим, кстати, что приведенное определение функции с использованием накапливающего параметра очень близко к стандартному определению функции в традиционном языке программирования с использованием цикла.

Многие из функций, рассматривавшихся нами ранее в первой главе, построены с помощью концевой рекурсии. Таковы, например, функции для определения того, является ли заданное натуральное число простым, функция для вычисления числа *e* суммированием членов ряда Тейлора, функция «переворачивания» списка и другие. Однако, большинство приводимых нами рекурсивных функций использовали рекурсию в самом общем виде, мы не заботились о том, чтобы рекурсия была концевой. На самом деле любую рекурсивную функцию можно преобразовать таким образом, чтобы все рекурсивные вызовы в ней были концевыми. В некоторых случаях такое преобразование делается очень просто, в других случаях требуется глубокий анализ функции. Практически всегда для приведения рекурсии к концевой форме приходится определять вспомогательную функцию с дополнительными аргументами. Приведем один пример.

Функция сортировки списка методом «простых вставок», которую мы описывали в главе 1, использовала рекурсию общего вида в функциях `simpleSort` и `insert`. Напомним определение этих функций (листинг 2.2):

Листинг 2.2. Сортировка списка методом «простых вставок»

```
simpleSort      :: (Ord a) => [a] -> [a]
```

```
insert          :: (Ord a) => a -> [a] -> [a]

simpleSort []       = []
simpleSort (x:s)    = insert x (simpleSort s)

insert elem []  = [elem]
insert elem list@(x:s) | elem < x   = elem:list
                       | otherwise  = x:(insert elem s)
```

Определение функции `simpleSort` очень легко изменить так, чтобы использовалась только концевая рекурсия. Для этого достаточно ввести дополнительный аргумент функции, представляющий уже отсортированную часть списка. Определение нового варианта функции приведено в листинге 2.3:

Листинг 2.3. Вариант функции simpleSort с концевой рекурсией

```
simpleSort      :: (Ord a) => [a] -> [a]
simpleSort'     :: (Ord a) => [a] -> [a] -> [a]

-- первый аргумент - отсортированная часть списка,
-- второй – неотсортированный остаток.
simpleSort  s       = simpleSort' [] s
simpleSort' s []    = s
simpleSort' s (x:u) = simpleSort' (insert x s) u
```

Функцию `insert`, которая вставляет элемент на свое место в упорядоченный список, превратить в функцию, содержащую только концевую рекурсию, несколько сложнее. Первое, что приходит в голову – это также ввести один дополнительный аргумент, содержащий уже просмотренную часть списка, в которой находятся элементы, меньшие вставляемого. Тогда определение этой функции вместе с новой вспомогательной функцией `insert'` будет выглядеть следующим образом (листинг 2.4):

Листинг 2.4. Вариант функции insert с концевой рекурсией

```
insert          :: (Ord a) => a -> [a] -> [a]
insert'         :: (Ord a) => [a] -> a -> [a] -> [a]

insert  e s          = insert' [] e s
insert' low e []     = low ++ [e]
insert' low e high@(x:s) | e < x     = low ++ (e:high)
                         | otherwise = insert' (low++[x]) e s
```

Очевидно, однако, что теперь, пытаясь повысить эффективность работы программы путем превращения всех рекурсивных вызовов в концевые, мы при программировании функции `insert` добились, скорее, обратного эффекта, поскольку использование операции соединения списков `(++)` в рекурсивном вызове функции `insert'` приводит к значительному замедлению работы. Несколько более быстрый вариант

функции получается, если вместо добавления элементов в конец списка использовать присоединение элемента к голове списка. Правда, в этом случае список получается «перевернутым», так что для восстановления порядка элементов придется его еще раз "переворачивать" с помощью функции `reverse`, которая, кстати, была нами запрограммирована с использованием концевой рекурсии (листинг 2.5).

Листинг 2.5. Еще один вариант функции insert с концевой рекурсией

```
insert          :: (Ord a) => a -> [a] -> [a]
insert'         :: (Ord a) => [a] -> a -> [a] -> [a]

insert  e s           = reverse (insert' [] e s)
insert' low e []      = e:low
insert' low e (x:high) | e < x     = insert' (e:low) x high
                       | otherwise = insert' (x:low) e high
```

Этот вариант несколько лучше предыдущего, однако все же остается хуже первоначального варианта, не использующего концевую рекурсию. Выгоды от использования концевой рекурсии в данном случае не смогут перевесить потерь от ухудшения алгоритма работы. Все-таки можно добиться того, чтобы сделать рекурсию концевой и в этом случае, не жертвуя при этом эффективностью работы алгоритма, однако, пожалуй, для функции `insert` лучше будет оставить первоначальный вариант определения, а вот функция `simpleSort` ничего не потеряла от превращения ее рекурсивного вызова в вариант с концевой рекурсией, поэтому если используемый вами компилятор может получить выгоду от концевой рекурсии, то следует сделать описанное преобразование для функции `simpleSort`, но, возможно, оставить без изменения функцию `insert`.

Рассмотренный пример демонстрирует технологию оптимизации программ с заменой простой рекурсии на концевую. Однако, как видно, такая оптимизация не всегда удается легко и иногда фактически приводит к замедлению скорости работы программы, так что следует выполнять эту оптимизацию с осторожностью.

В следующем разделе мы рассмотрим еще один способ преобразования функций, при котором от рекурсии избавляются вообще, перенося ее в определение других общих функций, которые можно запрограммировать очень эффективно уже на уровне реализации языка.

2.2. Функции высших порядков. Карринг

В этом разделе мы рассмотрим способы описания и примеры применения функций, аргументами или результатами которых также являются функции. В традиционных языках программирования такие функции часто бывает невозможно описать (см., например, попытку описания функции суперпозиции, которую мы предприняли в начале

первой главы). В функциональном программировании, однако, функции являются значениями «первого класса», так что подобные описания не только возможны, но и достаточно часто встречаются. В частности, функция, определяющая суперпозицию двух других функций, не только может быть легко описана средствами языка, но и является одним из самым важных инструментов построения программ, и поэтому содержится в языке в виде стандартной операции.

Мы, однако, начнем не с описания функции суперпозиции, а с других примеров, в которых функции используются в качестве аргументов других функций. В качестве первого примера рассмотрим стандартную функцию обработки списков `map`, которая, получив в качестве аргументов список и функцию преобразования элементов этого списка, выдает в качестве результата список из преобразованных элементов, полученных применением функционального параметра последовательно к каждому элементу списка. Если, например, функция `sqr` возводит в квадрат целое число, то с помощью функции `map` можно, имея список целых чисел, получить список их квадратов:

```
map sqr [1, 2, 5, -2]  ⇒  [1, 4, 25, 4]
```

Функция `map` – это стандартная функция, определенная в ядре языка. Однако, по уже сложившейся традиции приведем ее описание заново. Прежде всего, давайте определим ее тип. Первым аргументом нашей функции является функция, которая в нашем случае имеет один целый аргумент и возвращает целый результат. Тип этого первого аргумента будет `Integer->Integer`. Второй аргумент и результат функции – это списки целых типа `[Integer]`. Таким образом тип функции `map` будет выглядеть так:

```
map :: (Integer -> Integer) -> [Integer] -> [Integer]
```

На самом деле не столь важно, что исходный и результирующий списки содержат именно элементы типа `Integer`. Фактически исходные элементы могут иметь произвольный тип `a`. Тогда, если функция, являющаяся первым аргументом функции `map`, имеет тип `(a -> b)`, то результирующий список будет содержать элементы типа `b`. Теперь мы готовы описать полностью функцию `map`, включая и ее тип (листинг 2.6).

Листинг 2.6. Определение функции отображения списков – map

```
map             :: (a -> b) -> [a] -> [b]
-- результат применения map к пустому списку - пустой список
map _    []     = []
map func (x:s)  = (func x):(map func s)
```

Первое уравнение определяет результат работы отображения в случае пустого списка. Понятно, что в этом случае первый аргумент функции не используется вовсе, так как результатом будет пустой список

независимо от того, какая функция применяется для отображения его элементов. Второе уравнение использует рекурсивное обращение к функции `map` для того, чтобы построить отображение хвоста списка, а потом присоединяет к результату головной элемент, который получается с помощью применения к первому элементу исходного списка отображающей функции.

Функция, подобная `map`, которая в качестве аргумента получает другую функцию или выдает функцию в качестве результата, называется функцией *высшего порядка* или *функционалом*.

Если некоторая функция предназначена только для того, чтобы передать ее в качестве аргумента другой функции, то не имеет смысла давать ей постоянное имя и определять ее тип явно. В этом случае можно использовать так называемое *лямбда-выражение*, которое задает образцы для аргументов функции и правые части ее уравнений, но не связывает с этой функцией никакого имени, а тип такой функции определяется из контекста. Таким образом лямбда-выражение представляет собой функциональное значение, изображающее безымянную функцию.

В простейшем случае, когда функция может быть определена с помощью только одного уравнения, лямбда-выражение имеет точно такой же вид, как и это единственное уравнение, только в левой его части вместо имени функции стоит символ обратной косой черты (изображающий греческую букву «лямбда» – λ), а вместо знака равенства для отделения образцов для аргументов от правой части уравнения используется последовательность символов `'->'`. Например, если функция `sqr` для возведения числа в квадрат не определена, то мы можем написать лямбда-выражение для ее определения непосредственно там, где эта функция используется. Тогда вышеуказанный вызов функции `map` для получения списка квадратов будет выглядеть следующим образом:

```
map (\x -> x * x) [1, 2, 5, -2]
```

Если уравнений несколько, то их можно объединить в одно выражение либо с помощью условного выражения `if`, либо с помощью специальной конструкции для выбора по образцам – `case`. Так, например, для того, чтобы получить список из знаков заданного списка целых чисел с помощью функции `map` (знак числа – это функция, выдающая ноль, единицу или минус единицу в зависимости от того, является ли число нулевым, положительным или отрицательным), можно поступить следующими способами. Во-первых, можно определить отдельно функцию для вычисления знака числа, скажем, следующим образом:

```
sign 0 = 0
sign n | n < 0 = -1
       | otherwise = 1
```

Тогда вызов для нахождения списка знаков заданного списка чисел выглядел бы так:

```
map sign [2,5,0,-3,2,-2]
```
(в результате получится список `[1, 1, 0, -1, 1, -1]`)

Второй способ вызова, при котором функция вычисления знака числа определяется прямо в точке вызова, будет выглядеть так (вместо трех уравнений использовано условное выражение):

```
map (\n -> if n == 0 then 0 else if n < 0 then -1 else 1)
                                                [2,5,0,-3,2,-2]
```

Наконец, определение функции лямбда-выражением с использованием конструкции `case` будет выглядеть следующим образом:

```
map (\n -> case n of {0 -> 0; n | n < 0 -> -1; n -> 1})
                                                [2,5,0,-3,2,-2]
```

Мы не будем углубляться в тонкости синтаксиса для всестороннего изучения конструкции `case`, тем более, что она нам в дальнейшем почти и не понадобится. Однако заметим, что привычное определение именованной функции всегда может быть переформулировано с помощью лямбда-выражения. Так, приведенное выше определение функции `sign` на самом деле является лишь другой синтаксической формой для определения с помощью лямбда-выражения:

```
sign = \n -> case n of {0 -> 0; n | n < 0 -> -1; n -> 1}
```

Приведем и еще одну форму записи, в которой именованная функция определяется в выражении локально с помощью лямбда-выражения и *блока* `where`. В следующей строчке вычисляются факториалы первых десяти натуральных чисел.

```
map fact [1..10] where
         fact = \n -> if n == 0 then 1 else n * fact (n-1)
```

Заметим, что здесь в лямбда-выражении использован рекурсивный вызов. В данном случае это возможно, поскольку в предложении `where` с ним связывается имя.

Вообще, блок `where` может содержать и несколько определений функций, а каждая из этих функций может иметь и несколько уравнений и даже определение типа. Таким образом, локальное определение функций может быть сделано в точности в той же самой форме, что и глобальное:

```
map fact [1..10] where
         fact :: Integer -> Integer
         fact 0 = 1
         fact n | n > 0 = n * fact (n-1)
```

Но вернемся к функциям высших порядков. Функция `map` получает в качестве аргумента функцию одного аргумента и применяет ее к элементам списка, в результате чего получается новый список. Можно рассмотреть еще одну функцию высшего порядка, которая в качестве аргумента будет получать бинарную операцию – функцию двух аргументов. Такую бинарную операцию можно последовательно

применять к элементам списка так, чтобы в результате обработки всего списка получилось бы одно значение. Например, если бинарную операцию сложения применять последовательно ко всем элементам списка (в качестве начального значения можно взять ноль), то в результате получится сумма всех элементов списка. Если же в качестве бинарной операции взять операцию умножения, то в результате перемножения всех элементов списка получится произведение этих элементов (теперь в качестве начального значения нужно будет взять единицу).

В стандартной библиотеке языка *Haskell* имеются две подобные функции, одна из которых применяет заданную бинарную операцию к элементам списка от начала списка к его концу, а вторая начинает обработку элементов с конца списка, применяя заданную в качестве аргумента операцию последовательно, двигаясь по направлению к началу списка. Первая из этих операций называется `foldl`, а вторая – `foldr`. Общее название для этих операций – операции свертки списка. Приведем определения этих функций. В этих определениях (листинг 2.7) `f` – применяемая бинарная операция (или функция с двумя аргументами), `seed` – начальное значение (оно будет результатом работы функции свертки, если обрабатываемый список не содержит элементов).

Листинг 2.7. Определение функций свертки списков – foldl и foldr

```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl _ seed []    = seed
foldl f seed (x:s) = foldl f (f seed x) s

foldr :: (a -> b -> b) -> b -> [a] -> b
foldr _ seed []    = seed
foldr f seed (x:s) = f x (foldr f seed s)
```

Рассмотрим уравнения, определяющие эти функции подробнее (ограничимся только функцией `foldr`, уравнения для функции `foldl` совершенно аналогичны). Как и в случае функции `map`, первое уравнение определяет поведение функции свертки для случая, когда исходный список пуст. Как мы уже говорили, в этом случае результатом должно быть начальное значение, определяемое аргументом `seed`. Если же исходный список не пуст, то, прежде всего, построим свертку хвоста списка с помощью рекурсивного обращения к функции `foldr`, а затем применим нашу бинарную операцию `f` к первому элементу списка и результату свертки хвоста. В результате и получится искомый результат.

Сумму всех элементов заданного списка `list` теперь можно получить простым вызовом любой из функций `foldl` или `foldr` при условии, что в качестве начального значения выбирается ноль, а в качестве бинарной операции берется операция сложения:

```
foldl (+) 0 list
```

однако, функция свертки – это довольно мощная функция, которая может применяться в самых разных, порою, довольно неожиданных ситуациях. Так, например, если описать операцию присоединения элемента к началу списка в виде операции `addElem`

```
addElem list elem = elem:list
```

(заметим, что эта операция отличается от стандартного конструктора списков (:) только порядком аргументов), то функция «переворачивания» списка, которую мы описывали в первой главе в виде функции `reverse`, теперь может быть выражена с помощью применения функции `foldl`:

```
reverse list = foldl addElem [] list
```

В приведенной версии функции обращения списка нет явной рекурсии. Рекурсивные вызовы «спрятаны» внутри функции `foldl`. Вообще функции высших порядков часто позволяют писать короткие и выразительные программы, не содержащие в явном виде рекурсии. Вот, например, версия функции вычисления факториала, в которой для вычисления сначала строится список целых чисел от единицы до заданного значения аргумента, а потом все элементы полученного списка перемножаются для получения результата.

```
factorial n = foldr (*) 1 [1..n]
```

Функции высших порядков можно определять, чтобы строить новые функции на основе других. Наверное, одной из самых распространенных операций над функциями является их суперпозиция (или композиция). Такая операция позволяет по двум функциям `f` и `g` получить новую функцию `fg` такую, что результат ее применения к аргументу `fg x` будет тем же самым, что и результат последовательного применения функций `f` и `g`: `f (g x)`. В отличие от традиционных языков программирования, в которых описать подобную функцию, как правило, невозможно, в языке *Haskell* такое описание не составляет никакого труда:

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

Такая функция имеется в стандартной библиотеке языка *Haskell* в виде бинарной операции `(.)`. Это означает, что если имеется функция возведения числового значения в квадрат `sqr`, то функция возведения числа в четвертую степень `power4` может быть получена с помощью вызова

```
power4 = sqr . sqr
```

Выше при описании варианта определения функции обращения списка мы использовали определение функции `addElem`, которая отличается от стандартного конструктора списков `(:)` только порядком аргументов. Можно определить функцию, которая по заданной функции двух аргументов будет получать новую функцию, отличающуюся от

заданной только порядком аргументов. Такая функция также имеется в стандартной библиотеке языка *Haskell*, называется `flip` и может быть описана следующим образом:

```
flip :: (a -> b -> c) -> (b -> a -> c)
flip func = \x y -> func y x
```

Теперь функцию обращения списка можно определить, не используя явного описания вспомогательной функции `addElem`:

```
reverse list = foldl (flip (:)) [] list
```

Заметим, что описанный алгоритм обращения списка работает очень эффективно и обращает список за линейное время. Ранее такого результата мы добивались, используя преобразование функций с помощью введения дополнительных аргументов. Обратите внимание еще и на то, что функции `flip` передается аргументом конструктор списков `(:)`, который в данном случае использован в качестве бинарной операции. Это как раз тот случай, когда конструктор используется в роли обычной функции.

Иногда функцию свертки называют также операцией *вставки*, поскольку результат ее работы аналогичен вставке бинарной операции, которая задается в качестве первого аргумента, между элементами списка.

Функции, аналогичные функциям отображения и свертки списков, можно определять и для структур данных, определенных программистом. Заметим, что как свертка, так и отображение списка осуществляют перебор элементов списка (итерацию). Для функции `map` порядок прохождения элементов списка совершенно не важен, поскольку результат работы функции от этого не зависит. Для линейных списков два варианта функции свертки `foldl` и `foldr` соответствовали двум различным способам прохождения списков – в прямом и обратном направлении. Правда, для сложения или перемножения элементов списка порядок прохождения элементов тоже не важен, однако, в общем случае, эти два способа свертки списка дают разный результат (даже типы двух функций – `foldl` и `foldr` – разные).

Для двоичного дерева, определенного нами в главе 1, можно предложить несколько вариантов его прохождения; соответственно можно будет написать несколько вариантов функции свертки дерева согласно алгоритмам его прохождения. Например, если выбрать правосторонний порядок прохождения узлов, при котором любой узел проходится до прохождения узлов его левого поддерева и после прохождения узлов правого поддерева (упорядоченное дерево будет в этом случае проходиться в порядке убывания значений узлов), то функция свертки дерева `foldTree` может быть написана так, как представлено в листинге 2.8.

Поскольку, в отличие от списка, где в каждом узле имелась единственная ссылка на следующий элемент списка, в каждом узле дерева

имеются ссылки на два поддерева, то и в определении функции свертки вместо одного рекурсивного обращения к функции будут два рекурсивных обращения – соответственно для свертки отдельно левого и правого поддеревьев. Это, однако, не приводит к снижению эффективности работы. В любом случае каждый узел дерева просматривается ровно один раз.

Конечно, имея такую функцию, можно легко вычислить сумму значений узлов, хранящихся в дереве, или определить максимальное значение узла. Несколько неожиданным оказывается, что с помощью свертки можно легко определить функцию разглаживания дерева, которую мы определяли в главе 1. Действительно, если сворачивать двоичное дерево с помощью стандартного конструктора списков `(:)`, то в результате действительно получится список всех узлов дерева в порядке левостороннего обхода (листинг 2.8).

Листинг 2.8. Свертка и разглаживание дерева

```
foldTree :: (a -> b -> b) -> b -> Tree a -> b
foldTree _ seed Null           = seed
foldTree f seed (Tree a tl tr) =
                      foldTree f (f a (foldTree f seed tr)) tl
flatten   :: Tree a -> [a]
flatten t = foldTree (:) [] t
```

Этот вариант функции разглаживания дерева не только очень короткий, но и весьма эффективный. Функция свертки дерева просматривает все его узлы только один раз, а применяемая к узлам дерева функция – конструктор списков – имеет постоянное (и очень небольшое) время работы.

Приведенный вариант свертки дерева обходит дерево справа налево, однако, нетрудно определить и другие способы прохождения дерева. Приведем, например, еще два варианта обхода узлов. Первый вариант, представленный в листинге 2.9 функцией `foldTreeLR`, обходит дерево в порядке, противоположном только что описанному, то есть проходит узлы в левостороннем порядке. Для дерева поиска этот порядок соответствует перебору узлов в возрастающем порядке. Второй вариант обхода и свертки представлен функцией `foldTreeUpDown` и соответствует прохождению узлов «сверху вниз», при котором сначала проходится корень дерева, а после этого обходятся его поддеревья, разумеется, также в нисходящем порядке.

Листинг 2.9. Некоторые варианты свертки дерева

```
-- прохождение и свертка дерева в левостороннем порядке
foldTreeLR :: (a -> b -> b) -> b -> Tree a -> b
foldTreeLR _ seed Null           = seed
foldTreeLR f seed (Tree a tl tr) =
          foldTreeLR f (f a (foldTreeLR f seed tl)) tr
```

```
-- прохождение и свертка дерева в нисходящем порядке
foldTreeUpDown :: (a -> b -> b) -> b -> Tree a -> b
foldTreeUpDown _ seed Null           = seed
foldTreeUpDown f seed (Tree a tl tr) =
          foldTreeUpDown f (foldTreeUpDown f (f a seed) tl) tr
```

Часто бывает нужно определить функцию, алгоритм работы которой совпадает с алгоритмом уже имеющейся функции, но при этом уменьшить количество ее аргументов, зафиксировав значение одного или нескольких аргументов функции. Например, если мы хотим удвоить значения всех числовых элементов некоторого списка, то мы можем использовать для этого функцию `map` приблизительно следующим образом:

```
map double [1,5,12,-2]
```

где `double` – функция удвоения числа. Однако, функция удвоения – это просто функция умножения с фиксированным значением одного из параметров – значением `2`. Этот факт можно выразить, заменив идентификатор `double` на явно выписанное лямбда-выражение:

```
map (\x -> 2 * x) [1,5,12,-2]
```

В данном случае, однако, можно записать этот функциональный параметр и еще проще:

```
map (2 *) [1,5,12,-2]
```

образовав так называемое *сечение* - частично параметризованный вызов операции умножения. Вообще, функции в языке *Haskell* обладают той приятной особенностью, что всегда можно написать *частично параметризованный* вызов функции, указав вместо всех необходимых аргументов лишь первые несколько (разумеется, если функция, вообще говоря, имеет более одного аргумента). Например, саму функцию удвоения всех элементов списка можно описать как при помощи уравнения

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

так и просто выполнив частичную параметризацию функции `map`:

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

Можно сказать еще и по-другому: если функция, имеющая, вообще говоря, несколько аргументов, вызвана только с одним аргументом, то в результате такого вызова получается функция, у которой будет на один аргумент меньше, а эффект ее выполнения будет таким же, как и у исходной функции с уже заданным значением первого аргумента. Иногда говорят, что в языке *Haskell* любая функция – это функция с одним аргументом. Действительно, пусть мы имеем определение функции, скажем, с тремя аргументами. Тогда следующие уравнения, определяющие алгоритм работы этой функции будут равноправны:

```
myFunc x y z = x * y + z
myFunc = \x y z -> x * y + z
myFunc = \x -> \y -> \z -> x * y + z
myFunc x = \y z -> x * y + z
```

Наоборот, часто удобно определять функцию, задавая для нее «лишний» аргумент. Например, определяя функцию, которая по двум заданным функциям строит их суперпозицию, мы выписали следующее уравнение

```
comp f g = \x -> f (g x)
```

однако вместо этого мы могли бы написать более простое эквивалентное ему уравнение:

```
comp f g x = f (g x)
```

Соответственно, тип функции comp мог бы быть определен двумя эквивалентными способами

```
comp :: (b -> a) -> (c -> b) -> (c -> a)
```

или

```
comp :: (b -> a) -> (c -> b) -> c -> a
```

Другими словами, при определении типа функции знак `->`, отделяющий типы аргументов друг от друга и от типа результата, можно считать «право-ассоциативной» операцией, подразумевающей расстановку скобок справа налево. Это, кстати, объясняет и тот непривычный факт, почему один и тот же символ используется как для разделения типов аргументов, так и для отделения типов аргументов от типа результата функции: в действительности, как мы уже говорили, любая функция имеет только один аргумент, однако результатом может быть функция, которая будет также иметь один аргумент, и т.д.

Свойство функции `n` аргументов, благодаря которому можно вызывать ее с произвольным числом аргументов, меньшим или равным `n`, называется *каррингом* в честь математика Хаскелла Карри (Haskell Curry), предложившего такую концепцию. Функции, обладающие свойством карринга, называют карринговыми. Иногда, в противоположность карринговым функциям, удобно считать, что функция должна получать не два аргумента, а один аргумент, представляющий пару значений. Например, описывая функцию для нахождения наибольшего общего делителя двух натуральных чисел, мы считали, что функция будет иметь два аргумента типа `Integer`. Соответствующие описание типа и уравнения выглядели следующим образом:

```
gcd :: Integer -> Integer -> Integer
gcd m n | m < 0 = error "gcd: Wrong argument"
        | n < 0 = error "gcd: Wrong argument"
gcd m 0 = m
gcd m n = gcd n (m `mod` n)
```

Можно, однако, определить функцию и несколько по-другому, считая, что единственным аргументом функции должна быть пара целых чисел:

```
gcd :: (Integer, Integer) -> Integer
```

```
gcd (m, n) | m < 0 = error "gcd: Wrong argument"
           | n < 0 = error "gcd: Wrong argument"
gcd (m, 0) = m
gcd (m, n) = gcd (n, (m `mod` n))
```

Это определение функции уже не обладает свойством карринга; невозможно зафиксировать значение одного аргумента, превратив эту функцию в функцию с одним аргументом – целым числом. Однако, различие между этими двумя определениями чисто синтаксические, и можно легко перейти от одного определения к другому. В стандартной библиотеке языка *Haskell* имеются функции, позволяющие переходить от функции в «каррингговой» форме к функции, аргументом которой является кортеж из двух значений и наоборот. Мы приведем определения обеих функций (листинг 2.10).

Листинг 2.10. Каррирование и декаррирование функций

```
-- функция "каррирования"
curry   :: ((a, b) -> c) -> (a -> b -> c)
curry   f a b     = f (a, b)
-- функция "декаррирования"
uncurry :: (a -> b -> c) -> ((a, b) -> c)
uncurry f (a, b)  = f a b
```

Сама простота этих определений показывает всю силу и выразительность языка *Haskell* при работе с функциями. Может показаться, что в листинге 2.10 уравнения для функций `curry` и `uncurry` перепутаны, что аргументы для функции `curry` записаны «в каррингговой форме», а аргументы функции `uncurry` – наоборот, «в некаррингговой форме». На самом деле никакой ошибки нет. В этом можно убедиться, если понять, что функциональный аргумент в уравнении для функции `curry` – это «некаррингговая» функция, так что в правой части уравнения она совершенно корректно применяется к кортежу из двух аргументов, как и положено некаррингговой функции. В левой же части уравнения представлена на самом деле форма обращения к результату работы функции `curry`. Для лучшего понимания можно также переписать уравнение для функции `curry` в эквивалентном виде с использованием лямбда-выражения:

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

Здесь очевидно, что результатом применения функции `curry` является функция двух аргументов в каррингговой форме, которая применяет некаррингговую функцию `f` к кортежу, составленному из двух своих аргументов. Анализ функции `uncurry` можно провести совершенно аналогично.

Важной стандартной функцией высших порядков является также функция фильтрации списка – `filter`. Она часто используется для того, чтобы по заданному списку элементов получить список из тех его

элементов, которые удовлетворяют заданному условию. Если элементы списка имеют некоторый тип `a`, то условие на элементы естественно задавать с помощью характеристической функции типа `a -> Bool`. Таким образом, тип функции фильтрации можно записать в виде

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

а саму эту функцию можно определить с помощью уравнений

```
filter f [] = []
filter f (x:s) | f x = x : (filter f s)
               | otherwise = filter f s
```

Функция `filter` часто также используется в виде специальной синтаксической конструкции

```
[ x | x <- s, f x ]
```

которую можно прочитать следующим образом: список таких элементов `x`, которые принадлежат списку `s` и при этом удовлетворяют условию, наложенному функцией `f` (то есть для которых выражение `f x` истинно).

На самом деле представленная синтаксическая конструкция, представленная выше (будем называть ее обобщенным конструктором списков), более общая. С помощью нее можно легко построить довольно сложные списки. В общем виде она выглядит следующим образом:

```
[ элемент | источники,... условия,... ]
```

где *элементц* – это произвольное выражения для построения элементов нового списка из образцов, заданных в источниках, *источники* – это перечисленные через запятую конструкции вида

```
образец <- список
```

а *условия* – это перечисленные через запятую логические выражения, определяющие условия на выбираемые элементы. Например, обобщенный конструктор

```
[ (x,y) | x <- [1..10], y <- [1..10] ]
```

представляет список всех пар чисел с элементами из списка от 1 до 10. «Таблица умножения» – список списков произведений чисел от 1 до 10 на самих себя – может быть построена с помощью выражения

```
[ map (* x) [1..10] | x <- [1..10] ]
```

а если функция `isPrime` проверяет, является ли ее аргумент простым числом, то список всех простых чисел от 1 до 100 можно построить с помощью выражения

```
[ x | x <- [1..100], isPrime x ]
```

(конечно, в этом последнем случае проще применить функцию фильтрации списка `filter`)

```
filter isPrime [1..100]
```

Функции высших порядков могут использоваться в программировании на языках функционального программирования в

качестве адекватной формы представления данных. Например, одной из форм представления множества является его характеристическая функция. Характеристической функцией множества, состоящего из элементов некоторого типа `T`, будет логическая функция, определенная на значениях типа `T` и выдающая значение `True`, если значение аргумента принадлежит этому множеству, и `False` в противном случае. Несмотря на то, что такое представление не позволяет в явном виде быстро находить элементы множества, но зато оно имеет ряд других преимуществ. Прежде всего, представление одинаково хорошо приспособлено для представления как конечных, так и бесконечных множеств. Например, конечное множество целых чисел, состоящее из всех чисел из диапазона `1..10` может быть адекватно представлено функцией `\n-> (n>=1) && (n<=10)`, а бесконечное множество всех четных чисел с такой же (едва ли не большей) легкостью может быть представлено функцией `\n -> (n `mod` 2 == 0)`.

Представление множеств в виде характеристических функций позволяет легко выполнять такие обычные теоретико-множественные операции над множествами как объединение, пересечение, добавление или удаление отдельных элементов в множество, дополнение до универсума и др. Приведем определения этих функций (листинг 2.11).

Листинг 2.11. Представление множеств характеристическими функциями

```
-- Тип Set задает представление множеств в виде функций
type Set a          = a -> Bool

--  Функция объединения множеств
union               :: Set a -> Set a -> Set a
union s1 s2 e       = (s1 e) || (s2 e)

--  Функция пересечения множеств
intersect           :: Set a -> Set a -> Set a
intersect s1 s2 e   = (s1 e) && (s2 e)

--  Разность множеств
difference          :: Set a -> Set a -> Set a
difference s1 s2 e  = (s1 e) && not (s2 e)

--  Дополнение до универсума
complement          :: Set a -> Set a
complement s1 e     = not (s1 e)

--  Добавление элемента
addElem             :: Eq a => Set a -> a -> Set a
addElem set elem e  = (elem == e) || (set e)

--  Удаление элемента
```

```
remElem              :: Eq a => Set a -> a -> Set a
remElem set elem e  = (elem /= e) && (set e)
```

При определении уравнений для этих функций мы воспользовались тем, что в левой части уравнения можно задавать столько аргументов, сколько удобно. Например, формально функция объединения множеств `union` имеет два аргумента типа множества и результат типа множества. Соответственно, уравнение, определяющее поведение этой функции, надо было бы задать в виде

```
union s1 s2 = \e -> (s1 e) || (s2 e)
```

однако, поскольку в правой части этого уравнения находится лямбда-выражение, мы можем просто перенести аргумент этого лябмда-выражения в левую часть уравнения. Уравнение, которое при этом получается, можно прочитать так: «объединение множеств `s1` и `s2` содержит только такие элементы `e`, что `e` содержится в `s1` или `e` содержится в `s2`.»

Помимо описанных операций над множествами, представленными своими характеристическими функциями, можно выполнять и некоторые другие операции. Давайте, например, построим множество простых чисел от `2` до некоторого заданного целого значения, используя известный алгоритм *решета Эратосфена*. Для этого нам потребуется исключать из заданного множества все числа, кратные заданному числу. Соответствующая фильтрующая функция может выглядеть следующим образом:

```
filt :: Set Integer -> Integer -> Set Integer
filt set number elem = (set elem) && (elem `mod` number /= 0)
```

то есть в отфильтрованном множестве будут все элементы исходного множества, кроме тех, которые при делении на заданное число дают в остатке ноль.

Теперь искомое множество простых чисел можно получить, если последовательно исключать из заданного с самого начала множества чисел, больших или равных `2`, элементы, делящиеся на `2`, затем на `3`, и т.д. В результате получится следующая программа (листинг 2.12).

Листинг 2.12. Алгоритм «решета Эратосфена»

```
-- Основная функция primeTo вычисляет множество простых чисел
-- из диапазона от 2 до заданного значения
primesTo :: Integer -> Set Integer

-- Вспомогательная функция реализует алгоритм решета
-- Эратосфена для нахождения множества простых чисел.
-- Аргументы:
--  1) нижняя граница множества чисел, исследуемых на простоту
--  2) множество, содержащие только простые числа,
--     меньшие заданной границы
```

```
--  3) верхняя граница множества
erato :: Integer -> Set Integer -> Integer -> Set Integer

-- За исходное множество принимаем множество чисел 2..max
primesTo max = erato 2 (\n -> (n >= 2) && (n <= max)) max
erato n set max
      | n * n > max = set  -- множество уже отфильтровано
      | set n       = addElem (erato (n+1) (filt set n) max) n
      | otherwise   = erato (n+1) set max
```

В написанной программе действительно строится множество, содержащее все простые числа от `2` до заданного значения; это можно проверить, пробуя задать в качестве аргумента получившемуся множеству разные значения. Так, например, при вызове `(primesTo 100) 13` будет выдано значение `True`, что означает, что число `13` содержится в множестве простых чисел от `2` до `100`.

Написанная нами программа не слишком выразительна, прежде всего, потому, что в основной функции `erato` числа просто проверяются по очереди на принадлежность к множеству простых чисел. На самом деле, в алгоритме Эратосфена числа для фильтрации множества (просеивания) выбираются последовательно из самого строящегося множества простых чисел. Однако, при представлении множеств характеристическими функциями выборка элементов из множества невозможна. Точнее, такую выборку можно производить только путем последовательной проверки всех возможных элементов (в данном случае – натуральных чисел) на принадлежность множеству. Впоследствии, впрочем, мы еще раз вернемся к этой задаче и напишем более выразительную и эффективную функцию для решения той же самой задачи, правда, без использования представления множества своей характеристической функцией.

Однако идея представления множеств характеристическими функциями может быть использована и более продуктивно. Рассмотрим, например, построение множества слов в заданном алфавите (языка), заданного определенными правилами. В качестве исходного алфавита для задания языка будем рассматривать произвольные символы из множества, задаваемого типом `Char`. Тогда слова в этом алфавите – это просто списки символов (строки). Язык в таком алфавите может быть представлен характеристической функцией множества слов языка, то есть функцией, которая выдает значение `True` на словах, принадлежащих языку, и `False` на всех остальных словах. Описание типа `Lang` задает такое функциональное представление языков:

```
type Lang = String -> Bool
```

Конечно, тип `Lang` – это просто способ задания множества слов его характеристической функцией (или `Set String` в контексте наших предыдущих определений).

Теперь попробуем построить различные языки, используя в качестве способа описания языка задание регулярного выражения. Напомним основные правила построения регулярных выражений.

Отдельная буква алфавита представляет регулярное выражение, задающее язык, состоящий из единственного слова, содержащего эту единственную букву.

Специальное обозначение ε представляет регулярное выражение, задающее язык, содержащий только пустое слово.

Если α и β – два регулярных выражения, то (αβ) – тоже регулярное выражение, представляющее катенацию соответствующих языков (множество слов, образованных катенацией слов языка α со словами языка β).

Если α и β – два регулярных выражения, то (α | β) – тоже регулярное выражение, представляющее альтернацию соответствующих языков (множество слов, образованных объединением слов языка α со словами языка β).

Если α – регулярное выражения, то (α *) – тоже регулярное выражение, представляющее итерацию соответствующего языка (множество слов, образованных объединением пустого слова и слов, образованных катенацией произвольного количества слов языка α).

(α +) – регулярное выражение, являющееся просто сокращением для (α (α *)) Эту операцию мы будем называть операцией непустой итерации. Она не всегда рассматривается в качестве одной из основных операций, поскольку может быть выражена через обычную операцию итерации. Верно и обратное: операцию итерации можно выразить с помощью операции непустой итерации. Язык, задаваемый выражением (α *), совпадает с языком, задаваемым выражением (ε | (α +))

Операциям катенации, альтернации и итерации, использующимся для построения регулярных выражений, обычно приписывают приоритет в соответствии со следующими правилами: операцией с наивысшим приоритетом считается операция итерации, следующей по старшинству приоритетов является операция катенации, самой низкоприоритетной операцией считается операция альтернации. Правила приоритетов позволяют убирать скобки из регулярных выражений в тех случаях, когда порядок выполнения операций определяется их приоритетами. Например, регулярное выражение `(ab+)*` является сокращенной записью выражения `((a(b+))*)` и задает язык, содержащий слова, состоящие из букв `a` и `b`, начинающиеся с буквы `a`, в которых после любой буквы `a` обязательно стоит хотя бы одна буква `b`.

Мы опишем функции, которые позволят строить произвольные регулярные выражения. Фактически мы будем выполнять операции не над выражениями, а над соответствующими языками, так что получившееся выражение можно будет сразу использовать для проверки того,

принадлежит ли слово построенному языку. Очень просто описать функцию `letter`, которая строит регулярное выражение, состоящее из единственной буквы:

```
letter :: Char -> Lang
letter symbol word = word == [symbol]
```

то есть соответствующий регулярный язык содержит только такое слово `word`, которое совпадает со словом `[symbol]`, состоящим из единственного символа.

Язык, содержащий единственное пустое слово, соответствующий регулярному выражению ε, также описать не составляет труда:

```
empty :: Lang
empty word = word == []
```

Прежде, чем продолжать программирование построителя регулярных выражений, сделаем одно замечание относительно выполнения примитивных логических операций «и» и «или», которые в языке *Haskell* обозначаются символами бинарных операций `||` и `&&`. В различных языках программирования и системах программированиях различают два разных подхода к процессу исполнения этих операций. При первом подходе операнды операций всегда вычисляются до начала выполнения операции, даже если результат выполнения операции может быть определен по значению только одного из двух операндов. При втором подходе обязательно вычисляется только первый операнд операции, а второй вычисляется только в случае необходимости. Так, например, если при вычислении `a && b` значением первого операнда оказалось `False`, то второй операнд не вычисляется, поскольку результатом выполнения операции все равно будет `False`. Если же значением первого операнда оказалось `True`, то второй операнд вычисляется, и результат этого вычисления и является результатом выполнения операции `&&`. Говорят, что вычисления в этом случае производятся «по МакКарти» (John McCarthy – автор языка Лисп, первого функционального языка программирования, известный также своими работами в области математической логики).

Поведение программы может быть различным в зависимости от того, какой из двух возможных подходов принят в выбранном языке или системе программирования. Дело в том, что вычисление второго операнда может в некоторых случаях закончиться неудачно, как, например, в выражении `(x != 0) && (y/x > 3)`, когда значение переменной `x` окажется равным нулю, или может не закончиться никогда, если выражение, представляющее второй операнд, при некоторых значениях аргументов будет представлять собой «зациклившуюся» функцию. В языке *Haskell* вычисления логических выражений всегда производятся «по МакКарти». Подробнее логику порядка вычислений мы рассмотрим в следующем разделе, а пока просто отметим, что вычисление логических

выражений, содержащих знаки операций || и &&, «безопасно». «Безопасность» понимается в том смысле, что если значение первого операнда уже определяет результат операции, то в этом случае второй операнд не может повлиять на результат, даже если его вычисление может привести к ошибке.

Теперь можно рассмотреть две простые операции над языками, аналогичные операциям добавления и удаления элементов из множества. Операция addWord будет добавлять заданное слово в язык, а remWord – удалять слово из языка. Эти операции не используются напрямую для построения регулярных выражений, однако их можно использовать в качестве вспомогательных процедур при построении языков. Кроме того, на примере этих простых операций можно посмотреть технику работы с языками.

```
addword :: String -> Lang -> Lang
remword :: String -> Lang -> Lang

addWord w lang word = (word == w) || (lang word)
remWord w lang word = (word /= w) && (lang word)
```

Например, в уравнении для функции addWord написано буквально следующее: при добавлении слова w в язык lang получается такой язык, что слово word будет ему принадлежать, если оно совпадает с w или содержалось в исходном языке lang. Конечно, это просто операции добавления и удаления элемента из множества, уже рассмотренные нами.

Операция альтернации соответствует просто объединению языков. Подобную операцию мы описывали только что, когда представляли функции для работы с множествами, представленными характеристическими функциями:

```
alt :: Lang -> Lang -> Lang
(lang1 `alt` lang2) word = (lang1 word) || (lang2 word)
```

Гораздо сложнее оказывается описать операцию катенации. При катенации двух языков получается новый язык, содержащий слова, которые можно разбить на два слова так, что первая часть будет принадлежать первому языку, а вторая – второму. Такое разбиение слова на два других можно производить различными способами. Зададим тип операции катенации и напишем несколько уравнений, соответствующих различным способам разбиения слова. В первом уравнении будет представлен случай, когда пустое слово принадлежит результирующему языку. Пустое слово можно разбить на части единственным способом – на два пустых слова.

```
cat :: Lang -> Lang -> Lang
(lang1 `cat` lang2) [] = (lang1 []) && (lang2 [])
```

В анализе непустого слова выделим два случая. В первом из них слово разбивается на пустое слово и само это же слово. Второй вариант

разбиения получается, если отделить одну первую букву у первого слова и пытаться остаток слова разбить на два. Такое разбиение можно осуществить с помощью рекурсивного вызова операции катенации. Теперь уже можно дописать оставшееся уравнение для операции катенации.

```
(lang1 `cat` lang2) word@(x:s) =
     (lang1 [] && lang2 word) || (lang1' `cat` lang2) s
                                  where lang1' w = lang1 (x:w)
```

Следует внимательно изучить определение операции катенации, чтобы понять, как работает эта функция. Так же, как и многие другие рассмотренные нами ранее функции высших порядков, функция содержит рекурсивное обращение к той же самой операции. Ранее одним из аргументов функции был список, и рекурсивное обращение выполнялось для аргумента, представляющего фрагмент исходного списка. В операции катенации ситуация несколько сложнее: рекурсивное обращение выполняется для случая, когда один из аргументов – это новая функция, а к укороченному аргументу применяется результат рекурсивного обращения.

Остальные регулярные операции над языками определить уже несложно. Операция непустой итерации может быть определена рекурсивно с помощью обращения к операции катенации. Следует только позаботиться о том, чтобы избежать зацикливания в случае, когда итерируется язык, содержащий пустое слово. В этом случае возможна ситуация, когда программа будет пытаться разбивать анализируемое слово на бесконечную катенацию пустых слов. Поэтому следует рассмотреть отдельно два случая. Если анализируемое слово пустое, то оно принадлежит итерации языка, если исходный язык также содержал пустое слово. Если нет – то слово надо разбивать только на непустые подслова; этого можно добиться, если рассмотреть новый язык, полученный удалением пустого слова из исходного языка. Получается следующее определение операции непустой итерации:

```
iterPlus :: Lang -> Lang
iterPlus lang [] = lang []
iterPlus lang word =
    lang word || (lang' `cat` (iterPlus lang')) word
       where lang' = remWord [] lang
```

Теперь операция итерации может быть легко выражена через операцию непустой итерации, поскольку соответствующий язык получается простым добавлением пустого слова в язык, полученный с помощью непустой итерации.

```
iter :: Lang -> Lang
iter lang = addWord [] (iterPlus lang)
```

Давайте еще немного расширим наши возможности задания языков, введя еще одну дополнительную операцию над регулярными

выражениями. Будем считать, что если α – регулярное выражение, то [α] – тоже регулярное выражение, задающее тот же язык, что и (ε | α). Назовем функцию для образования такого языка `virtual`.

```
virtual :: Lang -> Lang
virtual = alt empty
```

Теперь мы имеем механизм, с помощью которого можно определять регулярные выражения, задающие те или иные интересующие нас языки. Если собрать все определения функций в единый листинг, то получится достаточно полный набор функций для определения регулярных выражений (листинг 2.13).

Листинг 2.13. Функции для определения регулярных выражений

```
-- Типы данных и функциий для определения регулярных выражений
-- Тип, задающий регулярный язык
type Lang = String -> Bool

-- Регулярное выражение, состоящее из единственной буквы
letter :: Char -> Lang
letter symbol word = word == [symbol]

-- Регулярное выражение, задающее язык, содержащий только
пустое слово
empty :: Lang
empty word = word == []

-- Функции добавления и удаления слова
addWord :: String -> Lang -> Lang
remWord :: String -> Lang -> Lang

addWord w lang word = (word == w) || (lang word)
remWord w lang word = (word /= w) && (lang word)

-- Функция, задающая операцию альтернации (A | B)
alt :: Lang -> Lang -> Lang
(lang1 `alt` lang2) word = (lang1 word) || (lang2 word)

-- Функция, задающая операцию катенации (A B)
cat :: Lang -> Lang -> Lang
(lang1 `cat` lang2) [] = (lang1 []) && (lang2 [])
(lang1 `cat` lang2) word@(x:s) =
    (lang1 [] && lang2 word) || (lang1' `cat` lang2) s
                          where lang1' w = lang1 (x:w)

-- Функция, задающая операцию непустой итерации (A+)
iterPlus :: Lang -> Lang

iterPlus lang [] = lang []
iterPlus lang word =
```

```
    lang word || (lang' `cat` (iterPlus lang')) word
        where lang' = remWord [] lang


-- Функция, задающая операцию итерации (A*)
iter :: Lang -> Lang

iter lang = addWord [] (iterPlus lang)

-- Функция, задающая выражение [A]
virtual :: Lang -> Lang
virtual = alt empty
```

Теперь мы можем использовать введенные функции для определения регулярных выражений, задающих те или иные языки. Например, определим язык, содержащий изображения вещественных чисел. Регулярное выражение для этого языка можно определить постепенно, вводя обозначения для различных частей числа. Так, определим сначала понятие десятичной цифры и обозначим соответствующее регулярное выражение идентификатором `digit`.

```
digit = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Непустую последовательность цифр можно задать с помощью регулярного выражения, в состав которого будет входить регулярное выражение, обозначенное нами идентификатором `digit`:

```
unsigned = digit +
```

Идентификатор `unsigned` обозначает язык, задающий изображения беззнаковых целых. Если приписать слева к последовательности цифр знак `'+'` или `'-'`, получится целое число со знаком. Понятие `integral` (целое число, возможно, со знаком) можно выразить следующим регулярным выражением.

```
integral = [ + | - ] unsigned
```

Наконец, полностью регулярное выражение, задающее изображения вещественных чисел, можно будет задать следующим образом:

```
number = integral [ . unsigned ] [ e integral ]
```

На языке *Haskell* программа, задающая описанные нами регулярные выражения, точно соответствует этим выражениям; разница состоит только в обозначениях операций (листинг 2.14).

Листинг 2.14. Регулярное выражение, определяющее синтаксис числа

```
digit = letter '0' `alt`
        letter '1' `alt`
        letter '2' `alt`
        letter '3' `alt`
        letter '4' `alt`
        letter '5' `alt`
        letter '6' `alt`
```

```
        letter '7' `alt`
        letter '8' `alt`
        letter '9'
unsigned = iterPlus digit
integral = virtual (letter '+' `alt` letter '-')
           `cat` unsigned
number   = integral `cat`
           virtual (letter '.' `cat` unsigned) `cat`
           virtual (letter 'e' `cat` integral)
```

Правильность работы программы можно проверить, подставляя в качестве аргумента построенной функции `number` строковые изображения чисел. Например, выражение `number "-3.14"` выдаст в качестве результата работы `True`, так же, как и выражение `number "12345.54321e12345"`; а `number "-3.14.2"` выдаст значение `False`.

Конечно, приведенное определение способа построения регулярных языков красиво, но не очень эффективно. Проверка принадлежности слова языку делается путем полного перебора всех возможных способа его построения. В теории формальных языков имеются гораздо более эффективные способы анализа регулярных языков. Построенная нами программа пригодна только для анализа довольно коротких строк. Анализ строк, состоящих из нескольких десятков символов, занимает уже заметное время, а анализ строк, длиной в сотни символов, таким способом вообще невозможен из-за слишком большого времени вычислений.

Примеры решения задач

Многие задачи обработки списков могут быть решены очень просто применением двух функций высших порядков – отображения и свертки. Функция отображения (`map`) применяет одно и то же преобразование (функцию) ко всем элементам заданного списка. Применение функции отображения часто помогает подготовить некоторый список для последующей обработки.

Функция свертки (часто называемая функцией `reduce`; в языке *Haskell* представлена двумя разновидностями – функциями `foldl` и `foldr`). Она позволяет последовательно применить одну и ту же операцию к элементам списка для того, чтобы получить некоторое «агрегированное» значение.

Комбинация применений `map` и `foldr` часто позволяет сразу получить нужный результат.

Приведем несколько примеров задач, легко решаемых с помощью применения комбинации функций отображения и свертки.

Задание 1. Написать функцию, вычисляющую длину самой длинной строки в заданном списке строк.

Решение. Искомая функция сводится к тому, чтобы вычислить длины всех строк в списке, а затем найти максимальную из них. Функция вычисления длины списка – это стандартная функция системы `length`, а операция, выдающая максимальное из двух числовых значений – это стандартная функция `max`. Очевидно, что длины всех строк исходного списка легко получить с помощью функции отображения `map`, таким образом, если `s` – это исходный список строк, то `map length s` – это список их длин. Максимальное значение в списке можно найти с помощью функции свертки, то есть, если `s` – это список неотрицательных целых чисел, то выражение `foldr max 0 s` выдает максимальное число в этом списке. Таким образом, искомая функция может быть построена следующим образом:

```
maxLength s = foldr max 0 (map length s)
```

Можно заметить, что фактически искомая функция является суперпозицией двух функций, полученных из функций `map` и `foldr` частичной параметризацией (каррингом). Такая частичная параметризация может быть задана с помощью выражений

```
foldr max 0
```

и

```
map length
```

а суперпозиция двух функций может быть выражена с помощью стандартной операции языка *Haskell* (`.`). Отсюда сразу же следует еще более короткое решение данной задачи, которое приведено в листинге У2.1.

Листинг У2.1. Длина максимально длинной строки в списке

```
-- Задание: Написать функцию, вычисляющую длину самой длинной
--          строки в заданном списке строк.

-- Функция легко получается суперпозицией применения
-- функций map и foldr
maxLength :: [String] -> Int
maxLength = (foldr max 0) . (map length)

-- Тестовая функция
-- Ожидаемый результат: [5, 0, 3, 0]
main = [maxLength ["ab","12345","","abcd"],
        maxLength ["",""],
        maxLength ["abc","ab","a"],
        maxLength []]
```

Задание 2. Написать функцию, разделяющую заданный текст на слова. Словом считать любую последовательность символов, ограниченную пробелом, то есть таким символом, что стандартная функция `isSpace` на этом символе выдает значение `True`.

Решение. Искомая функция должна осуществлять последовательный просмотр символов исходного текста, преобразуя его в список строк («слов»). Задачи последовательного просмотра списка и применения к его элементам некоторой функции удобно записывать с помощью одной из двух стандартных функций высших порядков – `map` или `foldr`. Функцию `map` удобнее применять в том случае, когда результат обработки сохраняет структуру списка-аргумента, а функцию `foldr` – в том случае, когда эта структура меняется. В данном случае список символов превращается в список строк, так что функция `map` для решения задачи не подходит. Функцию `foldr` применить можно, если определить, каким образом очередной элемент исходного списка преобразует формирующийся результат. Тогда, если задать подходящее начальное значение `seed`, то искомую функцию можно будет получить в виде выражения `foldr f seed`, где `f` – функция обработки очередного элемента, а `f` – выбранное начальное значение.

В данной задаче начальным значением может быть список, содержащий единственное пустое слово, а функция обработки очередного символа должна либо присоединять этот символ к первой строке списка, либо образовывать новое слово. Поясним эту идею на примере.

Пусть исходная строка выглядит следующим образом.

```
"на берегу пустынных волн стоял он, дум великих полн."
```

Пусть конец строки уже обработан (напомним, что функция `foldr` обрабатывает список в направлении от конца к началу), и при этом получен список слов

```
["ум", "великих", "полн."]
```

а очередным символом для обработки, соответственно, является символ `'д'`. Поскольку этот символ не является пробелом, то его нужно просто присоединить к первому слову формирующегося списка, при этом получится

```
["дум", "великих", "полн."]
```

Очередным обрабатываемым символом будет пробел. В этом случае наша функция должна образовать новое слово в списке, так что список будет иметь вид

```
["", "дум", "великих", "полн."]
```

Если слова в исходной строке разделяются ровно одним пробелом, то описанных двух случаев достаточно для обработки всей исходной строки. Если разделяющих пробелов может быть несколько, то для того, чтобы в списке не образовывались пустые слова, надо предусмотреть еще один случай – когда очередным обрабатываемым символом является пробел, а в списке слов первое слово – пустое. В этом случае пробел просто пропускается.

Итак, опишем функцию `glue` с двумя аргументами – символом и списком строк, которая присоединяет очередной символ к списку строк

вышеописанным способом. Описание функции может выглядеть, например, так.

```
glue c s@(x:t) =
    if isSpace c then if x == "" then s else "":s else (c:x):t
```

Теперь для обработки всего списка `s` достаточно использовать вызов

```
foldr glue [""] s
```

На самом деле этого недостаточно, если в качестве исходной строки может использоваться пустая строка или если перед первым словом исходной строки имеется некоторое количество пробелов. В обоих случаях в результирующем списке появляется «лишнее» пустое слово. Его можно удалить с помощью простой функции

```
delEmpty (""":t) = t
delEmpty s = s
```

которая оставляет свой аргумент-список без изменений, если только первым элементом этого списка не является пустая строка.

Полный текст решения приведен в листинге У2.2.

Листинг У2.2. Разложение строки на список составляющих эту строку слов.

```
{-- Задание: Написать функцию, разделяющую заданный текст
       на слова. Словом считать любую последовательность
       символов, ограниченную пробелом, то есть таким
       символом, что стандартная функция isSpace на этом
       символе выдает значение True.
--}

import Char(isSpace)

-- Функция "приклеивания" символа к списку слов.
-- Аргументы: c - приклеиваемый символ,
--            s - список строк.
-- Результат: исходный список строк с "приклееным" символом.
-- Функция присоединяет символ c к первой строке списка s,
-- если только это не пробел. Пробел либо начинает новую
-- строку в списке, либо оставляет список без изменений,
-- если новая строка в нем уже начата.
glue :: Char -> [String] -> [String]
glue c s@(x:t) = if isSpace c then
                   if x == "" then s else "":s
                 else (c:x):t

-- Функия отбрасывания пустой строки из начала списка
-- Аргумент: s - исходный список строк
-- Результат: тот же список без первого элемента,
--            если этим элементом является пустая строка
delEmpty :: [String] -> [String]
delEmpty (""":t) = t
delEmpty s      = s
```

```
-- Основная функция для решения задачи.
-- Последовательно применяет функцию glue к символам
-- исходной строки, формируя результирующий список строк.
-- Аргумент: символьная строка;
-- Результат: список слов исходной строки.
wordify :: String -> [String]
wordify = delEmpty . (foldr glue [""])

-- Тестовая функция
-- Тестовые строки:
test1 = "Darwinism is used by promotors"
test2 = ""
test3 = "  Another  example to illustrate are  systems  "

{-- Ожидаемый результат: [
  ["Darwinism","is","used","by","promotors"],
  [],
  ["Another","example","to","illustrate","are","systems"]]
--}

main = [wordify test1,
        wordify test2,
        wordify test3]
```

Задание 3. Дерево задано с помощью следующего описания структуры данных.

```
data Tree a = Node a [Tree a]
```

то есть дерево представляет собой корневой узел, содержащий некоторое значение произвольного типа `a` и список поддеревьев. Написать функцию, вычисляющую высоту дерева.

Решение. Функцию для вычисления высоты дерева легко определить рекурсивно. Для этого нужно просто для каждого узла вычислить максимальную из высот его поддеревьев, а затем добавить единицу к полученному результату. Если считать, что функция вычисления высоты имеет следующее описание типа

```
height :: Tree a -> Int
```

то для списка деревьев `list` можно построить список из их высот с помощью функции `map`:

```
map height list
```

Максимальное значение из списка неотрицательных значений выдается стандартной функцией `maximum`, однако, эта функция не работает для случая пустого списка. Мы можем добавить к списку высот число 0, чтобы функция maximum выдавала нужный нам результат, впрочем, нужный нам результат легко получить и с помощью функции `foldr`:

```
foldr max 0 list
```

Таким образом, функцию вычисления высоты дерева можно определить одной строчкой:

```
height (Node _ list) = 1 + foldr max 0 (map height list)
```

В этом задании самым трудоемким будет определение тестовых данных для проверки работоспособности получившейся функции. Пример полностью оформленного задания можно увидеть в листинге У2.3.

Листинг У2.3. Вычисление высоты дерева.

```
-- Задание: Дерево задано с помощью следующего описания
--          структуры данных.
data Tree a = Node a [Tree a]
--          то есть дерево представляет собой корневой узел,
--          содержащий некоторое значение произвольного типа a
--          и список поддеревьев. Написать функцию,
--          вычисляющую высоту дерева.

-- Функция вычисления высоты дерева.
-- Аргумент: исходное дерево
-- Результат: высота дерева.
height :: Tree a -> Int
height (Node _ list) = 1 + foldr max 0 (map height list)

-- Тестовая функция
-- Тестовые деревья:
tr1 = Node 1 [Node 2 [], Node 3 [], Node 4 []] -- высота = 2
tr2 = Node 1 [Node 2 [Node 3 [Node 4 []]]]     -- высота = 4
tr3 = Node 1 []                                -- высота = 1
tr4 = Node 1 [
        Node 2 [],
        Node 3 [
          Node 5 [],
          Node 6 []
        ],
        Node 4 [
          Node 7 [],
          Node 8 [
            Node 9 [],
            Node 10 []
          ]
        ]
      ]             -- высота = 4

-- Ожидаемый результат: [2,4,1,4]

main = [height tr1, height tr2, height tr3, height tr4]
```

Задание 4. Дерево задано с помощью следующего описания структуры данных.

```
data Tree a = Node a [Tree a]
```
то есть дерево представляет собой корневой узел, содержащий некоторое значение произвольного типа `a` и список поддеревьев. Написать функцию, которая для заданного дерева из строк выдает список всех строк, содержащих хотя бы одну цифру.

Решение. Прежде всего, определим функцию, которая проверяет, содержит ли строка хоть одну цифру. Это легко сделать с помощью все тех же функций высшего порядка `map` и `foldr`.

```
hasDigit = (foldr (||) False) . (map isDigit)
```

Здесь стандартная функция `isDigit` проверяет, является ли заданный символ цифрой, а функция `foldr` сворачивает получившийся список логических значений в единственное логическое значение результата. Далее надо проделать то же самое с деревом строк, то есть «свернуть» дерево в список. Для свертки нам потребуется функция с двумя аргументами, которая присоединяет строку к заданному списку строк, если эта строка содержит цифру. Подобная функция может выглядеть так:

```
glue s list | hasDigit s = s:list
            | otherwise = list
```

Теперь определим функцию свертки дерева, подобную функции `foldr` для списков или функции foldTree, приведенной в книге для свертки двоичных деревьев. Это – самая сложная функция в этой задаче. Тип функции свертки дерева должен быть аналогичен типу функции свертки двоичного дерева:

```
foldTree :: (a -> b -> b) -> b -> (Tree a) -> b
```

то есть первым аргументом функции является бинарная операция, вторым – начальное значение, и третьим – дерево из элементов некоторого типа. Результатом должно быть «свернутое» значение.

Понятно, что, поскольку дерево содержит в себе список поддеревьев, то для свертки этого списка надо применить одну из функций свертки списка `foldl` или `foldr`. Начальным значением для этой свертки может быть начальное значение, переданное в качестве аргумента нашей функции `foldTree`, а в качестве бинарной операции должна быть задана операция, которая получает на вход элемент списка – дерево – и некоторое значение, а выдает в качестве результата значение типа второго аргумента. Такой операцией может быть опять же операция свертки дерева. Действительно, если задана некоторая функция `f` типа `a -> b -> b`, то выражение `foldTree f` будет иметь тип `b -> (Tree a) -> b`, а это то, что нам нужно. Тип этой функции – подходящий для решения задачи с помощью свертки списка `foldl`.

Итак, если мы имеем список деревьев `treeList` и функцию свертки дерева `foldTree`, то список деревьев можно свернуть с помощью вызова

```
foldl (foldTree f) seed treeList
```

Чтобы получить окончательный результат, нужно применить заданную функцию `f` к корню дерева и результату свертки списка поддеревьев. Таким образом, функция свертки дерева приобретает вид

```
foldTree f seed (Node el treeList) =
                f el (foldl (foldTree f) seed treeList)
```

Теперь искомую функцию для решения поставленной задачи определить совсем просто:

```
listDigits = foldTree glue []
```

Здесь в качестве начального значения берется пустой список строк, а в качестве бинарной операции для свертки дерева – операция условного присоединения строки к списку, описанная нами выше. Полное решение поставленной задачи представлено в листинге У2.4.

Листинг У2.4. Список строк дерева, содержащих цифры.

```
import Char

-- Задание: Дерево задано с помощью следующего
--          описания структуры данных.
data Tree a = Node a [Tree a]
--          Написать функцию, которая для заданного дерева
--          из строк выдает список всех строк,
--          содержащих хотя бы одну цифру.

-- Функция проверки, есть ли в заданной строке
-- хотя бы одна цифра.
-- Аргумент: исходная строка
-- Результат: True, если в строке есть хотя бы одна цифра,
--            False в противном случае
-- Функция описывается как стандартная суперпозиция
-- функций map и foldr.
hasDigit :: String -> Bool
hasDigit = (foldr (||) False) . (map isDigit)

-- Функция условного присоединения строки к списку.
-- Аргументы: s - строка для присоединения;
--            list - список строк
-- Результат: список, полученный в результате присоединения
--            первого аргумента ко второму, если первый
--            аргумент содержал хотя бы одну цифру. Если нет –
--            то второй аргумент выдается без изменения.
glue :: String -> [String] -> [String]
glue s list | hasDigit s = s:list
            | otherwise = list

-- Функция свертки дерева аналогично функции свертки списков.
-- Аргументы: f – бинарная операция, применяемая для свертки;
```

```
--            seed – начальное значение;
--            (Node el treeList) – сворачиваемое дерево.
-- Результат: результат последовательного применения
--            операции f к узлам дерева.
foldTree :: (a -> b -> b) -> b -> (Tree a) -> b
foldTree f seed (Node el treeList) =
                f el (foldl (foldTree f) seed treeList)

-- Функция для решения посталенной задачи получения списка
-- строк, содержащих хотя бы одну цифру.
-- Аргумент: дерево строк.
-- Результат: искомый список элементов дерева.
listDigits :: Tree String -> [String]
listDigits = foldTree glue []

-- Тестовые деревья:
tree1 = Node "1" [Node "22" [], Node "a" [], Node "a4" []]
tree2 = Node "1b" [Node "22" [Node "" [Node "aaa" []]]]
tree3 = Node "" []
tree4 = Node "abc" [
          Node "2a2" [],
          Node "s3s" [
            Node "***" [],
            Node "hello" []
          ],
          Node "123456789" [
            Node "H2O" [],
            Node "2+3=5" [
              Node "2*2=4" [],
              Node "10" []
            ]
          ]
        ]

-- Тестовая функция
main = [listDigits tree1,
        listDigits tree2,
        listDigits tree3,
        listDigits tree4]

-- Ожидаемый результат:
--     [["1","22","a4"],
--      ["1b","22"],
--      [],
--      ["2a2","s3s","123456789","H2O","2+3=5","2*2=4","10"]]
```

Конечно, приведенное решение далеко не единственное. Например, функция `glue` немного выбивается из общего стиля решения, так что можно попробовать от нее избавиться. Вместо этого можно сначала построить список всех элементов дерева (с помощью все той же функции

свертки дерева), а потом отфильтровать лишь те строки, которые содержат хотя бы одну цифру. В результате получится следующее выражение для функции `listDigits`:

```
listDigits t = [s | s <- foldTree (:) [] t, hasDigit s]
```

или даже еще короче

```
listDigits = (filter hasDigit) . (foldTree (:) [])
```

Это решение выглядит более изящным, в нем не требуется более определять функцию «условного присоединения строки», но оно несколько менее эффективно, поскольку уже построенный список строк просматривается затем вторично только для того, чтобы исключить строки, не содержащие цифр.

Задание 5. Двоичное дерево задано с помощью следующего описания структуры данных.

```
data Tree a = Empty | Node (Tree a) a (Tree a)
```

то есть дерево либо является пустым, либо представляет собой корневой узел, содержащий некоторое значение произвольного типа `a` и два поддерева – левое и правое. Написать функцию, которая проверяет, является ли заданное дерево из элементов класса `Ord` упорядоченным деревом поиска.

Решение. Дерево поиска – это дерево, узлы которого упорядочены таким образом, что для каждого узла в его левом поддереве находятся элементы, меньшие корневого, а в правом поддереве – большие корневого. Если воспользоваться этим определением напрямую, то можно написать функцию, которая для каждого узла вычисляет максимальное и минимальное значения узлов в поддереве, корнем которого является этот узел и заодно проверяет, является ли дерево деревом поиска. Функция может выглядеть следующим образом.

```
ordTree :: (Ord a) => Tree a -> (Bool, a, a)
ordTree Empty = error "Not applicable"
ordTree (Node Empty e Empty) = (True, e, e)
ordTree (Node Empty e t2)    =
      (ord_t2 && e <= min_t2, e, max_t2)
           where (ord_t2, min_t2, max_t2) = ordTree t2
ordTree (Node t1 e Empty)    =
      (ord_t1 && e >= max_t1, min_t1, e)
           where (ord_t1, min_t1, max_t1) = ordTree t1
ordTree (Node t1 e t2)       =
      (ord_t1 && ord_t2 && e >= max_t1 && e <= min_t2,
       min_t1, max_t2)
           where (ord_t1, min_t1, max_t1) = ordTree t1
                 (ord_t2, min_t2, max_t2) = ordTree t2
```

Это традиционное рекурсивное решение довольно длинно и неуклюже. С помощью функций высших порядков решение задачи получается, может быть, ненамного короче, но несомненно элегантнее.

Для этого, прежде всего, определим функцию свертки двоичного дерева, которая обходит дерево в порядке справа налево (это, конечно, в точности та же функция, которая приводилась нами в тексте книги).

```
foldTree :: (a -> b -> b) -> b -> Tree a -> b
foldTree f seed Empty           =  seed
foldTree f seed (Node t1 e t2)  =
                    foldTree f (f e (foldTree f seed t2)) t1
```

Теперь можно определить функцию (бинарную операцию), которая будет последовательно обрабатывать узлы дерева. Ей будет необходимо сравнивать по величине два значения из последовательных узлов дерева, поэтому в результате обработки узла функция должна выдавать не только результат сравнения, но и значение текущего узла для последующего сравнения. Таким образом, тип функции обработки узлов дерева типа `a` получается следующим:

```
comp :: Ord a => a -> (Bool, a) -> (Bool, a)
```

а тело функции можно записать следующим образом

```
comp elem1 (b, elem2) = (b && elem1 <= elem2, elem1)
```

Здесь, впрочем, есть одна тонкость: первое сравнение проводить не с чем. Эту трудность можно преодолеть различными способами. Например, можно предоставлять функции проверки в качестве дополнительного аргумента некоторое «бесконечно большое» значение. Можно вычислить «максимальное» значение с помощью сравнительно короткого прохода по правым веткам исходного дерева. Мы предложим третье решение. На самом деле ситуация, когда значение может отсутствовать, встречается достаточно часто, так что в стандартной библиотеке языка *Haskell* специально для этих случаев имеется определение типа `Maybe`, которое выглядит так:

```
data Maybe a = Nothing | Just a
```

Можно воспользоваться этим определением и переопределить функцию сравнения узлов следующим образом:

```
comp :: Ord a => a -> (Bool, Maybe a) -> (Bool, Maybe a)
comp elem1 (b, Nothing)    = (b, Just elem1)
comp elem1 (b, Just elem2) = (b && elem1 <= elem2, Just elem1)
```

Теперь функция для проверки того, является ли заданное дерево упорядоченным деревом поиска, будет выглядеть совсем коротко:

```
ordTree t = s where (s, _) = foldTree comp (True, Nothing) t
```

Полностью оформленное решение задачи представлено в листинге У2.5.

Листинг У2.5. Проверка двоичного дерева поиска

```
-- Задание: Двоичное дерево задано с помощью следующего
--          описания структуры данных.
data Tree a = Empty | Node (Tree a) a (Tree a)
```

```
--          то есть дерево либо является пустым, либо
--          представляет собой корневой узел, содержащий
--          некоторое значение произвольного типа a и два
--          поддерева – левое и правое. Написать функцию,
--          которая проверяет, является ли заданное дерево из
--          элементов класса Ord деревом поиска.

-- Функция свертки дерева аналогично функции свертки списков.
-- Аргументы: f – бинарная операция, применяемая для свертки;
--            seed – начальное значение;
--            arg3 – сворачиваемое дерево.
-- Результат: результат последовательного применения операции
--         f к узлам дерева в порядке правостороннего обхода.
foldTree :: (a -> b -> b) -> b -> Tree a -> b
foldTree f seed Empty           =  seed
foldTree f seed (Node t1 e t2)  =
                foldTree f (f e (foldTree f seed t2)) t1

-- Функция сравнения двух узлов дерева.
-- Аргументы: arg1 – первый узел для сравнения;
--            arg2 – пара из результата предыдущего
--                   сравнения и второго узла.
-- Результат: результат сравнения узлов, соединенный с
--            предыдущим результатом сравнения операцией &&
comp :: Ord a => a -> (Bool, Maybe a) -> (Bool, Maybe a)
comp elem1 (b, Nothing)    = (b, Just elem1)
comp elem1 (b, Just elem2) = (b && elem1 <= elem2, Just elem1)

-- Основная функция проверки дерева.
-- Аргумент: исходное двоичное дерево
-- Результат: True, если дерево является деревом поиска,
--            False в противном случае
ordTree :: Ord a => Tree a -> Bool
ordTree t = s where (s, _) = foldTree comp (True, Nothing) t

-- Тесты для проверки и главная программа.
t1 = Node
       (Node (Node Empty 1 Empty) 3 (Node Empty 4 Empty))
       5
       (Node (Node Empty 6 Empty) 8 (Node Empty 10 Empty))
t2 = Node
       (Node (Node Empty 1 Empty) 6 (Node Empty 4 Empty))
       5
       (Node (Node Empty 6 Empty) 9 (Node Empty 8 Empty))

main = [ordTree t1, ordTree t2]
-- Ожидаемый результат: [True,False]
```

Задание 6. Структура графа задана списками смежности номеров вершин, то есть списком, элементами которого являются пары, состоящие из номера вершины и списка номеров вершин, инцидентных ей.

```
type Graph = [(Int, [Int])]
```

Написать функцию, которая проверяет, существует ли в графе путь, соединяющий вершины с двумя заданными номерами.

Решение. Вершина `a` соединена некоторым путем с вершиной `b`, если это либо одна и та же вершина, либо из вершины `a` можно попасть в вершину `b` за некоторое число *шагов*. Основной задачей является определение функции, которая проверяет, можно ли попасть из вершины `a` в вершину `b` за `n` шагов. Тогда, если такая функция имеется, то можно подсчитать количество вершин в графе и проверить, можно ли попасть из вершины `a` в вершину `b` за число шагов, равное количеству вершин в графе.

Итак, пусть определены следующие функции:

```
-- Функция, определяющая число вершин в графе:
count :: Graph -> Int
-- Функция (steps n a b g) для проверки, можно ли в графе g
-- попасть из вершины a в вершину b не более, чем за n шагов.
steps :: Int -> Int -> Int -> Graph -> Bool
```

Тогда искомая функция проверки существования пути `(path a b g)` может быть задана следующим выражением:

```
path a b g = steps (count g) a b g
```

Сначала определим более простую функцию подсчета числа вершин в графе. Для этого последовательно зададим следующие функции обработки списков, состоящих из целых чисел.

```
-- функция вставки целого числа в упорядоченный список целых
insert :: Int -> [Int] -> [Int]
insert n [] = [n]
insert n lst@(x:l) | n < x     = n : lst
                   | n == x    = lst
                   | otherwise = x : insert n l
-- функция сортировки списка целых чисел
-- методом «простых вставок»
sort :: [Int] -> [Int]
sort = foldr insert []
-- функция слияния двух упорядоченных списков
-- (без повторений элементов)
merge :: [Int] -> [Int] -> [Int]
merge [] l = l
merge l [] = l
merge lst1@(x1:l1) lst2@(x2:l2)
           | x1 < x2   = x1 : merge l1 lst2
           | x1 == x2  = x1 : merge l1 l2
           | otherwise = x2 : merge lst1 l2
```

```
-- функция слияния нескольких упорядоченных списков
-- (списка списков) в один список (без повторений).
mergeAll :: [[Int]] -> [Int]
mergeAll = foldr merge []
```

Наконец, теперь можно подсчитать число вершин в графе. Для этого надо составить упорядоченный список его вершин, последовательно обработав все списки смежности, а затем вычислить длину этого списка.

```
-- Упорядоченный список вершин графа
listVert :: Graph -> [Int]
listVert =
         mergeAll . (map (\(a, list) -> insert a (sort list)))
-- функция подсчета числа вершин в графе
count :: Graph -> Int
count = length . listVert
```

Функция получилась довольно сложной. Дело в том, что мы считали, что вершины могут иметь произвольные номера, и что в графе могут существовать вершины, для которых список смежности вообще не задан. Если считать, что любая вершина должна иметь в представлении графа свой список смежности (может быть, пустой), то количество вершин в графе можно определить, просто вычислив длину списка, задающего граф:

```
count = length
```

Теперь займемся основной функцией поиска путей в графе. Сначала определим функцию, которая для заданной вершины `a` находит ее список смежности, то есть список вершин, в которые из `a` ведет дуга.

```
-- функция поиска списка смежности
near :: Int -> Graph -> [Int]
near a g = case lookup a g of
                          Nothing -> []
                          Just lst -> lst
```

В этой функции для поиска списка смежности используется стандартная функция *Haskell* `lookup`. Она пытается в списке, заданным вторым аргументом, найти пару, первый элемент которой (ключ) совпадает с первым аргументом. Если такая пара найдена, то функция выдает второй элемент пары, если же нет – то значение `Nothing`. Вообще говоря, эта функция предназначена для поиска в так называемом «ассоциативном списке» (такие списки нам встретятся позже при анализе программ). Ассоциативный список – это список пар (ключ, значение). Смысл такого списка состоит в представлении некоторой таблицы, в которой «ключам» сопоставлены связанные с ними «значения». Функция `lookup` как раз и осуществляет поиск в таком списке.

Теперь определим функцию, которая проверяет, соединены ли в графе две вершины дугой. Для этого нужно найти в графе список смежности, соответствующий первой вершине, и определить, есть ли в этом списке вторая вершина. Соответствующая функция показана ниже.

```
-- функция проверки наличия дуги в графе
junc :: Int -> Int -> Graph -> Bool
junc a b g = elem b (near a g)
```

Путь из `a` в `b` длиной не более `n` существует в графе, если в нем либо вершины `a` и `b` соединены дугой, либо в списке вершин, смежных с `a`, найдется такая вершина `c`, что существует путь длины не более `(n-1)` из `c` в `b`. Отсюда сразу же получаем определение функции:

```
steps :: Int -> Int -> Int -> Graph -> Bool
steps n a b g = n > 0 && (junc a b g ||
                any (\c -> steps (n-1) c b g) (near a g))
```

Полностью вся программа показана в листинге У2.6.

Листинг У2.6. Поиск путей в графе

```
-- Задание. Структура графа задана списками смежности номеров
--          вершин, то есть списком, элементами которого
--          являются пары, состоящие из номера вершины и
--          списка вершин, инцидентных ей.
--          type Graph = [(Int, [Int])]
--          Написать функцию, которая проверяет,
--          существует ли в графе путь, соединяющий
--          вершины с двумя заданными номерами.

type Graph = [(Int, [Int])]

-- Функция вставки целого числа в упорядоченный список целых.
-- Аргументы: arg1 - вставляемый элемент,
--            arg2 - упорядоченный список.
-- Результат: объединенный упорядоченный список
--            (без повторений).
insert :: Ord a => a -> [a] -> [a]
insert n [] = [n]
insert n lst@(x:l) | n < x     = n : lst
                   | n == x    = lst
                   | otherwise = x : insert n l

-- Функция сортировки списка целых чисел
-- методом «простых вставок».
-- Если в списке есть одинаковые элементы,
-- то они остаются в единственном экземпляре.
-- Аргумент: исходный список.
-- Результат: отсортированный список без повторений.
sort :: Ord a => [a] -> [a]
sort = foldr insert []

-- Функция слияния двух упорядоченных списков
-- (без повторений элементов).
-- Аргументы - два упорядоченных списка.
-- Результат - список, объединяющий элементы исходных списков
```

```
--             без повторений.
merge :: Ord a => [a] -> [a] -> [a]
merge [] l = l
merge l [] = l
merge lst1@(x1:l1) lst2@(x2:l2)
                  | x1 < x2   = x1 : merge l1 lst2
                  | x1 == x2  = x1 : merge l1 l2
                  | otherwise = x2 : merge lst1 l2

-- функция слияния нескольких упорядоченных списков
-- (списка списков).
-- Аргумент - список упорядоченных списков.
-- Результат - упорядоченное объединение списков
--             без повторений.
mergeAll :: Ord a => [[a]] -> [a]
mergeAll = foldr merge []

-- Упорядоченный список вершин графа.
-- Аргумент - исходный граф.
-- Результат - упорядоченный список вершин графа.
listVert :: Graph -> [Int]
listVert =
         mergeAll . (map (\(v, list) -> insert v (sort list)))

-- функция подсчета числа вершин в графе.
-- Аргумент - исходный граф.
-- Результат - число вершин графа.
count :: Graph -> Int
count = length . listVert

-- Функция, выдающая список смежных вершин.
-- Аргументы: a - номер вершины,
--            g - исходный граф.
-- Результат: список вершин графа g, смежных с a.
near :: Int -> Graph -> [Int]
near a g = case lookup a g of
                           Nothing -> []
                           Just lst -> lst

-- Функция проверки наличия дуги в графе.
-- Аргументы: a, b - номера вершин,
--            g - исходный граф.
-- Результат: True, если в графе есть дуга, ведущая из a в b,
--            False в противном случае
junc :: Int -> Int -> Graph -> Bool
junc a b g = elem b (near a g)

-- Проверка, есть ли в графе путь длины не больше заданной.
-- Аргументы: n - предельное значение длины пути,
--            a, b - номера вершин,
```

```
--            g - исходный граф.
-- Результат: True, если в графе есть путь длины не больше n,
--                  ведущий из a в b,
--            False в противном случае.
steps :: Int -> Int -> Int -> Graph -> Bool
steps n a b g = n > 0 && (junc a b g ||
                any (\c -> steps (n-1) c b g) (near a g))

-- Проверка, есть ли в графе путь из заданной вершины a в b.
-- Аргументы: a, b - номера вершин,
--            g - исходный граф.
-- Результат: True, если в графе g есть путь, ведущий
--            из a в b, False в противном случае.
path :: Int -> Int -> Graph -> Bool
path a b g = a == b || steps (count g) a b g

-- Тестовый пример графа
gr :: Graph
gr = [(1, [5,20]),
      (5, [10]),
      (10, [25,12]),
      (20, [25]),
      (25, [5,1,12])]

-- Проверочная программа
main = [path 1 1 gr,
        path 1 12 gr,
        path 12 1 gr,
        path 5 1 gr]

-- Ожидаемый результат
-- [True, True, False, True]
```

Задание 7. Граф с конечным числом вершин, пронумерованных от единицы до `n`, представлен парой из количества вершин `n` и характеристической функции множества его дуг типа `Int -> Int -> Bool`, которая выдает `True`, если аргументы представляют собой номера вершин, соединенных дугой, и `False` в противном случае.

```
type Graph = (Int, Int -> Int -> Bool)
```

Составить функцию, которая проверяет, имеется ли в графе маршрут, соединяющий две заданные вершины.

Решение. Это задание совпадает с предыдущим, однако представление графа выбрано другим. Можно свести решение задачи к предыдущему, если написать функцию, которая преобразует граф в одном представлении в граф в другом представлении. Мы будем решать задачу другим способом, однако, для справки приведем все же и функцию преобразования графов, тем более что она оказывается совсем простой.

Действительно, довольно просто по заданному графу и номеру вершины `a` получить список смежных с `a` вершин. Для этого достаточно отфильтровать список всех вершин `[1..n]` функцией, задающей ребра графа:

```
listCont :: Graph -> Int -> [Int]
listCont (n, fc) a = [ b | b <- [1..n], fc a b ]
```

или еще короче:

```
listCont (n, fc) a = filter (fc a) [1..n]
```

Теперь функция преобразования графа в представление в виде списков смежности получается в одну строку:

```
convert :: Graph -> [(Int, [Int])]
convert g@(n, fc) = map (\a -> (a, listCont a g)) [1..n]
```

В полученном представлении все вершины имеют номера от `1` до `n`, и каждая вершина имеет свой список смежности, возможно, пустой. Для такого графа решение предыдущей задачи получается даже проще, чем это было представлено в листинге У2.6.

Мы, однако, не будем менять представление графа и попытаемся найти решение, использующее заданное представление. Для этого построим функцию, которая находит список всех достижимых из заданной вершины вершин графа. Как и в предыдущей задаче это можно сделать с помощью поиска в ширину.

Классический алгоритм поиска в ширину состоит из шагов, на каждом из которых имеется три множества вершин, которые условно можно назвать «черными», «серыми» и «белыми». Черные вершины – это те, которые уже были пройдены во время работы алгоритма. Если в процессе поиска алгоритм снова находит черную вершину, то она игнорируется. Серые вершины – это те, которые образуют «фронт волны», то есть те, списки смежности которых будут исследованы на очередном шаге алгоритма. Наконец, белые вершины – это вершины, которые еще не попали ни в список черных, ни в список серых вершин; это вершины, еще не исследованные алгоритмом.

В начале работы множество черных вершин пусто, множество серых вершин содержит единственную начальную вершину, остальные вершины – белые. На очередном шаге алгоритма исследуются все вершины, смежные с серыми. Если исследуемая вершина является серой или черной, то она просто пропускается, а если вершина – белая, то она перекрашивается в серый цвет и будет исследоваться на следующем шаге работы алгоритма. Все вершины, которые на предыдущем шаге работы алгоритма были серыми, становятся черными на следующем шаге.

Алгоритм заканчивает работу, когда множество серых вершин оказывается пустым. В этот момент множество черных вершин как раз и будет составлять множество всех достижимых из начальной вершины вершин графа.

Для решения задачи будем представлять множества вершин графа их характеристическими функциями:

```
type VertSet = Int -> Bool
```

Для работы с такими множествами нам понадобятся функции объединения и разности множеств и проверки того, является ли множество пустым. Первые две функции мы уже представляли в тексте книги, а последнюю функцию в общем случае написать невозможно, но для нашего представления графа достаточно лишь проверить, что ни одно из чисел от 1 до n не принадлежит множеству. Итак, имеем следующие функции:

```
(<+>) :: VertSet -> VertSet -> VertSet -- объединение множеств
s1 <+> s2 = \a -> s1 a || s2 a
(<->) :: VertSet -> VertSet -> VertSet -- разность множеств
s1 <-> s2 = \a -> s1 a && not (s2 a)
isEmpty :: Int -> VertSet -> Bool
-- первый аргумент - число вершин графа
isEmpty n s = not (any s [1..n])
```

Заметим еще, что множество, состоящее из единственного элемента a можно представить функцией (== a). Теперь мы готовы описать алгоритм поиска в ширину, который по заданному графу и начальной вершине выдает множество всех достижимых из нее вершин графа. Решение представленио функцией breadthFirst, имеющей в качестве аргументов исходный граф и множества черных и серых вершин. Функция осуществляет один шаг алгоритма и обращается рекурсивно к себе для выполнения следующих шагов, заканчивая работу в тот момент, когда множество серых вершин становится пусто. В определении функции используется еще две вспомогательных функции, первая из которых – toList – конвертирует множество вершин в список, а вторая производит объединение списка множеств в одно множество:

```
toList :: Int -> VertSet -> [Int]
toList n s = filter s [1..n]
union :: [VertSet] -> VertSet
union = foldr (<+>) (\a -> False)
```

Полное решение задачи, включающее все описанные ранее функции, а также функцию breadthFirst и собственно функцию, решающую задачу, представлено в листинге У2.7.

Листинг 2.7. Поиск в ширину в графе, представленном характеристическими функциями смежности.

```
-- Задание. Граф с конечным числом вершин представлен
--          парой из количества вершин и функции типа
--          Int -> Int -> Bool, которая выдает True,
--          если аргументы представляют собой номера
--          вершин, соединенных ребром.
type Graph = (Int, Int -> Int -> Bool)
--          Составить функцию, которая проверяет, имеется ли
```

```
--          в графе маршрут, соединяющий две заданные вершины.

-- Представление множества характеристической функцией
type VertSet = Int -> Bool

(<+>) :: VertSet -> VertSet -> VertSet -- объединение множеств
s1 <+> s2 = \a -> (s1 a) || (s2 a)
(<->) :: VertSet -> VertSet -> VertSet -- разность множеств
s1 <-> s2 = \a -> (s1 a) && (not (s2 a))

-- пустое множество
empty :: VertSet
empty = \u -> False

-- проверка пустоты ограниченного множества
isEmpty :: Int -> VertSet -> Bool
-- первый аргумент - число вершин графа
isEmpty n s = not (any s [1..n])

-- список элементов множества
toList :: Int -> VertSet -> [Int]
toList n s = filter s [1..n]

-- объединение множеств из списка
union :: [VertSet] -> VertSet
union = foldr (<+>) empty

-- обход графа в ширину.
-- Аргументы: g - исходный граф;
--            blacks - множество "черных" вершин
--            greys - множество "серых" вершин
-- Результат: множество вершин, достижимых из greys
breadthFirst :: Graph -> VertSet -> VertSet -> VertSet
breadthFirst g@(n, fc) blacks greys
   | isEmpty n greys = blacks
   | otherwise = breadthFirst g newBlacks
                              (newFront <-> newBlacks)
      where newBlacks = blacks <+> greys
            newFront = union (map fc (toList n greys))

-- Функция проверки существования пути в графе.
-- Аргументы: g - исходный граф;
--            a, b - начальная и конечная вершины
path :: Graph -> Int -> Int -> Bool
path g a b = (breadthFirst g empty (==a)) b

-------------------------------------------------------------
-- список дуг тестового графа
edges :: [(Int, Int)]
edges = [(1, 2), (2, 3), (1, 6), (1, 4),
```

```
         (3, 1), (4, 5), (4, 3), (6, 5)]
-- тестовый граф
test :: Graph
test = (6, f)
       where f a b = (a,b) `elem` edges

-- ожидаемый результат: [True, True, False, True]
main = [path test 1 6, path test 5 5,
        path test 6 1, path test 4 2]
```

Задания для самостоятельной работы

Задание 1. Дерево задано с помощью следующего описания структуры данных.

```
data Tree a = Node a [Tree a]
```

то есть дерево представляет собой корневой узел, содержащий некоторое значение произвольного типа `a` и список поддеревьев. Написать функцию, проверяющую, что дерево является пирамидой, то есть значение в каждом из его узлов меньше, значений, хранящихся в поддеревьях этого узла.

Задание 2. В условиях предыдущего задания написать функцию, которая проверяет, верно ли, что значения, хранящиеся в корнях поддеревьев каждого узла, упорядочены в списке поддеревьев по возрастанию.

Задание 3. Структура графа задана списками смежности номеров вершин, то есть списком, элементами которого являются пары, состоящие из номера вершины и списка вершин, инцидентных ей.

```
type Graph = [(Int, [Int])]
```

Написать функцию, которая выдает длину кратчайшего маршрута между двумя заданными вершинами. Длиной считать количество вершин, встретившихся в данном маршруте. Если маршрута между вершинами не существует, то функция должна выдавать ноль.

Задание 4. В условиях предыдущей задачи написать функцию, которая выдает кратчайший маршрут (в виде списка номеров вершин), соединяющий две заданные вершины в графе. Если такого маршрута не существует, то функция должна выдавать пустой список. Если номера первой и последней вершины совпадают, то функция выдаст список из единственной вершины.

Задание 5. Неориентированный граф с конечным числом вершин представлен парой из количества вершин и функции типа `Int -> Int -> Bool`, которая выдает `True`, если аргументы представляют собой номера вершин, соединенных ребром. Функция заведомо коммутативная, то есть если вершина *a* связана с вершиной *b*, то и вершина *b* связана с вершиной *a*.

```
type Graph = (Int, Int -> Int -> Bool)
```

Написать функцию, которая проверяет, является ли граф двудольным.

Задание 6. В условиях предыдущей задачи написать функцию, которая по номерам двух вершин, соединенных некоторым ребром, проверяет, является ли это ребро мостом в графе.

2.3. Ленивые вычисления

До сих пор мы почти не рассматривали вопрос о том, как передаются аргументы в вызываемую функцию. Тем не менее, вопрос этот чрезвычайно важен. Традиционно при описании языков программирования вопросу о способах передачи аргументов уделяется особое внимание. Например, в языке *C#* параметры бывают входными, выходными и входно-выходными. В языке *Паскаль* имеются два способа передачи аргументов: передача аргументов *по значению* и *по ссылке* (в языке *Паскаль* второй способ обычно называют передачей *аргументов-переменных*, однако, мы используем более традиционный для теории языков программирования термин). Есть в языке еще и третий способ передачи аргумента – передача процедуры или функции, но разные реализации языка подходят к вопросам передачи процедур в качестве аргументов по-разному; в наиболее популярной реализации *Паскаля* для операционной системы Windows – реализации фирмы *Borland* – этот способ передачи аргументов сведен к передаче аргумента по значению, когда таким значением является функция или процедура. В языке *Java* есть только один способ передачи аргументов в функцию – передача аргумента *по значению*, однако, если таким аргументом является указатель на объект, то изменение этого объекта внутри функции приводит к «побочному эффекту» - изменению переданного значения.

Давайте рассмотрим способы передачи аргументов в *Паскале* немного подробнее.

Если в процедуре или функции аргумент объявлен аргументом-значением (это наиболее частый способ передачи аргументов простых типов), то при вызове этой процедуры (или функции) фактическим аргументом может быть произвольное выражение *совместимое по присваиванию* с формальным аргументом. В момент вызова выражение вычисляется, и полученное при вычислении значение передается в процедуру. В дальнейшем процедура работает со своим аргументом как с локальной переменной, объявленной внутри этой процедуры.

Передача аргумента-переменной происходит несколько иначе. Во-первых, фактическим аргументом должна быть переменная (возможно, компонент сложной структуры или массива), тип которой в точности совпадает с объявленным типом формального аргумента. В момент вызова происходит *отождествление* фактического и формального аргументов. В дальнейшем процедура в действительности работает непосредственно с

переданной ей переменной. На самом деле отождествление фактического и формального аргументов происходит путем передачи в процедуру *ссылки* на переданную переменную (или, говоря техническим языком, *адреса*), поэтому такой способ передачи аргументов и называют передачей аргумента по ссылке.

Помимо двух описанных способов передачи аргументов известны и некоторые другие способы. Так, например, в языках семейства *Алгол*, популярных в 60-х и 70-х годах 20-го века, существовал способ передачи аргумента *по наименованию*, при котором в момент вызова процедуры фактический аргумент вообще не вычислялся, а вместо этого в процедуру передавался «рецепт» вычисления адреса или значения фактического параметра. По существу, неявно создавалась функция, которая содержала в себе программу вычисления аргумента, и именно эта функция и передавалась в качестве аргумента в процедуру. В дальнейшем при каждом обращении изнутри процедуры к ее аргументу эта функция вызывалась, вычисляя фактическое значение или адрес аргумента. Если случалось, что в некотором конкретном вызове процедуры обращений к аргументу не было вовсе, то и вычислений фактического аргумента не происходило. Если же обращений к аргументу было несколько, то работа неявной функции тоже происходила несколько раз.

Если фактический аргумент представлял собою выражение, требующее для своего вычисления значительного времени, то, с одной стороны, можно было рассчитывать на некоторую экономию времени, если фактически обращения к аргументу не было. С другой стороны, если обращений к аргументу было несколько, то многократное вычисление одного и того же сложного выражения могло, напротив, привести к замедлению работы программы.

Хорошим компромиссом является способ передачи аргументов *по необходимости*. В этом способе, как и при передаче аргумента по наименованию, фактически в процедуру передается функция для вычисления значения или адреса фактического аргумента. Однако, при первом же обращении к аргументу изнутри процедуры вычисленное однажды значение (или адрес) запоминаются. При последующих обращениях к этому же аргументу используется уже вычисленное однажды значение. Передача аргументов по необходимости является хорошим компромиссом между универсальностью передачи аргументов по наименованию и эффективностью передачи аргументов по значению. Платой за такое удобство является относительная сложность реализации, при которой передаваемый аргумент превращается в функцию вычисления значения («запроцедуривается» в терминологии языка Алгол-68).

Если все аргументы во все процедуры передаются по необходимости, то такой способ работы называют *ленивыми вычислениями*. Такой термин вполне оправдан, поскольку при нем никакие

аргументы не вычисляются до тех пор, пока реально не понадобятся их значения. В языке *Haskell*, в основном, используются ленивые вычисления.

Давайте рассмотрим несколько примеров, в которых результат работы программы зависит от того, принята ли схема ленивых вычислений с передачей аргументов по необходимости, или схема *энергичных* вычислений с передачей аргументов по значению (понятно, что поскольку в языке *Haskell* нет переменных, то и никакой передачи аргументов по ссылке быть не может). В качестве первого примера заново определим и подробно рассмотрим определение функции *логического "или"*, которую мы активно использовали в предыдущем разделе при программировании регулярных выражений. Уравнение, определяющее работу функции может выглядеть так:

```
a `cor` b = if a then True else b
```

(обозначение операции `cor` означает «условное или» – «conditional or»). Попробуем вычислить результат вызова этой функции, передав ей в качестве аргументов два логических выражения:

```
(x == 0) `cor` (y / x < 5)
```

Если в текущем контексте `x=2` и `y=6`, то вычисление происходит одинаково, независимо от способа передачи аргументов в функцию `cor`. Первый аргумент при вычислении в данном контексте выдаст значение `False`, а второй аргумент – значение `True`. Результатом вычтсления всего выражения будет, соответственно, `True`. Однако, при `x=0` и `y=1` вычисление второго аргумента приведет к ошибке, несмотря на то, что первый аргумент будет иметь значение `True`, и ветвь вычислений `else`, в которой происходит обращение ко второму аргументу `b`, вообще не будет выполняться, так что от значения второго аргумента результат работы функции не зависит. Но если аргументы будут передаваться в функцию по значению, то вычисление значений аргументов будет произведено еще до начала работы функции, так что в данном контексте программа закончится аварийно. Если же аргументы будут передаваться по наименованию или по необходимости, то до обращения ко второму аргументу дело не дойдет, поскольку функция после вычисления значения первого аргумента сразу же выдаст значение `True`. Собственно говоря, это тот самый эффект, на который мы и рассчитывали при написании этой функции, и который использовался нами в предыдущем разделе при определении регулярных языков. Условное «или» (по МакКарти) именно тем и отличается от традиционного логического «или», что значение второго аргумента не будет вычисляться, если при вычислении значения первого аргумента будет получено значение `True`.

В языке *Haskell* по умолчанию принята схема передачи аргументов по необходимости, так что используемые нами функции логических «или» и «и» будут работать так, как задумано. Поскольку никакие выражения в

Haskell, вообще говоря, не могут обладать побочным эффектом (вычисление или невычисление выражения никак не влияет на глобальный контекст вычислений), то применение ленивой схемы вычислений не может изменить результат работы программы по сравнению с энергичной схемой вычислений, конечно, если результат вообще может быть вычислен. Разница в поведении программ при использовании энергичной и ленивой схем вычислений будет проявляться в тех случаях, когда энергичная схема вычислений приводит к «зацикливанию» или аварийному завершению работы программы, так что результат вообще не может быть вычислен.

Все рассмотренные нами ранее программы будут выдавать один и тот же результат независимо от применяемой схемы вычислений. Исключение составляет последний пример с определением языков с помощью регулярных выражений. Если функции условных логических операций будут работать по энергичной схеме передачи аргументов, то при анализе строки на принадлежность языку может происходить зацикливание работы программы.

Говорят, что функция *строгая* по какому-нибудь из ее аргументов, если для ее вычисления требуется обязательное вычисление этого ее аргумента. Если функция строгая по всем своим аргументам, то говорят просто о *строгой* функции. Например, операция логического «или» `(||)` – строгая по первому аргументу и нестрогая по второму. А вот все встроенные арифметические функции языка *Haskell* (сложение, вычитание и др.) – строгие. Если известно, что функция строгая по какому-нибудь из аргументов, то соответствующий фактический аргумент можно передавать по значению, поскольку такое вычисление обычно реализуется эффективнее, чем передача значения по необходимости. Если же функция нестрогая по какому-то из аргументов, то передача этого ее аргумента по значению опасна: при вычислении аргумента программа может зациклиться или не сможет выполнить какую-то операцию, в то время как значение этого аргумента может и не потребоваться для работы функции.

Как мы уже упомянули, встроенные арифметические и многие другие встроенные функции языка *Haskell* – строгие. Это и понятно, поскольку в противном случае вычисления вообще никогда не приводили бы ни к какому результату. Однако имеется также множество других встроенных функций, таких как функции обработки списков, встроенные функции высших порядков и другие, которые являются нестрогими по одному или всем своим аргументам. Нестрогим является также конструктор списков `(:)`. Конструкторы вообще все и всегда нестрогие по всем своим аргументам. Это позволяет записывать и обрабатывать такие выражения, которые вообще никогда не могли бы быть вычислены до конца при энергичной схеме вычислений. Такова, например, техника

обработки потенциально бесконечных списков. В языке *Haskell* вполне корректным будет следующее определение списка из целых:

```
ones :: [Integer]
ones = 1:ones
```

Конечно, если вы попробуете найти сумму всех элементов такого списка или обработать его иным способом, в котором потребуются *все* элементы списка, то программа зациклится в безнадежной попытке вычислить все единицы в этом списке. Однако, функция, которая суммирует лишь первые несколько элементов, вполне может быть применена к такому «бесконечному» списку. Действительно, пусть имеется следующая функция, вычислющая сумму первых `n` элементов целочисленного списка:

```
summ :: Integer -> [Integer] -> Integer
summ 0 _ = 0
summ n (x:s) = x + summ (n-1) s
```

Попробуем теперь вычислить значение `summ 4 ones`, учитывая, что вычисления в *Haskell* – ленивые. Мы получим следующую цепочку преобразований.

```
summ 4 ones                          ⇒
    (надо произвести сопоставление с образцом. Первое
     уравнение, очевидно, не подходит, а во втором для
     успешного сопоставления надо произвести подстановку
     определяющего выражения для ones)
summ 4 (1:ones)                      ⇒
    (сопоставление с образцом произведено,
     используем второе уравнение)
1 + summ 3 ones                      ⇒
    (функция арифметического сложения строгая, так что ее
     второй аргумент необходимо вычислить)
1 + (1 + summ 2 ones)                ⇒
1 + (1 + (1 + summ 1 ones))          ⇒
1 + (1 + (1 + (1 + summ 0 ones)))    ⇒
    (шаги по вычислению второго аргумента производились точно
     так же, как и первые два шага; теперь, однако,
     сопоставление с первым уравнением произойдет успешно без
     вычисления второго аргумента функции summ, и мы получим)
1 + (1 + (1 + (1 + 0)))              ⇒
4
```

Итак, оказалось, что работа с бесконечным списком окончилась вполне успешно. Это произошло потому, что на самом деле нам не потребовался бесконечный список; после вычисления первых четырех элементов остальные не понадобились. Подобная техника может применяться в *Haskell* всегда, когда это удобно. Следующая функция

выдает в качестве результата бесконечный список последовательных целых значений, начинающийся с заданного значения.

```
fromInt :: Integer -> [Integer]
fromInt n = n : fromInt (n+1)
```

Теперь вызов этой функции `fromInt 1` дает в результате весь натуральный ряд. Только не надо просить найти последний элемент в этом списке! Это бесполезно; соответствующая программа просто зациклится. Вообще, бесконечную арифметическую прогрессию, подобную только что полученной, можно получить в *Haskell* и еще проще, используя обычное синтаксическое обозначение для арифметической прогрессии `[1..]`. От конечной прогрессии это обозначение отличается тем, что верхняя граница прогрессии не указана. Бесконечные списки можно использовать везде, где можно использовать и обычные конечные списки, до тех пор, пока не потребуется вычисление всех его элементов. Вот как, например, получить сумму первых десяти квадратов натуральных чисел с использованием нашей функции `summ`:

```
summ 10 [x*x | x <- [1..]]
```

Нетрудно получить и более сложные списки, чем список квадратов. Например, ранее мы строили множество простых чисел, не превышающих заданного значения с помощью алгоритма решета Эратосфена. Функция получилась не очень простой, однако, оказывается, что при работе с бесконечными списками тот же самый алгоритм можно реализовать значительно проще, чем представляя множество чисел характеристической функцией. Основой для реализации этого алгоритма может послужить конструкция, выкидывающая из списка все числа, кратные заданному числу `n`: `[x | x <- s, x `mod` n /= 0]`. Тогда алгоритм решета Эратосфена может быть выражен с помощью рекурсивной функции просеивания `sieve`:

```
sieve :: [Integer] -> [Integer]
sieve (n:s) = n : sieve [x | x <- s, x `mod` n /= 0]
```

Функция просеивания берет первый элемент некоторого, вообще говоря, бесконечного списка, и выкидывает из хвоста этого списка все элементы, делящиеся на первый элемент. После этого к результату фильтрации снова применяется функция просеивания, а первый элемент вновь присоединяется к результату этого просеивания.

Теперь список всех простых чисел получается применением функции просеивания к бесконечному списку чисел, начиная с двойки: `primes = sieve [2..]`. Отметим, что функция работает достаточно эффективно, последовательно удаляя из списка всех натуральных чисел те из них, которые делятся хотя бы на одно из уже полученных простых чисел.

Помимо бесконечной арифметической прогрессии имеется еще довольно много стандартных функций, с помощью которых можно построить бесконечные списки из конечных списков или отдельных значений. Приведем некоторые из них.

Функция `repeat` строит бесконечный список, содержащий один и тот же элемент в бесконечном числе экземпляров. Например, выражение `(repeat 1)` задает бесконечный список из единиц: `[1,1,1,...]`.

Функция `cycle` строит бесконечный список, повторяя бесконечно один и тот же фрагмент, заданный конечным списком – аргументом функции. Например, выражение `(cycle [1..3])` задает бесконечный список `[1,2,3,1,2,3,1,2,...]`.

Функция высшего порядка `iterate` строит бесконечный список из начального элемента, последовательно применяя заданную функцию к элементам строящегося списка. Каждый следующий элемент списка получается применением этой функции к предыдущему. Например, бесконечную геометрическую прогрессию со знаменателем 2 можно построить с помощью применения функции `iterate` следующим образом: `iterate (*2) 1`.

Еще один простой, но интересный пример – это получение списка, представляющего собой последовательность чисел Фибоначчи. Если мы уже имеем последовательность чисел Фибоначчи в виде бесконечного списка (обозначим его `fib`), то прежде всего образуем еще один бесконечный список, `fib0`, приписав ноль к началу последовательности чисел Фибоначчи: `fib0 = (0:fib)`. Теперь заметим, что если сложить почленно две наши последовательности `fib` и `fib0`, то в результате опять получится последовательность чисел Фибоначчи, только без первого члена. Операция почленного сложения списков является частным случаем более общей стандартной функции над списками `zipWith`. Эта функция высшего порядка выполняет заданную операцию над элементами двух списков почленно, формируя третий список из получающихся при этом результатов. Определение функции `zipWith` можно записать следующим образом:

```
zipwith  :: (a -> b -> c) -> [a] -> [b] -> [c]
zipwith _ [] []             = []
zipwith f (x1:s1) (x2:s2)  = (f x1 x2) : zipWith f s1 s2
```

Тогда функция почленного сложения двух списков - это просто `zipWith (+)`. Теперь последовательность чисел Фибоначчи можно будет сформировать с помощью следующей несложной конструкции:

```
fib = 1 : zipwith (+) fib (0 : fib)
```

Эту строчку можно прочитать так: "последовательность чисел Фибоначчи получается, если приписать единицу к результату почленного сложения самой этой последовательности и той же последовательности, к

которой приписан в начале нулевой элемент". Данное определение рекурсивно определяет последовательность Фибоначчи и действительно порождает бесконечный список чисел Фибоначчи, но, конечно, только благодаря ленивым вычислениям в программах на языке *Haskell*.

Стоит упомянуть еще одну интересную технику написания программ для работы с бесконечными списками. В этой технике, называемой *завязыванием узлов*, программа сначала представляется в виде схемы, на которой бесконечные списки представлены в виде *потоков* и изображаются на схеме стрелочками. Так, например, упомянутая выше последовательность чисел Фибоначчи будет представлена в виде потока, изображенного на рисунке 2.1

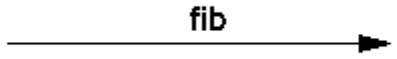

Рис. 2.1. Изображение потока

Потоки могут обрабатываться почленно функциями, подобными показанной выше функции `zipWith (+)`, при этом из одного или нескольких потоков образуется новый поток. Например, если потоки нужно почленно складывать, то на схеме изображается элемент, выполняющий сложение, в виде овала, содержащего обозначение операции сложения. Потоки-операнды входят в этот овал, а поток-результат – выходит. Часть схемы, содержащая такую обработку, будет выглядеть так, как показано на рисунке 2.2:

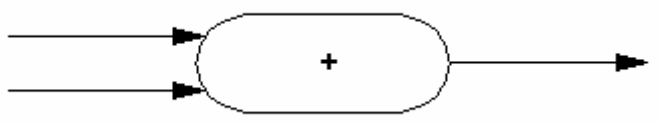

Рис. 2.2. Сложение потоков

Наконец, если к потоку надо приписать дополнительный первый элемент (это соответствует применению конструктора списков `(:)`), то на схеме рисуют треугольник, входами в который являются поток и добавляемый в него элемент, а выходом будет результирующий поток с добавленным первым элементом. Так, например, приписывание нуля к потоку, изображающему последовательность чисел Фибоначчи, на схеме будет выглядеть так, как показано на рисунке 2.3.

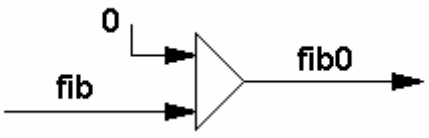

Рис. 2.3. Приписывание элемента к потоку

Описанное выше взаимодействие потоков, в результате которого получается последовательность чисел Фибоначчи, можно теперь изобразить в виде схемы, показанной на рисунке 2.4.

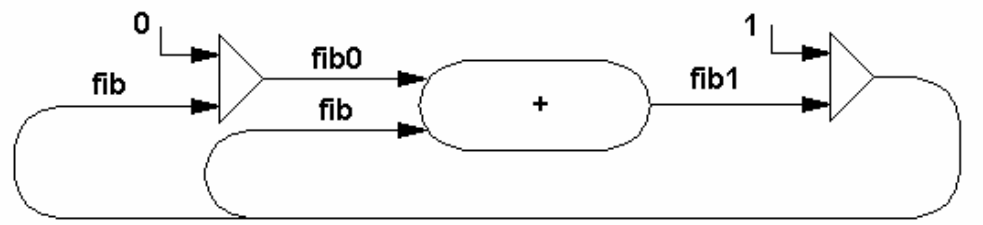

Рис. 2.4. Схема потоков для вычисления последовательности чисел Фибоначчи

Теперь программу на языке *Haskell*, реализующую изображенную схему взаимодействия потоков, можно получить автоматически. Для этого нужно просто описать все узлы данной схемы в виде серии взаимно-рекурсивных определений. В этих определениях каждый поток получается как результат применения функций к другим потокам и отдельным значениям. Например, поток, выдающий последовательность чисел Фибоначчи, описывается следующей программой, которая в точности следует нарисованной схеме:

```
fib where
    fib0 = 0:fib
    fib1 = zipwith (+) fib0 fib
    fib = 1:fib1
```

Примеры решения задач

В заданиях по этой теме используются «бесконечные» структуры данных, обработка которых возможна при вычислениях по схеме «ленивых» вычислений.

Задание 1. Сформировать список разностей соседних элементов в последовательности квадратов натуральных чисел. Убедиться, что эта последовательность представляет собой список последовательных нечетных чисел.

Решение. Список квадратов целых чисел можно получить очень просто с помощью конструкции `[x*x | x <- [1..]]`. Искомую последовательность можно получить, если составить список почленных разностей между элементами этого списка и того же списка без первого члена. Отсюда сразу же следует решение в одну строчку:

```
diff = zipwith (-) sqrs (1:sqrs)
           where sqrs = [x*x | x <- [2..]]
```

Задание 2. Близнецами называется пара простых чисел, разность между которыми равна двум. Составить список первых 50 пар близнецов.

Решение. Не очень ясно, какой диапазон чисел необходимо исследовать, поскольку распределение близнецов среди множества целых чисел, да и распределение самих простых чисел среди всех натуральных

чисел, неизвестно. Поэтому естественным будет строить потенциально бесконечные списки, из которых затем можно будет выбрать необходимое число элементов.

Список всех простых чисел может быть построен с помощью алгоритма «решета Эратосфена», программа для которого была приведена в основном тексте книги.

```
sieve :: [Integer] -> [Integer]
sieve (n:s) = n : sieve [x | x <- s, x `mod` n /= 0]
primes = sieve [2..]
```

Теперь можно вычислить все пары соседних элементов полученного списка простых чисел и отобрать те из них, разность между которыми равна двум.

```
pairs :: [Integer] -> [(Integer, Integer)]
pairs (x:s@(y:_)) = (x,y) : pairs s
twins s = [(x,y) | (x,y) <- s, y-x == 2]
```

Первые 50 элементов полученного потенциально бесконечного списка можно отобрать с помощью стандартной функции `take`, которая выбирает заданное количество первых элементов списка. Полная программа представлена в листинге 2.8.

Листинг 2.8. Список близнецов.

```
-- Задание. Близнецами называется пара простых чисел,
--          разность между которыми равна двум.
--          Составить список первых 50 пар близнецов.

-- Список простых чисел, полученный методом "решета".
sieve :: [Integer] -> [Integer]
sieve (n:s) = n : sieve [x | x <- s, x `mod` n /= 0]

primes :: [Integer]
primes = sieve [2..]

-- Список пар последовательных элементов
pairs :: [Integer] -> [(Integer, Integer)]
pairs (x:s@(y:_)) = (x,y) : pairs s

-- Список близнецов
twins :: [(Integer, Integer)]
twins = [(x,y) | (x,y) <- pairs primes, y-x == 2]

-- Основная программа выводит первые 50 пар близнецов
main = take 50 twins
-- ожидаемый результат:
--
[(3,5),(5,7),(11,13),(17,19),(29,31),(41,43),...,(1487,1489)]
```

Задание 3. Бесконечная упорядоченная последовательность целых чисел составлена из степеней двойки и чисел вида $2 \cdot 3^n$. Составить программу для вычисления n-го члена этой последовательности.

Решение. Бесконечные списки, содержащие последовательные степени двойки и последовательные числа вида $2 \cdot 3^n$ составить очень просто, если воспользоваться стандартной функцией `iterate`. После этого можно слить оба списка в один упорядоченный список, воспользовавшись алгоритмом слияния упорядоченных списков. Таким образом получается очень простая программа, решающая поставленную задачу, которая приведена полностью в листинге 2.9.

Листинг 2.9. Упорядоченная последовательность чисел специального вида.

```
-- Задание. Бесконечная упорядоченная последовательность
--          целых чисел составлена из степеней двойки
--          и чисел вида 2*(3^n). Составить функцию
--          для вычисления n-го члена последовательности.

-- Формирование списков степеней двойки и чисел вида 2*(3^n).
powers2 = iterate (2*) 2  -- первый элемент – 2^1
powers3 = iterate (3*) 6  -- первый элемент – 2*(3^1)

-- Слияние упорядоченных списков
merge :: Ord a => [a] -> [a] -> [a]
merge ls1@(x:t1) ls2@(y:t2) = if x < y then x:(merge t1 ls2)
                                       else y:(merge ls1 t2)

-- Упорядоченный список чисел вида (2^n) и (2*(3^n)).
list :: Integer
list = merge powers2 powers3

-- Функция для решения данной задачи
member :: Int -> Integer
member n = list !! n

-- Тестовая функция строит первые 17 членов
-- искомой последовательности.
main = take 17 list
-- Ожидаемый результат:
-- [2,4,6,8,16,18,32,54,64,128,162,256,486,512,1024,1458,2048]
```

Задание 4. «Бесконечное» двоичное дерево задано следующим описанием структуры данных.

```
data BinTree a = Node a (BinTree a) (BinTree a)
```

Построить дерево, содержащее все натуральные числа в узлах, соответствующих порядку естественного обхода дерева сверху вниз. Построенное дерево, очевидно, является *пирамидой* (напомним, что пирамидой или *кучей* называется дерево, в каждом узле которого находится значение, меньшее, чем во всех узлах-потомках этого узла).

Написать программу удаления из этой пирамиды минимального значения. Вывести значения, которые будут находиться на четвертом уровне дерева после удаления первых пяти узлов.

Решение. Если некоторый узел искомого дерева содержит значение n, то его непосредственные потомки должны содержать значения $(2n)$ и $(2n+1)$. Отсюда сразу получаем программу построения искомого дерева.

```
tree :: Integer -> BinTree Integer
tree n = Node n (tree (2*n)) (tree (2*n + 1))
```

Удаление минимального узла из пирамиды производится с помощью алгоритма «протаскивания», при котором значение в каждом из удаляемых узлов заменяется на минимальное из двух его непосредственных потомков, после чего тот же алгоритм применяется рекурсивно к этому потомку. Алгоритм можно запрограммировать «напрямую»:

```
remove :: Ord a => BinTree a -> BinTree a
remove (Node n t1@(Node n1 _ _) t2@(Node n2 _ _)) =
               if n1 < n2 then Node n1 (remove t1) t2
                          else Node n2 t1 (remove t2)
```

Вывод содержимого i-го уровня дерева может быть выполнен с помощью следующей рекурсивной функции (вспомогательная функция `level'` введена для повышения эффективности работы добавлением накапливающего аргумента).

```
level :: Integer ->  BinTree a -> [a]
level' :: [a] -> Integer ->  BinTree a -> [a]
level = level' []
level' s 1 (Node n _ _) = n:s
level' s i (Node _ t1 t2) =
                    level' (level' s (i-1) t2)) (i-1) t1)
```

Полная программа, выполняющая все необходимые вычисления, приведена в листинге 2.10.

Листинг 2.10. Бесконечная пирамида.

```
-- Задание. «Бесконечное» двоичное дерево задано следующим
--          описанием структуры данных.
data BinTree a = Node a (BinTree a) (BinTree a)
--          Построить пирамиду, содержащую все натуральные
--          числа в узлах, соответствующих порядку
--          естественного обхода дерева сверху вниз.
--          Написать программу удаления из этой пирамиды
--          минимального числа.  Вывести значения,
--          которые будут находиться на четвертом уровне
--          дерева после удаления первых пяти узлов.

-- Построение исходной пирамиды
tree :: Integer -> BinTree Integer
tree n = Node n (tree (2*n)) (tree (2*n + 1))
```

```
-- Удаление минимального узла с вершины пирамиды
remove :: Ord a => BinTree a -> BinTree a
remove (Node n t1@(Node n1 _ _) t2@(Node n2 _ _)) =
             if n1 < n2 then Node n1 (remove t1) t2
                        else Node n2 t1 (remove t2)

-- Повторение применения функции заданное число раз
repeatN :: Int -> (a -> a) -> a -> a
repeatN 0 _ seed = seed
repeatN n f seed = repeatN (n-1) f (f seed)

-- Вывод содержимого i-го уровня дерева
level :: Integer ->  BinTree a -> [a]
level' :: [a] -> Integer ->  BinTree a -> [a]
level = level' []
level' s 1 (Node n _ _) = n:s
level' s i (Node _ t1 t2) =
           level' (level' s (i-1) t2) (i-1) t1

-- Основная программа выводит искомый результат –
-- список узлов четвертого уровня в пирамиде после
-- удаления из нее первых пяти чисел.
main = level 4 (repeatN 5 remove (tree 1))
-- ожидаемый результат: [16,18,20,11,24,13,28,15]
```

Задание 5. «Бесконечная» матрица содержит все натуральные числа в «диагональном» порядке. Первые несколько элементов матрицы выглядят следующим образом:

```
 1  2  4  7 11 16...
 3  5  8 12 17...
 6  9 13 18...
10 14 19...
```

Построить эту матрицу в виде бесконечного списка бесконечных списков и вычислить сумму первых 10 элементов 5-го столбца матрицы.

Решение. Если построить первую строку матрицы, то каждая следующая строка получается из предыдущей, если отбросить из нее первый элемент, а затем все элементы увеличить на единицу. Таким образом, если мы имеем строку матрицы `s`, то следующая строка получается из нее с помощью выражения `map (+1) (tail s)`. Вся матрица может быть получена из первой строки `s1` применением к ней функции итерации

```
iterate ((map (+1)) . tail) s1
```

Первую строку матрицы можно построить примерно так же, как мы строили последовательность чисел Фибоначчи. Действительно, если почленно добавить к этой строке числа из натурального ряда чисел `[1..]`,

то в результате опять получится та же строка без первого члена. Отсюда сразу получается формула

```
s1 = 1 : (zipWith (+) s1 [1..])
```

Для построенной матрицы `m` сумма первых `n` элементов ее `j`-го столбца может быть получена с помощью функции

```
sum :: [[Integer]] -> Int -> Int
sum _ 0 _ = 0
sum (h:t) n j = (h !! j) + sum t (n-1) j
```

Полностью программа для решения задания приведена в листинге У2.11.

Листинг 2.11. Бесконечная диагональная матрица.

```
-- Задание. «Бесконечная» матрица содержит все натуральные
--          числа в «диагональном» порядке.
--          Построить эту матрицу в виде бесконечного списка
--          бесконечных списков и вычислить
--          сумму первых 10 элементов 5-го столбца матрицы.
type Matrix = [[Integer]]

-- Построение исходной матрицы.
matrix :: Matrix
matrix = iterate ((map (+1)) . tail) s1
         where s1 = 1 : (zipWith (+) s1 [1..])

-- суммирование первых n элементов j-го столбца.
sum :: Matrix -> Int -> Int -> Integer
sum _ 0 _ = 0
sum (h:t) n j = (h !! j) + sum t (n-1) j

-- основная функция строит сумму первых 10 элементов
-- 5-го столбца и еще несколько других сумм в тестовых целях.
main = [sum matrix 10 5, sum matrix 3 3, sum matrix 4 2]
-- ожидаемый результат: [595, 37, 44]
```

Задания для самостоятельной работы

Задание 1. Бесконечная упорядоченная последовательность целых чисел без повторений составлена из всех квадратов, кубов и факториалов натуральных чисел. Составить программу для вычисления *n*-го члена этой последовательности.

Задание 2. Найти первые несколько простых чисел вида 2^n+1.

Задание 3. Составить (бесконечный) список частичных сумм ряда, представляющего собой разложение числа *e*, полученное подстановкой единицы в ряд Тейлора для экспоненты.

Задание 4. Бесконечная пирамида (см. задание 4 данной темы) содержит в узлах все последовательные нечетные числа. Составить программу для построения такой пирамиды. Какие числа будут

содержаться на первых четырех уровнях этой пирамиды после добавления в нее первых пяти четных чисел? Добавление можно производить в пирамиду с помощью алгоритма «протаскивания».

Глава 3. Лямбда-исчисление

3.1. Формальные теории в программировании

Функциональные языки программирования появились в качестве средства для написания программ, не содержащих в явном виде понятий ячеек памяти для хранения значений (переменных) и последовательности вычислений как процесса изменения состояния памяти. Тем не менее, основа для создания таких языков была предложена еще в середине 30-х годов 20-го века Алонзо Черчем (Alonzo Church) и Стефаном Клини (Stephen Kleene). Мы рассмотрим теорию Черча, названную им лямбда-исчислением, в качестве теоретической основы и "минимального" функционального языка программирования. В конечном итоге любую программу, написанную на языке *Haskell* или любом другом функциональном языке программирования, можно сравнительно несложными преобразованиями свести к формуле лямбда-исчисления.

Разумеется, ни Черч, ни Клини не создавали язык программирования. Вопрос, которым они интересовались – это вопрос о формализации понятия вычислимой функции и создания универсального математического аппарата для определения и работы с вычислимыми функциями. В то время эта область математики активно развивалась в трудах уже упомянутых Черча и Клини, а также Тьюринга (Alan Turing), Поста (Emil Post) и других. Во многих работах предлагался следующий подход к формализации понятия функции: в качестве основы берется некоторый "минимальный" набор базовых функций и предлагаются способы построения из них более сложных функций с помощью тех или иных *комбинаторов* (функций высшего порядка). Таков, например, подход, использованный Геделем (Kurt Friedrich Gödel) при создании им теории рекурсивных функций. Напротив, в созданном им лямбда-исчислении Черч сумел построить такую систему, при которой базовых функций нет вовсе, а вместо способов построения сложных функций из простых (комбинаторов) используются правила преобразований, с помощью которых можно получать из одних функций другие, эквивалентные им. В результате Черчу удалось создать исчисление чистых функций, в котором все математические понятия, такие, как числа, арифметика и другие, сводятся к понятию функции. Процесс преобразования формул в лямбда-исчислении Черча напоминает процесс вычисления функции, происходящий при исполнении программы, написанной на функциональном языке программирования. Именно поэтому лямбда-исчисление и легло в основу определения языка ЛИСП в начале 60-х годов 20 века, а затем оказалось в основе и других функциональных языков программирования.

Рассмотрим теорию Черча более подробно. Основным понятием в лямбда-исчислении является понятие *выражения* или *формулы*. Его можно определить рекурсивно. Прежде всего, зафиксируем набор идентификаторов, которые в дальнейшем будем называть *переменными*. Мы будем использовать в качестве имен латинские буквы *x*, *y*, *f* и др. В формулах переменные обычно обозначают аргументы функций, задаваемых лямбда-выражениями, однако сама по себе переменная является простейшим видом выражения. Два других вида выражений – это определение безымянной функции (*лямбда-выражение*) и *применение функции*.

Лямбда-выражение имеет вид (*λx.e*), где *x* - имя переменной, а *e* - выражение. Семантически такое выражение обозначает функцию с аргументом *x* и телом *e*. Применение функции записывается в виде (*e1 e2*), где *e1* и *e2* - выражения (*e1* - функция, а *e2* - ее аргумент). Приведем несколько примеров.

λx.x – простейшая функция, выдающая свой аргумент; скобки опущены, поскольку это не вызывает неоднозначности.

λf.λx.f x – функция с двумя аргументами, применяющая свой первый аргумент ко второму. Строго говоря, надо было бы расставить скобки, чтобы выражение приняло вид *λf.*(*λx.*(*f x*)), однако, принято соглашение, по которому "операция" применения функции к аргументу имеет более высокий приоритет, чем "операция" образования лямбда-выражения, при этом функции применяются в порядке слева направо, то есть выражение *f x y* понимается как применение функции *f* к аргументу *x*, и применение полученного результата к аргументу *y*.

(*λx.x x*)(*λx.x x*) – применение функции, заданной лямбда-выражением (*λx.x x*), к аргументу, представляющему собой такое же лямбда-выражение. Внутри тела, задающего лямбда-выражение, аргумент *x* применяется к себе.

Мы будем рассматривать не классическое лямбда-исчисление, в котором кроме функций и их применений к другим функциям больше ничего нет, а некоторое его расширение. В нашем расширенном лямбда-исчислении помимо безымянных функций, заданных лямбда-выражениями, будут использоваться константы, смысл которых задан вне лямбда-исчисления. Это будут привычные нам по языкам программирования константы, обозначающие целые числа, символы и логические значения, константа, обозначающая пустой список `NIL` (этим идентификатором мы будем обозначать объект, который ранее в языке *Haskell*, всегда обозначался нами парой квадратных скобок `[]`), а также константы, задающие обозначения примитивных функций. Примитивные функции нашей системы следует рассмотреть особо.

Многие примитивные функции уже хорошо нам знакомы по языку *Haskell* и многим другим языкам программирования. Это обычные

арифметические операции сложения, умножения и другие, операции сравнения величин "больше", "меньше", "равны" и другие, операции над логическими значениями "и", "или" и другие. Мы будем пользоваться ими без какого-либо объяснения. Единственное отличие от стандартного использования этих и других операций от их использования в других языках программирования будет заключаться в том, что мы всегда будем использовать только префиксную запись операций, то есть вместо привычного `3+5` будем записывать выражение `+ 3 5`. Конечно, все функции в нашем расширенном лямбда-исчислении будут карринговыми, то есть выражение `+ 3` также имеет смысл и представляет собой применение функции `+` к константе `3`, в результате которого получается функция увеличения целого аргумента на `3`.

Следует заметить также и то, что если в классическом лямбда-исчислении применение любой функции к любому аргументу всегда осмысленно, поскольку любой "объект" – как аргумент, так и результат – всегда представляет собой лямбда-выражение – функцию одного аргумента, то в нашем расширенном лямбда-исчислении примитивные функции можно применять только к "правильным" аргументам. Так, бессмысленным будет выражение `+ TRUE 0`, поскольку невозможно выполнить сложение логического значения `TRUE` с числом `0`. Также бессмысленным будет выражение `/ 3 0`, поскольку результат деления числа `3` на число `0` не определен. Мы не будем рассматривать бессмысленные выражения.

Некоторые важные синтаксические конструкции функциональных языков программирования также будут представлены в нашем расширенном лямбда-исчислении в виде стандартных функций. Так, например, условные выражения `if B then E1 else E2` могут быть представлены применением стандартной функции `IF` с тремя аргументами `B`, `E1` и `E2`. Кортежи будем представлять в виде применения стандартных функций кортежирования `TUPLE-n`, где `n` – произвольное натуральное число. Такая функция будет иметь `n` аргументов и выдавать в результате работы кортеж, содержащий эти аргументы. Обратная функция извлечения элемента кортежа по заданному номеру будет представлена стандартной функцией `INDEX`, аргументами которой будут номер элемента кортежа и сам этот кортеж. Например, если составить кортеж из чисел 2, 4 и 10, а потом извлечь из него второй элемент, то мы должны в результате получить число 4. Соответствующая конструкция расширенного лямбда-исчисления будет выглядеть следующим образом:

```
INDEX 1 (TUPLE-3 2 4 10)
```

Обратите внимание, что элементы кортежа нумеруются с нуля, так что второй элемент кортежа будет иметь номер 1.

В лямбда-исчислении определены эквивалентные преобразования выражений. С их помощью можно переходить от одних выражений к другим, эквивалентным им. В функциональном программировании аналогом этому процессу служит процесс вычисления выражения, поэтому мы иногда будем говорить об эквивалентном преобразовании выражений в лямбда-исчислении как о вычислении выражений. Как правило, осмысленными будут являться те преобразования, которые позволяют упростить выражение, однако, все преобразования классического лямбда-исчисления на самом деле обратимы. В следующем разделе мы введем правила преобразований для выражений, которые позволят нам рассматривать лямбда-исчисление как своеобразный язык программирования, позволяющий по заданной программе (выражению) вычислить результат работы этой программы (эквивалентное ему "простейшее" выражение).

3.2. Система вывода результатов

Прежде, чем рассматривать преобразования выражений, введем важное понятие *свободной* и *связанной* переменной. Неформально говоря, вхождение переменной в некоторое выражение будет связанным, если оно находится внутри некоторого лямбда-выражения, в заголовке которого эта переменная упомянута в качестве аргумента. Другими словами, если имеется выражение, в которое в качестве подвыражения входит $\lambda x.e$, то все вхождения переменной x в выражении e будут связанными. Если переменная не является связанной в некотором выражении, то она в нем будет свободной. Можно определить понятия свободных и связанных переменных и более формально. Для этого рассмотрим понятие "множества свободных переменных" некоторого выражения. Те переменные, которые не будут входить в это множество, будут в этом выражении связанными.

Итак, если выражение E представляет собой переменную x, то множество свободных переменных этого выражения $F(E)$ содержит только эту переменную: $F(x) = \{ x \}$. Если выражение представляет собой одну из стандартных констант нашего расширенного лямбда-исчисления, то оно не содержит ни свободных, ни связанных вхождений переменных, $F(c) = \{\}$. Если выражение является применением функции к аргументу и имеет вид $E = e1\ e2$, то множество свободных переменных этого выражения является объединением множеств свободных переменных выражений $e1$ и $e2$: $F(e1\ e2) = F(e1) \cup F(e2)$. Наконец, если выражение имеет вид лямбда-выражения $E = \lambda x.e$, то его множество свободных переменных получится, если из множества свободных переменных выражения e убрать переменную x: $F(\lambda x.e) = F(e) \setminus \{ x \}$.

Следует отметить, что одна и та же переменная может быть связанной в некотором выражении и одновременно свободной в некотором

его подвыражении. Так, например, в выражении $\lambda f.f\,x$ переменная x – свободная, а переменная f – связанная. Однако если рассматривать только тело этого лямбда-выражения $f\,x$, то в нем обе переменных f и x свободные. Вообще говоря, выражения, содержащие свободные переменные, хотя и являются формально допустимыми, но, как правило, бессмысленны. Например, выражение $\lambda f.f\,x$ представляет собой функцию, которая применяет свой аргумент к переменной "x". Поскольку переменная x – свободная, то ее смысл в этом выражении не определен. Если же связать эту переменную, то мы получим уже вполне осмысленное выражение $\lambda x.\lambda f.f\,x$, представляющее собой функцию, которая применяет свой второй аргумент к первому. Обычно мы рассматриваем выражения, содержащие свободные переменные, только в качестве составных частей других выражений, в которых рассматриваемые переменные уже являются связанными.

Мы рассмотрим четыре способа преобразования выражений в лямбда-исчислении. Первое из рассматриваемых преобразований называется переименованием переменных или α-преобразованием. Смысл этого преобразования состоит в том, что суть функции не меняется, если заменить имя ее формального аргумента. Формально α-преобразование заключается в замене в выражении $\lambda x.e$ имени переменной x на любое другое имя с одновременной заменой всех *свободных* вхождений этой переменной в выражение e. Преобразование возможно, если только новая переменная уже не входит свободно в выражение e, а также если при этом свободное вхождение переменной не окажется связанным. Так, например, в выражении $\lambda x.\lambda f.f\,x\,y$ можно заменить переменную x, скажем, на переменную z. В результате получится выражение $\lambda z.\lambda f.f\,z\,y$. Разумеется, новое выражение эквивалентно исходному и имеет тот же смысл. Однако, в том же выражении переменную x нельзя заменить на y, поскольку переменная y входит в тело лямбда-выражения свободно, так что получившееся после замены выражение $\lambda y.\lambda f.f\,y\,y$ уже будет иметь другой смысл – в нем оба вхождения переменной y оказываются связанными. Нельзя также произвести и замену x на f, поскольку это приведет к тому, что в теле лямбда-выражения свободное вхождение переменной x превратится в связанное вхождение переменной f, и получившееся выражение $\lambda f.\lambda f.f\,f\,y$ также будет иметь уже другой смысл.

Переименование не приводит к изменению длины или структуры выражения, оно применяется не как средство для "вычисления" выражения, но только для "технических" целей. Два следующих преобразования – это "вычислительные" преобразования, которые, вообще говоря, приводят к упрощению выражения, сведению его к кратчайшей форме. Поэтому эти преобразования называют еще *редукциями* от слова *reduce* – сокращать, редуцировать. В программировании эти две редукции

соответствуют применению "встроенных" и "определенных программистом" функций к фактическому аргументу или аргументам.

Преобразование, называемое *δ-редукцией*, соответствует применению "встроенной" функции к константным аргументам. Правило δ-редукции имеет следующий вид. Пусть имеется выражение *e e1 e2... ek*, где *e* – константа, представляющее имя "встроенной" функции с *k* аргументами, а *e1*, *e2*,... *ek* – значения, могущие служить аргументами этой функции. Тогда такое выражение можно заменить на эквивалентное ему выражение, представленное значением, получающимся как результат применения функции к заданным значениям аргумента. Так, например, если константа `+` представляет функцию арифметического сложения целых, а константа `OR` - функцию логического "или", то в результате δ-редукции выражение `+ 1 4` может быть преобразовано к выражению `5`, а выражение `OR TRUE FALSE` – к выражению `TRUE`. Теоретически δ-редукция обратима, то есть можно действовать и наоборот, представляя константы в виде применения встроенных функций к аргументам-константам. Однако на практике желание представить константу `5` в виде `+ 2 3` никогда не возникает.

Подчеркнем, что δ-редукция возможна только в том случае, когда аргументы встроенной функции уже представлены вычисленными значениями, как правило, константами нужного типа, причем имеется необходимое число этих аргументов. Так, например, возможно применение δ-редукции к выражению `+ 2 3`, однако, к выражению `+ 2` δ-редукция не применима (недостаточно аргументов) или к выражению `+ 2 (* 2 3)` (второй аргумент не является константой).

Преобразование, называемое β-редукцией, соответствует применению функции, представленной лямбда-выражением, к аргументу. Если исходное выражение имело вид (*λx.e*) *a*, то в результате применения β-редукции оно будет преобразовано в *e*{*x*|*a*} – выражение *e*, в котором все свободные вхождения переменной *x* заменены на выражение *a*. Например, выражение `((λx.+ x x) 3)` в результате применения β-редукции будет преобразовано в `(+ 3 3)`, которое, в свою очередь, может быть преобразовано в `(6)` с помощью применения δ-редукции.

Если в некоторое выражение в качестве его подвыражения входит такое выражение, к которому можно применить одну из редукций, то такое подвыражение называется *редуцируемым* или сокращенно *редексом* (от *redex – reducible expression*). Таким образом, процесс преобразования выражения сводится, в основном, к применению β- и δ-редукций к редексам, содержащимся в исходном выражении.

Заметим, что при применении β-редукции замене подлежат именно свободные переменные тела лямбда-выражения. Рассмотрим, например, следующее выражение: `((λx.+ x ((λx.* x x) 2)) 4)`. В этом

выражении имеются два подвыражения-редекса, к которым можно применить β-редукцию. Первый из этих редексов – это само выражение целиком, второй – подчеркнутое подвыражение. Если применить β-редукцию к первому из этих редексов – внешнему, – то получится выражение `(+ 4 ((λx.* x x) 2))`. Здесь при применении редукции свободное вхождение переменной `x` было заменено константой `4`, а связанное вхождение этой переменной во внутреннем редексе осталось неизменным. Мы можем, однако, применить β-редукцию к внутреннему редексу, тогда исходное выражение будет преобразовано в выражение `((λx.+ x (* 2 2)) 4)`.

Последнее преобразование, называемое η-преобразованием, выражает тот факт, что две функции, которые при применении к одному и тому же аргументу дают один и тот же результат, эквивалентны. Формально η-преобразование может быть записано следующим образом: выражение *λx.(E x)* эквивалентно *E* при условии, что выражение *E* не содержит свободных вхождений переменной *x*. Действительно, первое из этих выражений – *λx.(E x)* – представляет собой функцию, которая при задании ей в качестве аргумента некоторого значения выдает результат, эквивалентный результату применения функции *E* к этому аргументу. Возможность применения η-преобразования означает, что полученная функция эквивалентна функции *E*.

Основное назначение лямбда-исчисления состоит в том, чтобы показать, что любая вычислимая функция может быть представлена в виде лямбда-выражения. При этом редукции и другие преобразования служат необходимым аппаратом для доказательства того, что построенная функция действительно дает нужный результат при применении ее к тем или иным аргументам. Например, легко видеть, что функция, представленная лямбда-выражением `(λx.* x x)` представляет собой функцию возведения в квадрат. Чтобы показать, что это действительно так, можно попробовать применить эту функцию к некоторым аргументам и посмотреть, действительно ли получается нужный результат (разумеется, это не строгое доказательство, а, скорее, процесс отладки, призванный убедить автора функции в том, что его функция "работает правильно"). Проверим, например, что функция действительно выдаст результат `36`, если применить ее к аргументу `6`. Для этого составим выражение `((λx.* x x) 6)` и проведем его редукции. Сначала в результате применения β-редукции получится выражение `(* 6 6)`, к которому можно, в свою очередь, применить δ-редукцию и окончательно получить в результате значение `36`.

Вообще, редукции можно применять до тех пор, пока в выражении имеется хотя бы один редекс. Если ни одного редекса в выражении нет, то говорят, что выражение находится в нормальной форме. Наш процесс

"отладки" сводится к преобразованию исходного выражения к его нормальной форме.

Как мы уже видели, в одном и том же выражении может содержаться несколько редексов, а значит, процесс преобразования выражения к нормальной форме – это не однозначный процесс. Например, мы видели, что исходное выражение `((λx.+ x ((λx.* x x) 2)) 4)` может быть преобразовано двумя разными способами либо к `(+ 4 ((λx.* x x) 2))`, либо к `((λx.+ x (* 2 2)) 4)`. В любом случае получившееся выражение еще не находится в нормальной форме, и может быть преобразовано далее. В первом случае процесс дальнейшего преобразования выражения происходит однозначно: сначала применяется β-редукция к единственному имеющемуся в выражении редексу, а потом два раза применяется δ-редукция. В результате последовательно получим выражения `(+ 4 (* 2 2))`, затем `(+ 4 4)` и, наконец, `8`. Во втором случае опять имеем два редекса в выражении – δ-редекс `(* 2 2)` и все выражение, представляющее собой β-редекс. Последовательно преобразуя это выражение применением δ- и β-редукций, получим сначала `((λx.+ x 4) 4)`, потом `(+ 4 4)` и, наконец `8`. Как и следовало ожидать, несмотря на разные способы преобразования выражения результат – нормальная форма – получился одним и тем же. Это, в частности, свидетельствует о том, что наша система преобразований "разумна".

Справедлив и более общий факт: если у некоторого выражения существует нормальная форма, то она единственна, то есть какими бы способами ни выполнять преобразования, результат всегда будет одним и тем же, если только вообще он будет достигнут. На самом деле нормальная форма существует не всегда, так же как не любая программа обязана завершиться выдачей результата. Например, уже рассматривавшееся нами ранее выражение `(λx.x x)(λx.x x)` не имеет нормальной формы. Действительно, само это выражение не является нормальной формой, поскольку к нему можно применить β-редукцию. Однако, при подстановке аргумента `(λx.x x)` вместо переменной `x` в тело лямбда-выражения `x x`, мы получим опять то же самое выражение `(λx.x x)(λx.x x)`. Выражение, подобное только что рассмотренному, является простейшей формой "зацикливающейся" программы.

Когда мы изучали программирование на языке *Haskell*, нам встречались функции, которые могли выдавать значение при одном способе вычисления, и "зацикливаться" при другом способе вычисления. В качестве таких примеров можно было рассмотреть любую из функций обработки "бесконечных" списков, которые были весьма полезны и выдавали осмысленный результат при ленивом способе передачи аргументов в функцию, однако, при энергичном способе вычисления эти функции никогда не заканчивали работу. Подобные примеры выражений

можно привести и для лямбда-исчисления. Например, выражение `(λx.λy.y)((λx.x x)(λx.x x))` можно привести к нормальной форме, если выполнить β-редукцию, считая все выражение редексом. Действительно, выражение представляет собой применение функции `(λx.λy.y)` к некоторому аргументу. Однако, аргумент `x` не используется в теле функции – `λy.y`, так что независимо от того, что представляет собой этот аргумент, результатом такой β-редукции будет выражение `λy.y`. С другой стороны, в исходном выражении есть и еще один редекс – "зацикливающееся" выражение `(λx.x x)(λx.x x)`. Если применять β-редукцию к нему, то вычисления никогда не закончатся.

Можно найти и еще одну аналогию. Программы на языке *Haskell* могли "зациклиться" при энергичном способе вычислений и выдавать осмысленный результат при ленивых вычислениях. Однако, невозможно привести пример, при котором некоторая программа нормально заканчивает работу при энергичном способе вычислений и не может закончить работу при ленивых вычислениях. В этом смысле ленивые вычисления – это более "аккуратный" способ исполнения программы, при котором программа всегда выдает некоторый результат, если только она может выдать результат хотя бы при одном способе вычислений. Аналогично этому, в лямбда-исчислении существует такой порядок применения редукций, который гарантирует получение нормальной формы, если только нормальная форма вообще существует для заданного выражения. Рассмотрим различные порядки применения редукций.

Пусть в некотором выражении имеется несколько редексов. Заметим прежде всего, что любые два редекса могут быть расположены только таким образом, что они либо совсем не пересекаются, либо один из них содержится внутри другого. Так, например, в выражении `(λx.λy.y)((λx.x x)(λx.x x))` имеются два редекса, при этом один из них – это все выражение целиком, а другой, содержащийся в нем, представляет собой аргумент вызова. Назовем редекс *самым внешним*, если он не содержится ни в каком другом из редексов рассматриваемого выражения. Так, в рассматриваемом выражении редекс, представляющий собой все выражение, является самым внешним. Напротив, назовем редекс *самым внутренним*, если внутри него не содержится редексов. Второй из рассматриваемых в нашем выражении редексов – аргумент вызова – является самым внутренним. Конечно, может оказаться, что в выражении есть несколько самых внутренних и несколько самых внешних редексов, причем одни и те же редексы могут быть как самыми внешними, так и самыми внутренними. Разумеется, если редекс содержится внутри другого и при этом внутри себя также содержит редексы, то он не будет ни самым внешним, ни самым внутренним.

Рассмотрим следующий канонический порядок редукций: пусть преобразования происходят таким образом, что редукция всегда

применяется к самому левому из **самых внутренних** редексов. При этом может оказаться, что в некоторый момент выражение окажется в нормальной форме, и тогда "вычисления" будут закончены. Может, конечно, оказаться и так, что этот процесс никогда не будет закончен. В выражении `(λx.λy.y)((λx.x x)(λx.x x))` самым внутренним является "зацикливающийся" редекс `(λx.x x)(λx.x x)`, поэтому наш "канонический" порядок вычислений не сможет привести выражение к нормальной форме. Этот описанный порядок редукций называется *аппликативным*, сокращенно – АПР (аппликативный порядок редукций). Аппликативный порядок редукций можно представлять себе как порядок, при котором в каждом вызове функции прежде всего "вычисляется" значение аргумента вызова, а потом уже происходит подстановка этого значения в тело вызываемой функции. Таким образом, этот порядок редукций соответствует энергичному способу вычисления значения функций в функциональном программировании.

Можно рассмотреть и еще один порядок применения редукций. Можно всегда применять редукцию к самому левому из **самых внешних** редексов. Этот порядок редукций называется *нормальным* или сокращенно НПР (нормальный порядок редукций). Таким образом, если выражение представляет собой применение функции к аргументам, то подстановка аргумента в тело функции происходит до того, как преобразования будут производиться над самими аргументами. Такая схема преобразования выражения похожа на процесс ленивых вычислений в функциональных языках программирования, но если обращение к аргументу в теле функции происходит несколько раз, то при схеме ленивых вычислений в функциональных языках программирования вычисление аргумента все-таки осуществляется лишь однажды – при первом обращении к аргументу; в дальнейшем происходит обращение к уже вычисленному значению. В нашей же схеме применения редукций к выражениям в лямбда-исчислении процесс преобразования выражения, которое служит аргументом применения функции, будет происходить столько раз, сколько раз аргумент появляется в определении этой функции.

Рассмотрим, например, следующее выражение, представляющее собой применение функции возведения в квадрат к результату сложения двух чисел: `(λx.* x x)(+ 3 4)`. Все выражение представляет собой самый внешний редекс, а выражение аргумента – это самый внутренний редекс. Поэтому при аппликативном порядке редукций выражение будет приведено к нормальной форме с помощью следующих трех шагов преобразования. Сначала, после применения δ-редукции к выражению аргумента получим `(λx.* x x) 7`, потом β-редукция преобразует это выражение к `(* 7 7)` и, наконец, последнее применение δ-редукции приведет выражение к нормальной форме `49`. При нормальном порядке редукций первой будет применена β-редукция ко всему выражению, в

результате чего получится выражение (* (+ 3 4) (+ 3 4)), представляющее собой применение функции умножения к двум одинаковым аргументам, представляющим собой сдублированное выражение, которое в исходном выражении было представлено лишь в одном экземпляре. Теперь для получения окончательного результата потребуются еще три шага δ-редукции, после которых будут последовательно получены выражения (* 7 (+ 3 4)), затем (* 7 7) и, наконец, 49.

Приведенный пример показывает, что при применении НПР для приведения выражения к нормальной форме может потребоваться больше шагов, чем при применении АПР, однако, НПР более "надежен", так как он позволяет получить нормальную форму выражения всегда, если только такая форма вообще существует. Так, например, уже рассмотренное нами ранее выражение (λx.λy.y)((λx.x x)(λx.x x)) может быть приведено к нормальной форме за один шаг при применении НПР, однако, АПР не позволит найти эту нормальную форму ни за какое конечное число шагов

До сих пор мы преобразовывали выражения, используя только β- и δ-редукции. В применении α- и η-преобразований просто не было необходимости. Рассмотрим, однако, следующее выражение: λy.(λy.+ 2 y)(+ 1 y). В этом выражении имеется функция, тело которой представляет собой применение некоторой простой функции к аргументу (+ 1 y). К сожалению, формальные аргументы обеих функций представлены переменной с одним и тем же именем – y. Это может привести к ошибке, если подстановку производить неправильно. Пусть, например, описанное выражение применяется к константе 1. При применении НПР в выражение (λy.+ 2 y)(+ 1 y) надо подставить единицу вместо свободного вхождения переменной y. В результате получится выражение (λy.+ 2 y)(+ 1 1), которое в результате следующих шагов будет преобразовано последовательно в (+ 2 (+ 1 1)), затем в (+ 2 2) и, наконец, в 4. Если, однако, ошибочно выполнить подстановку вместо **всех** вхождений переменной y в тело функции, то мы получим выражение (λy.+ 2 1)(+ 1 1), которое в результате дальнейших преобразований примет вид сначала (+ 2 1) и после следующего шага – 3. Это, очевидно, неправильно, но для того, чтобы такой ошибки избежать, надо при каждой подстановке определять, какие вхождения переменной являются свободными, а какие связанными. Если выражения достаточно сложные, то это не так-то просто.

Можно, однако, с самого начала постараться избежать подобных ошибок, выполнив α-преобразование перед применением редукций. Так, если сначала преобразовать исходное выражение к виду λz.(λy.+ 2 y)(+ 1 z), то при применении его к константе 1 ошибку

сделать будет уже гораздо труднее. Правда, ошибиться можно и при выполнении начального α-преобразования.

В вышеописанном примере α-преобразование (замена переменной) использовалось только для того, чтобы облегчить применение редукций; если преобразование происходит автоматически, и программа, выполняющая преобразование, достаточно "интеллектуальна", чтобы отличать свободные вхождения переменных от связанных, то переименование можно было и не производить. Однако бывают случаи, когда без переименования переменных не обойтись. Рассмотрим, например, следующее выражение: `λy.(λx.λy.+ x y) y`. Здесь также имеются два лямбда-выражения, имеющих одну и ту же связанную переменную – `y`. Соответственно, в теле внешнего лямбда-выражения одно вхождение переменной `y` является свободным, а другое – связанным. В этом выражении имеется единственный редекс, представляющий собой тело функции, определенной внешним лямбда-выражением. Попытка применить β-редукцию без предварительного проведения α-преобразования приведет к ошибке: мы получим выражение `λy.(λy.+ y y)`, в котором свободное вхождение переменной `y` попало в позицию, где переменная `y` связана.

Очевидно, что полученное выражение не эквивалентно исходному. Чтобы убедиться в этом, можно попробовать применить исходную и результирующую функцию к одним и тем же аргументам, например, к аргументам `1` и `2`. Для исходного выражения, применяя β-редукции в нормальном порядке, последовательно получим следующие выражения: `((λy.(λx.λy.+ x y) y) 1 2)`, затем `((λx.λy.+ x y) 1) 2)`, `((λy.+ 1 y) 2)`, `(+ 1 2)`, и, наконец, `3`. Для выражения, полученного в результате неправильного применения редукции, получим: `((λy.(λy.+ y y)) 1 2)`, `((λy.+ y y) 2)`, `(+ 2 2)` и, наконец, `4`. Итак, видим, что иногда применение β-редукции без предварительного переименования переменных может привести к ошибке.

Можно высказать и более общее утверждение: β-редукция может привести к ошибке тогда, когда выражение, имеющее в своем составе свободную переменную, подставляется в качестве аргумента в выражение, имеющее в своем составе связанную переменную с тем же именем. Заметим, что необходимость в такой подстановке возникает лишь в определенном случае, а именно, тогда, когда редукции производятся внутри лямбда-выражения, то есть преобразованию подвергается тело определяемой функции. Если имеется некоторое лямбда-выражение, содержащее внутри себя редексы, то при применении АПР редукции в теле лямбда-выражения всегда будут производиться до того, как это лямбда-выражение будет применено к аргументу. Однако, при применении НПР необходимость в выполнении редукций внутри лямбда-выражения может возникнуть только тогда, когда это лямбда-выражение не применяется уже

ни к каким аргументам, в противном случае, сначала будет произведена подстановка аргумента, а затем уже будет производиться редукция полученного выражения. Это означает, что мы можем избежать необходимости переименования переменных в выражении, если будем применять НПР, оставляя редексы внутри лямбда-выражений без преобразования. Следуя этому правилу, мы в результате последовательного применения β- и δ-редукций не получим нормальной формы выражения, однако, получим довольно близкий ее эквивалент – так называемую *слабую заголовочную нормальную форму* (сокращенно – СЗНФ).

Ранее мы определяли нормальную форму как выражение, не содержащее внутри себя редексов. Если рассмотреть все возможные типы выражений, то мы увидим, что выражение, находящееся в нормальной форме может иметь один из следующих видов:

константа *c* (в том числе – одна из встроенных функций);

переменная *x*;

лямбда-выражение $\lambda x.e$, где *e* – выражение, находящееся в нормальной форме;

f e1 e2... ek, где *f* – встроенная функция с *n* аргументами, *e1*, *e2*,... *ek* – выражения в нормальной форме, причем $k < n$.

Последний вид нормальной формы выражения – это применение встроенной функции с недостаточным для выполнения δ-редукции числом переданных ей аргументов, например, `(+ 1)`. Заметим, что этот последний вид нормальной формы выражения эквивалентен (вспомним η-редукцию!) некоторому лямбда-выражению, находящемуся в нормальной форме. Так, например, выражение `(+ 1)` эквивалентно выражению `(λx.+ 1 x)`. Теперь сходным образом можно определить и СЗНФ. Будем говорить, что выражение находится в СЗНФ, если оно имеет один из следующих видов:

константа *c* (в том числе - одна из встроенных функций);

переменная *x*;

лямбда-выражение $\lambda x.e$, где *e* – произвольное выражение;

f e1 e2... ek, где *f* – встроенная функция с *n* аргументами, *e1*, *e2*,... *ek* – произвольные выражения, причем $k < n$.

Если записать последнее выражение в эквивалентном виде лямбда-выражения, то можно видеть, что нормальная форма отличается от СЗНФ только тем, что в СЗНФ не производятся преобразования редексов внутри тела лямбда-выражения.

Теперь можно определить процесс "вычисления" выражения как процесс приведения его к СЗНФ с помощью β- и δ-редукций, применяемых в нормальном порядке. При таком процессе никогда не потребуется применять α-преобразований для "безопасного" переименования переменных. Таким образом можно считать, что у нас имеется достаточно

простой и формальный процесс преобразования выражений лямбда-исчисления в простую форму с помощью только β- и δ-редукций. Такой процесс преобразования примерно соответствует исполнению программы в некотором простом языке функционального программирования. В дальнейшем мы покажем, как можно сравнительно простыми преобразованиями свести программу, написанную на языке, подобном языку *Haskell*, к вычислению выражения, записанного средствами лямбда-исчисления, поэтому можно считать, что лямбда-исчисление является основой для реализации исполнения функциональных программ.

Выше мы уже отмечали, что процесс преобразования формы выражения при аппликативном порядке редукций напоминает процесс выполнения функциональной программы при энергичном способе исполнения, то есть при таком способе вызова функций, когда перед началом работы тела функции происходит вычисление фактических аргументов и передача полученных значений в функцию для вычисления. Напротив, при нормальном порядке применения редукций вычисления происходят таким образом, что подстановка аргументов происходит до того, как будут вычислены их значения, что примерно соответствует ленивой схеме исполнения функциональной программы. Однако, имеется и существенная разница между ленивыми вычислениями и текстовыми преобразованиями выражений так, как это принято в лямбда-исчислении. При исполнении функциональной программы, даже если при вызове функции ее аргумент и не вычисляется сразу же, он все же будет вычисляться не более одного раза – при первом обращении к этому аргументу из тела функции, когда он передается одной из встроенных функций или производится сопоставление этого аргумента с некоторым образцом. В случае же преобразований выражений с помощью редукций вычисление аргументов будет происходить при **каждом** обращении к аргументу в теле вычисляемой функции. Поясним сказанное примером.

Пусть требуется привести к СЗНФ выражение `(λx.* x x)(+ 2 3)`. В этом выражении имеются два редекса, однако, если мы используем НПР, то прежде всего выполняется β-редукция, при которой выражение аргумента `(+ 2 3)` подставляется вместо свободного вхождения переменной `x` в тело лямбда-выражения, при этом получается выражение `(* (+ 2 3) (+ 2 3))`. После этого выражение `(+ 2 3)` придется вычислить два раза применением к нему δ-редукции, и только затем уже будет получено окончательное значение `25`. С точки зрения получения правильного результата не имеет значения то, сколько раз будет применяться редукция к одним и тем же выражениям, поэтому отмеченная нами "неэффективность" вычислений несущественна с точки зрения построения теории вычислимых функций. Однако для того, чтобы строить адекватную модель вычислений, средств, предложенных нами в качестве описания преобразования выражений, недостаточно.

Расширим наше лямбда-исчисление, введя еще одну дополнительную конструкцию в качестве допустимого выражения. Новая конструкция будет иметь вид `let x=e1 in e2`, где `x` – переменная, а `e1` и `e2` – произвольные выражения. Новый вид выражения будет полностью эквивалентен выражению `(λx.e2) e1`, однако процесс его редуцирования будет происходить в соответствии с правилами ленивых вычислений, а именно, следующим образом. Данное выражение считается эквивалентным выражению `e2`, однако, если при его преобразовании потребуется произвести δ-редукцию, в которой одним из операндов встроенной функции является переменная `x`, упомянутая в заголовке выражения, то прежде всего выражение `e1` приводится к СЗНФ, после чего результат подставляется вместо всех свободных вхождений переменной `x` в текущее выражение, после чего вычисления производятся обычным образом. Теперь модифицируем правило выполнения β-редукции: при β-редукции выражение `(λx.e) a` будет преобразовываться в `let x=a in e`. Посмотрим, как теперь будут выполняться вычисления в НПР.

Снова рассмотрим исходное выражение `(λx.* x x)(+ 2 3)`. Теперь, однако, выполнение β-редукции приведет нас к выражению `let x = + 2 3 in (* x x)`. Теперь требуется преобразовывать выражение `(* x x)`, однако, здесь перед проведением δ-редукции потребуется сначала привести к СЗНФ выражение `(+ 2 3)`. После того, как будет получен результат вычисления – выражение `5`, – этот результат будет подставлен в выражение `(* x x)` вместо переменной `x`. В результате мы получим выражение `(* 5 5)`, которое превратится в результат вычислений – число `25` уже в результате обычной δ-редукции. На этом примере хорошо видно, что вычисления происходят именно так, как это было описано при описании схемы ленивых вычислений, так что "лишних" вычислений не производится, однако, с чисто формальной точки зрения только что описанный способ преобразования выражений, конечно же, гораздо сложнее и сложнее поддается анализу.

Примеры решения задач

В этом разделе приведем примеры анализа и преобразования выражений с помощью системы редукций лямбда-исчисления.

Задание 1. Укажите связанные и свободные переменные в выражении

```
(λx.λy.z (λz.z (λx.y))
```

Решение. Для каждой из входящих в выражение переменных необходимо проанализировать структуру вложенности блоков и найти определяющее вхождение (стоящее непосредственно за символом

«лямбда»). Если такое определяющее вхождение находится, то переменная связана, иначе – свободна.

В данном выражении имеются использующие вхождения для двух переменных – два вхождения переменной z и одно вхождение переменной y. Переменная x, хотя и определена дважды в двух различных блоках, но не имеет ни одного использующего вхождения. Первое использующее вхождение переменной z – свободно, поскольку находится за пределами блоков, определяющих эту переменную. Второе вхождение переменной z, а также единственное вхождение переменной y – связанные.

Задание 2. Для выражения

(λx.λy.x (λz.y z))(((λx.λy.y) 8) (λx.(λy.y) x))

найти самый левый из самых внешних и самый левый из самых внутренних редексов.

Решение. Внешняя структура выражения представляет собой применение одного лямбда-выражения к другому. Первое из них не является редексом, так что самым внешним (и единственным самым внешним) редексом является все выражение в целом. Чтобы найти самый левый из самых внутренних редексов, надо найти выражения в форме применений функции к аргументам, внутри которых не содержится других применений функций. Таких редексов в выражении два:

(λx.λy.y) 8

и

(λy.y) x

Из них первый расположен текстуально левее, поэтому именно он и является самым левым из самых внутренних редексов.

Задание 3. Выполнить редукцию выражения

(λx.λy.x (λz.y z))(((λx.λy.y) 8) (λx.(λy.y) x))

в нормальном и аппликативном порядке редукций. В обоих случаях найти нормальную форму (НФ) и слабую заголовочную нормальную форму (СЗНФ) выражения.

Решение. При нормальном порядке редукций редукция всегда применяется к самому левому из самых внешних редексов. Выполняя β-редукции, последовательно получим следующие выражения (в каждом выражении подчеркнуты редексы, используемые на следующем шаге преобразования).

(λx.λy.x (λz.y z))(((λx.λy.y) 8) (λx.(λy.y) x))
λy.(((λx.λy.y) 8) (λx.(λy.y) x)) (λz.y z)

Данное выражение находится в СЗНФ, поскольку имеет вид λy.E. Однако, оно еще не находится в НФ. Поэтому для получения НФ продолжим преобразования с помощью β-редукций.

λy.((λy.y) (λx.(λy.y) x)) (λz.y z)
λy.(λx.(λy.y) x) (λz.y z)

```
λy.(λy.y) (λz.y z)
λy.λz.y z
```

При аппликативном порядке редукций редукции применяются к самым левым из самых внутренних редексов, поэтому последовательность выражений получится следующей.

```
(λx.λy.x (λz.y z))((λx.λy.y) 8) (λx.(λy.y) x))
(λx.λy.x (λz.y z))((λy.y) (λx.(λy.y) x))
(λx.λy.x (λz.y z))((λy.y) (λx.x))
(λx.λy.x (λz.y z))(λx.x)
λy.(λx.x) (λz.y z)
```

Полученное выражение находится в СЗНФ, но не в НФ. Для получения НФ необходимо выполнить еще один шаг редукции.

```
λy.λz.y z
```

Задания для самостоятельной работы

Задание 1. Укажите связанные и свободные переменные в выражении

```
(λx.λy.x z (y z))(λx.y (λy.y))
```

Задание 2. Найдите все β- и δ-редексы в заданном выражении и укажите самые внешние и самые внутренние.

```
(λh.(λx.h (x x))(λx.h (x x)))((λy.y)(+ 1 5))
```

Задание 3. Для выражения из предыдущего задания выполните редукцию выражения до НФ и СЗНФ в аппликативном и нормальном порядках.

```
(λh.(λx.h (x x))(λx.h (x x)))((λy.y)(+ 1 5))
```

3.3. Чистое лямбда-исчисление

По сравнению с языками функционального программирования, подобным *Haskell*, в лямбда-исчислении не хватает чисто практических средств, позволяющих удобно записывать функции. Прежде всего, бросается в глаза отсутствие средств описания рекурсивных функций. Действительно, рекурсивные обращения всегда происходят с использованием имени функции, однако лямбда-исчисление – это язык для описания безымянных функций! Конечно, иногда мы можем обойтись и без использования рекурсии, если имеется достаточно богатый набор встроенных функций. Так, например, имея функцию высшего порядка `foldr` в качестве одной из встроенных функций, мы можем написать функцию вычисления факториала натурального числа и без использования рекурсии. Аналогично, функция свертки дерева `foldTree` позволила нам в главе 2 написать без использования рекурсии функцию разглаживания дерева `flatten`. На самом деле можно показать, что, имея достаточно богатый набор мощных функций высшего порядка, можно всегда обойтись без использования рекурсивных вызовов (такой стиль программирования

называется программированием с помощью *комбинаторов* или *комбинаторным программированием*). Однако, чистое лямбда-исчисление не предполагает использования встроенных функций вообще! Возникает вопрос: достаточно ли средств лямбда-исчисления для того, чтобы выразить в нем всевозможные вычислимые функции, если в нем невозможно обычным образом задавать рекурсивные функции, а встроенных функций, позволяющих обойти эту проблему, нет вообще?

В этом разделе мы постепенно построим чистое лямбда-исчисление, последовательно выразив в виде лямбда-выражений все имевшиеся у нас ранее константы и встроенные функции, и задав с помощью тех же лямбда-выражений все управляющие конструкции языков функционального программирования, включая рекурсию, условное вычисление и др. Начнем мы именно с выражения прямой и косвенной рекурсии в чистом лямбда-исчислении.

Пусть у нас имеется функция, в определении которой есть прямое рекурсивное обращение к себе, например, такое, как определение функции вычисления факториала в языке *Haskell*.

```
fact n = if n == 0 then 1 else n * fact(n-1)
```

Прежде всего, зададим эту функцию с помощью конструкций лямбда-исчисления, считая, что у нас имеются встроенные функции для вычитания (функция `-`), умножения (функция `*`) и сравнения с нулем (функция `eq0`), кроме того, считаем, что у нас также есть функция `if` для условного вычисления и константы `0` и `1`. Тогда определение функции будет выглядеть следующим образом.

```
fact = λn.if (eq0 n) 1 (* n (fact (- n 1)))
```

Здесь мы пока по-прежнему использовали задание имени для функции `fact` для того, чтобы выразить рекурсию. Построим теперь новую функцию, в которой вызываемая функция `fact` будет аргументом:

```
sFact = λfact.λn.if (eq0 n) 1 (* n (fact (- n 1)))
```

Конечно, эта новая функция *не* будет тем факториалом, который мы пытаемся построить хотя бы потому, что ее аргументом является не целое число, а некоторая функция, однако она некоторым вполне определенным образом связана с той функцией вычисления факториала, которую мы пытаемся построить. Зададимся задачей найти такую функцию `f`, что она является *неподвижной точкой* определенной нами функции `sFact`, то есть такую `f`, что выполнено равенство `f = sFact f`. Очевидно, что если такую неподвижную точку удастся найти, то найденная функция `f` как раз и будет тем самым факториалом, который мы ищем, поскольку будет выполнено равенство

```
f = sFact f = λn.if (eq0 n) 1 (* n (f (- n 1)))
```

Итак, задача нахождения функции, эквивалентной заданной рекурсивной функции, свелась к задаче построения неподвижной точки

для некоторой другой функции. Кажется, что эту задачу – задачу нахождения неподвижной точки – в общем виде решить не удастся. *A priori* вообще не ясно, всегда ли такая неподвижная точка существует. Однако оказывается, что удается построить функционал, который для заданного функционального аргумента вычисляет его неподвижную точку. Если обозначить через `Y` такой функционал нахождения неподвижной точки, то для любой функции `f` должно быть справедливо равенство `Y f = f(Y f)`. Другими словами, результатом применения функции `Y` к функции `f` должно быть такое значение `x`, что `x = f(x)`. Одну из таких функций предложил Хаскелл Карри, и теперь эта функция в его честь называется *Y-комбинатором Карри*. Вот запись этой функции в нотации лямбда-исчисления:

```
Y = λh.(λx.h (x x))(λx.h (x x))
```

Проверим, действительно ли для этой функции выполнено равенство `Y f = f(Y f)`. Для этого запишем выражение `Y f` и попробуем привести его к СЗНФ. После того, как вместо `Y` будет подставлено соответствующее лямбда-выражение, мы сможем выполнить один шаг β-редукции, и в результате получим выражение `(λx.f (x x))(λx.f (x x))`. Это выражение еще не находится в СЗНФ, так что мы можем выполнить еще один шаг и с помощью β-редукции привести его к виду `f ((λx.f (x x))(λx.f (x x)))`. Это выражение также не находится в СЗНФ, более того, теперь видно, что привести это выражение к СЗНФ вообще никогда не удастся, так как каждый следующий шаг редукции приводит только к увеличению длины выражения. Однако, из проведенных шагов хорошо видно, что выражение `Y f` действительно эквивалентно выражению `f(Y f)`, поскольку второе получается из первого за два шага редукций.

Итак, мы получили, что выражение для рекурсивной функции можно получить, если построить некоторое вспомогательное выражение, а затем применить к нему Y-комбинатор Карри. Получившееся при этом выражение не может быть приведено к СЗНФ, однако оно все же будет работать как соответствующая рекурсивная функция. Давайте убедимся в этом, построив описанным способом функцию для вычисления факториала заданного числа, а затем применим ее к конкретному числу, скажем, числу 2, и проверим, как получается результат вычисления – число 2, равное `fact(2)`. Для этого попробуем привести к СЗНФ выражение `(Y sFact) 2`, где `Y` обозначает Y-комбинатор Карри, а `sFact` – функцию, приведенную выше, полученную из рекурсивного определения факториала. Последовательные шаги по преобразованию выражения к СЗНФ показаны ниже.

```
Y sFact 2
sFact (Y sFact) 2
```

```
(λfact.λn.if (eq0 n) 1 (* n (fact (- n 1)))) (Y sFact) 2
(λn.if (eq0 n) 1 (* n ((Y sFact) (- n 1)))) 2
if (eq0 2) 1 (* 2 ((Y sFact) (- 2 1)))
(* 2 (Y sFact 1))
```

Остановимся пока здесь. Мы видим, что последовательное выполнение β- и δ-редукций привело нас от выражения `Y sFact 2` к выражению `(* 2 (Y sFact 1))`. Это уже показывает, что преобразование выражения происходит именно так, как ведет себя рекурсивное вычисление факториала. Однако, давайте продолжим преобразования.

```
Y sFact 2
sFact (Y sFact) 2
(λfact.λn.if (eq0 n) 1 (* n (fact (- n 1)))) (Y sFact) 2
(λn.if (eq0 n) 1 (* n ((Y sFact) (- n 1)))) 2
if (eq0 2) 1 (* 2 ((Y sFact) (- 2 1)))
(* 2 (Y sFact 1))
(* 2 (sFact (Y sFact) 1)
(* 2 (λfact.λn.if (eq0 n) 1 (* n (fact (- n 1)))) (Y sFact) 1)
(* 2 (λn.if (eq0 n) 1 (* n ((Y sFact) (- n 1)))) 1)
(* 2 (if (eq0 1) 1 (* 1 ((Y sFact) (- 1 1)))))
(* 2 (* 1 (Y sFact 0)))
(* 2 (* 1 (sFact (Y sFact) 0)))
(* 2 (* 1 ((λfact.λn.if (eq0 n) 1
                        (* n (fact (- n 1)))) (Y sFact) 0)))
(* 2 (* 1 ((λn.if (eq0 n) 1 (* n ((Y sFact) (- n 1)))) 0)))
(* 2 (* 1 (if (eq0 0) 1 (* 0 ((Y sFact) (- 0 1))))))
(* 2 (* 1 1))
2
```

Преобразования закончились вполне предсказуемым результатом, так что можно заключить, что применение Y-комбинатора Карри действительно приводит к нужному результату. Существенно, что мы использовали при вычислениях нормальный порядок редукций, так как при аппликативном порядке редукций вычисления просто никогда не были бы закончены. Существуют и другие формы Y-комбинаторов, в частности, такие, которые применимы и при АПР, однако, все другие известные выражения для Y-комбинаторов выглядят сложнее, чем Y-комбинатор Карри.

Если несколько функций определяются таким образом, что в теле каждой из них имеются вызовы других функций из этого набора, то говорят, что мы имеем дело с взаимно-рекурсивными функциями. На самом деле этот случай можно свести к прямой рекурсии и, тем самым, выразить в лямбда-исчислении не только прямую, но и взаимную рекурсию. Для того чтобы свести взаимную рекурсию к прямой, нам потребуются некоторые дополнительные встроенные функции. Во-первых, введем серию функций кортежирования, которые составляют кортеж из своих аргументов. Будем считать, что если `k` – некоторое натуральное

число, то функция TUPLE-k имеет k аргументов и выдает в качестве результата кортеж из k элементов. Еще одна функция нужна для того, чтобы, наоборот, выделять элемент с заданным номером из кортежа. Назовем эту функцию INDEX. Функция INDEX имеет два аргумента: n – номер элемента кортежа – натуральное число, не превосходящее длины кортежа T, который является вторым аргументом этой функции. Результатом работы функции служит n-ый элемент кортежа T. Если число n – не натуральное или превосходит число элементов кортежа, заданного вторым аргументом, то результат работы функции не определен.

Теперь пусть имеются определения взаимно-рекурсивных функций

$f_1 = F_1(f_1, f_2, \ldots f_n)$
$f_2 = F_2(f_1, f_2, \ldots f_n)$
...
$f_n = F_n(f_1, f_2, \ldots f_n)$

где все F_i - выражения, в которых встречаются рекурсивные обращения к определяемым функциям $f_1, f_2, \ldots f_n$. Прежде всего, образуем новое выражение, представляющее собой кортеж, в котором собраны все определения функций:

T = TUPLE-n
$F_1(f_1, f_2, \ldots f_n)\ F_2(f_1, f_2, \ldots f_n) \ldots F_n(f_1, f_2, \ldots f_n)$

каждая из определяемых функций $f_1, f_2, \ldots f_n$ может теперь быть выражена с помощью выделения соответствующего элемента этого кортежа:

f_i = INDEX i T

поэтому если мы теперь подставим в определение кортежа T вместо всех вхождений f_i их выражения в виде элемента кортежа, то мы получим рекурсивное определение для кортежа T с прямой рекурсией:

```
T = TUPLE-n F1((INDEX 1 T),... (INDEX n T))...
            Fn((INDEX 1 T),... (INDEX n T))
```

Кортеж T теперь может быть представлен как и раньше с помощью Y-комбинатора

```
Y (λT.TUPLE-n F1((INDEX 1 T),... (INDEX n T))...
              Fn((INDEX 1 T),... (INDEX n T)))
```

а каждая из отдельно взятых функций может быть получена в виде элемента этого кортежа.

Теперь попробуем представить в виде лямбда-выражений все стандартные константы и функции, использованные нами ранее. Для этого нам надо будет выбрать представление и разработать технику выполнения операций над целыми числами, логическими значениями и списками (кортежами). Основным критерием для выбора того или иного представления будет функциональность этого представления, то есть нам надо, чтобы все представляемые нами константы и стандартные функции

"вели себя правильно", в соответствии с нашими представлениями о результатах выполнения операций. Проще всего определить логические константы и операции над ними, поскольку таких констант всего две – истина и ложь, а операции над ними подчиняются очень простым законам. С этого и начнем.

Основное назначение логических значений состоит в том, чтобы организовать выбор в ходе вычислений, поэтому начать следует с того, как можно реализовать функцию выбора IF. Эта функция имеет три аргумента, при этом первым аргументом как раз и является логическое значение. Если логическое значение надо представлять некоторой функцией, то проще всего сделать так, чтобы само это логическое значение и отвечало за выбор альтернативы при выполнении функции IF. Для этого реализуем функцию так, чтобы она просто применяла свой первый аргумент к остальным двум, а уже этот аргумент отбрасывал бы один из аргументов и оставлял второй. Тогда реализация констант TRUE и FALSE, а также функция IF получают следующее представление.

```
IF = λp.λt.λe.p t e
TRUE = λx.λy.x
FALSE = λx.λy.y
```

Проверим, что каковы бы ни были выражения A и B, выражение IF TRUE A B эквивалентно выражению A, а выражение IF FALSE A B – эквивалентно B. Действительно, при подстановке наших лямбда-выражений после преобразований в НПР последовательно получаем:

```
IF TRUE A B
(λp.λt.λe.p t e) (λx.λy.x) A B
(λt.λe.(λx.λy.x) t e) A B
(λe.(λx.λy.x) A e) B
(λx.λy.x) A B
(λy.A) B
A
```

Разумеется, совершенно аналогично выражение IF FALSE A B будет преобразовано в B.

Над логическими значениями TRUE и FALSE можно выполнять обычные операции логического сложения, умножения, отрицания и пр. Разумеется, их надо определить так, чтобы при их применении получались правильные результаты. Это нетрудно сделать; вот как могут выглядеть, например, операции AND, OR и NOT:

```
AND = λx.λy.x y FALSE
OR = λx.λy.x TRUE y
NOT = λx.x FALSE TRUE
```

Здесь в определении новых операций использованы уже определенные ранее константы TRUE и FALSE. Сами операции определены совершенно естественным образом "по МакКарти".

Аналогично можно определить и другие операции над логическими значениями, и, по существу, этим и исчерпываются все необходимые средства для работы с логическими значениями. Теперь приступим к определению арифметических значений и функций в терминах чистого лямбда-исчисления.

Мы ограничимся только арифметикой натуральных чисел, все остальные числа – рациональные, вещественные, комплексные и др. можно получить, комбинируя натуральные числа и определяя соответствующие операции над такими числами. Натуральные же числа представляют собой абстракцию подсчета тех или иных объектов. Для построения модели счета нужно выбрать, что мы будем считать. Вообще говоря, считать можно что угодно, при этом можно получить весьма различные модели арифметики, мы, однако, следуя Тьюрингу, будем подсчитывать, сколько раз некоторая функция применяется к своему аргументу. Это приводит нас к следующему определению натурального числа N. Число N представляется функцией с двумя аргументами, которая выполняет N-кратное применение своего первого аргумента ко второму. Более точно, сначала запишем определение для функции ZERO, представляющей число нуль, а затем покажем, как определяется число N+1, если число N уже определено. Тем самым будет возможно построить определение любого натурального числа N.

```
ZERO = λf.λx.x
```

функция f ноль раз применяется к аргументу x, так что аргумент возвращается неизменным

```
(N+1) = λf.λx.f (N f x)
```

тело функции (N+1) представляет собой однократное применение аргумента f к N-кратному применению функции f к аргументу x.

Заметим, кстати, что константа ZERO определена в точности так же, как константа FALSE, однако далеко идущих выводов из этого сходства делать все же не следует. Из приведенного определения целых чисел немедленно следует определение функции следования, которая добавляет единицу к своему аргументу:

```
SUCC = λn.λf.λx.f (n f x)
```

Теперь нетрудно также определить операции сложения и умножения целых чисел. Так, например, можно сказать, что для того, чтобы сложить два числа m и n, надо m раз увеличить число n на единицу. Аналогично, чтобы умножить m на n, надо m раз применить функцию увеличения на n к нулю. Отсюда следуют определения:

```
PLUS = λm.λn.m SUCC n
MULT = λm.λn.m (PLUS n) ZERO
```

К сожалению, определить операции вычитания в представленной нами арифметике совсем не так просто. Прежде всего, определим функцию

PRED, которая выдает предыдущее натуральное число для всех чисел, больших нуля, а для аргумента, равного нулю, выдает также ноль. Одно из возможных определений такой функций выглядит следующим образом:

```
PRED = λn.λf.λx.n (λg.λh.h (g f)) (λu.x) (λu.u)
```

Проверьте, что применение этой функции к значению ZERO действительно выдает в качестве результата ZERO, а применение этой функции к, скажем, значению 2 (представленному функцией λf.λx.f (f x)) может быть преобразовано к значению 1 (функции λf.λx.f x). Такое упражнение позволит вам понять, как происходит уменьшение количества применений функции f к аргументу. Теперь уже нетрудно определить и функцию вычитания на множестве натуральных чисел, которая для заданных аргументов m и n выдает значение (m−n) при m n и выдает ноль при m < n.

```
MINUS = λm.λn.n PRED m
```

Арифметику целых чисел можно теперь связать с логическими значениями, определив функции сравнения чисел. Проще всего определить функцию сравнению числа с нулем. Вот одно из возможных определений для такой функции.

```
EQ0 = λn.n (λx.FALSE) TRUE
```

Очевидно, что если применить функцию к значению ZERO, то функция (λx.FALSE) не будет применена ни разу к своему аргументу, поэтому результатом применения функции будет значение TRUE. Если же значение аргумента отлично от нуля, то функция (λx.FALSE) после первого же применения выдаст значение FALSE, и в дальнейшем, сколько бы раз ее ни применять, выдаваемым значением так и будет оставаться FALSE. Теперь можно определить функции сравнения двух чисел с помощью операций GE ("больше или равно") и LE ("меньше или равно"), используя только что определенную операцию сравнения с нулем и операцию вычитания натуральных чисел MINUS.

```
GE = λm.λn.EQ0 (MINUS m n)
LE = λm.λn.EQ0 (MINUS n m)
```

Остальные операции сравнения легко выражаются через эти операции сравнения и уже определенные ранее логические операции:

```
GT = λm.λn.NOT (LE m n)
LT = λm.λn.NOT (GE m n)
EQ = λm.λn.AND (GE m n) (LE m n)
NEQ = λm.λn.NOT (EQ m n)
```

Теперь, когда уже определена арифметика и логика, давайте, вооруженные опытом описания различных стандартных функций, построим функции для формирования составных значений, подобных кортежам или спискам. Представляемые нами составные значения будут

больше всего напоминать списки, как они определены в языке *LISP*, то есть списки элементов самого разного типа (элементом такого списка может быть любое значение, в том числе другой список, число, логическое значение или, вообще говоря, любая функция). Для создания таких списков необходимо определить одно базовое значение – пустой список – и функции, позволяющие из двух заданных значений создавать их пару (функция CONS), а также выделять первый и второй элемент пары (функции HEAD и TAIL). Для того, чтобы можно было исследовать списки в программах, требуется определить также, по крайней мере, одну логическую функцию NULL, которая выдает значение TRUE или FALSE в зависимости от того, является ли ее аргумент пустым списком или нет.

Функция CONS может соединять в пару два своих аргумента, просто создавая функцию, которая будет применять свой аргумент к обеим частям пары как к двум своим аргументам. Другими словами, если H и T - два произвольных значения, то их пару можно представить функцией (λs.s H T). Таким образом, функция CONS получает следующее определение:

```
CONS = λh.λt.λs.s h t
```

Для того, чтобы из составного значения (λs.s H T) выделить первый элемент H, надо применить эту функцию к такому значению s, которое выдаст первый из двух своих аргументов. Таким значением является уже определенная нами ранее константа TRUE. Ясно, что результатом применения (λs.s H T) TRUE будет значение H. Таким образом, функция HEAD получает следующее определение:

```
HEAD = λl.l TRUE
```

Разумеется, функция будет работать "правильно", только если ее аргументом будет пара, составленная с помощью функции CONS. Аналогичным образом определяется и функция TAIL.

```
TAIL = λl.l FALSE
```

Функция NULL должна выдавать в качестве результата FALSE для любой пары вида (λs.s H T). Поэтому можно определить эту функцию таким образом, чтобы она применяла свой аргумент к функции, выдающей значение FALSE для любых двух аргументов: λh.λt.FALSE. Отсюда следует определение

```
NULL = λl.l (λh.λt.FALSE)
```

Однако, та же функция должна выдавать в качестве результата TRUE, если ее аргументом будет пустой список. Отсюда можно легко вывести простой способ представления пустого списка NIL:

```
NIL = λx.TRUE
```

Давайте проверим, что наши определения работают, на простом примере: проверим, что значением выражения

NULL (HEAD (CONS NIL A)) будет TRUE, каково бы ни было выражение A. Для этого будем последовательно подставлять в наше выражение определения введенных нами функций и производить редукции выражения по мере возможности. В результате получим следующую последовательность эквивалентных выражений.

```
NULL (HEAD (CONS NIL A))
(λl.l (λh.λt.FALSE)) (HEAD (CONS NIL A))
(HEAD (CONS NIL A)) (λh.λt.FALSE)
((λl.l TRUE) (CONS NIL A)) (λh.λt.FALSE)
((CONS NIL A) TRUE) (λh.λt.FALSE)
(((λh.λt.λs.s h t) NIL A) TRUE) (λh.λt.FALSE)
((λs.s NIL A) TRUE) (λh.λt.FALSE)
(TRUE NIL A) (λh.λt.FALSE)
NIL (λh.λt.FALSE)
(λx.TRUE) (λh.λt.FALSE)
TRUE
```

Мы получили ожидаемый результат. Подобным же образом можно представить в чистом лямбда-исчислении любое выражение, в том числе содержащее рекурсивные вызовы (применяя преобразование, использующее Y-комбинатор). Таким образом, можно сказать, что мы получили возможность эквивалентного представления любой функциональной программы, написанной в нашем расширенном лямбда-исчислении, с помощью лишь средств чистого лямбда-исчисления.

На самом деле использовать чистое лямбда-исчисление в практических целях, конечно же, неудобно. Даже для того, чтобы сложить числа 2 и 5, потребуется выполнить огромное количество редукций, а реализация рекурсии в виде применения Y-комбинатора – это тоже далеко не самый простой способ выполнения рекурсивных функций. Поэтому в дальнейшем мы будем использовать только расширенное лямбда-исчисление, содержащее встроенные константы и функции, а также конструкцию let для эффективной реализации механизма ленивых вычислений. Более того, мы еще расширим наше лямбда-исчисление для того, чтобы явно представить в нем рекурсию, что позволит нам обойтись без применения Y-комбинатора. Новая конструкция по виду и по действию будет напоминать конструкцию let, разница будет состоять только в том, что в новой конструкции letrec x = e1 in e2 в выражении e1 допускаются рекурсивные обращения к переменной x. Это, в частности, приводит к тому, что новая конструкция уже не может быть представлена в виде эквивалентного выражения (λx.e2) e1, так как в выражении аргумента e1 могут встречаться обращения к связанной переменной x. Эквивалентное выражение в чистом лямбда-исчислении может быть получено путем применения Y-комбинатора.

Рассмотрим правила редукций, применяемые в условиях существования рекурсивных определений, на примере преобразования выражения

```
letrec FACT = λn.IF(EQ0 n) 1 (MULT n (FACT(PRED n))) in
                  FACT 2
letrec FACT = λn.IF (EQ0 n) 1 (MULT n (FACT (PRED n))) in
       (λn.IF (EQ0 n) 1 (MULT n (FACT (PRED n)))) 2
letrec FACT = λn.IF (EQ0 n) 1 (MULT n (FACT (PRED n))) in
       IF (EQ0 2) 1 (MULT 2 (FACT (PRED 2)))
letrec FACT = λn.IF (EQ0 n) 1 (MULT n (FACT (PRED n))) in
       MULT 2 (FACT 1)
letrec FACT = λn.IF (EQ0 n) 1 (MULT n (FACT (PRED n))) in
       MULT 2 ((λn.IF (EQ0 n) 1 (MULT n (FACT (PRED n)))) 1)
letrec FACT = λn.IF (EQ0 n) 1 (MULT n (FACT (PRED n))) in
       MULT 2 (IF (EQ0 1) 1 (MULT 1 (FACT (PRED 1))))
letrec FACT = λn.IF (EQ0 n) 1 (MULT n (FACT (PRED n))) in
       MULT 2 (MULT 1 (FACT 0))
letrec FACT = λn.IF (EQ0 n) 1 (MULT n (FACT (PRED n))) in
       MULT 2 (MULT 1 ((λn.IF (EQ0 n) 1
                              (MULT n (FACT (PRED n)))) 0))
letrec FACT = λn.IF (EQ0 n) 1 (MULT n (FACT (PRED n))) in
       MULT 2 (MULT 1 (IF (EQ0 0) 1 (MULT 0 (FACT (PRED 0)))))
letrec FACT = λn.IF (EQ0 n) 1 (MULT n (FACT (PRED n))) in
       MULT 2 (MULT 1 1)
MULT 2 (MULT 1 1)
2
```

В этом примере в конструкции `letrec x = e1 in e2` выражение `e1` уже находится в СЗНФ, поэтому при вычислении выражения `e2` происходит просто подстановка значения `e1` в выражение `e2` вместо `x`. Однако, поскольку это выражение содержит рекурсивные обращения к определяемой переменной (в данном примере – `FACT`), то для дальнейшего вычисления требуется сохранить связь переменной со значением `e1`. Эта связь исчезает только тогда, когда в результате преобразований из выражения `e2` исчезает сама переменная `x`.

В следующей главе мы будем использовать наше расширенное лямбда-исчисление в роли простого языка функционального программирования для того, чтобы более точно исследовать семантику исполнения функциональных программ. При этом мы не будем заниматься чисто текстовыми преобразованиями выражений, как мы это проделывали на протяжении всей этой главы. Вместо этого мы будем использовать более традиционные для программирования способы хранения и преобразования значений.

Примеры решения задач

Задание 1. Для чистого лямбда-исчисления, представленного в курсе лекций покажите, что

```
AND TRUE TRUE = TRUE
```

Решение. Выражения для функций `TRUE` и `AND` представлены следующим образом.

```
TRUE = λx.λy.x
AND = λx.λy.x y (λx.λy.y)
```

Таким образом, указанное выражение в лямбда-исчислении будет выглядеть следующим образом.

```
(λx.λy.x y (λx.λy.y)) (λx.λy.x) (λx.λy.x)
```

Выполним преобразования этого выражения в НПР.

```
(λx.λy.x y (λx.λy.y)) (λx.λy.x) (λx.λy.x)
(λy.(λx.λy.x) y (λx.λy.y)) (λx.λy.x)
(λx.λy.x) (λx.λy.x) (λx.λy.y)
(λy.(λx.λy.x)) (λx.λy.y)
λx.λy.x
```

Получили выражение, эквивалентное определению значения `TRUE`.

Задание 2. Определите функцию возведения в степень для чистого лямбда-исчисления.

Решение. Выражние для функции возведения в степень получается из функции умножения по аналогии с тем, как функция умножения получается из функции сложения. Напомним определение функции умножения:

```
MULT = λm.λn.m (PLUS n) 0
```

Аналогично можно определить и функцию возведения в степень:

```
POWER = λm.λn.n (MULT m) 1
```

Задание 3. Определите в чистом лямбда-исчислении функцию вычисления арифметического квадратного корня, то есть для заданного аргумента `n` функция должна вычислять такое значение `q`, что $q^2 <= n$, но $(q+1)^2 > n$.

Решение. Несложно написать рекурсивную функцию для вычисления квадратного корня из числа. Если не заботиться об эффективности вычислений, то искомая функция на языке *Haskell* может выглядеть следующим образом:

```
sqrt n = sqrt1 0 n
sqrt1 q n | (q+1)*(q+1) > n = q
          | otherwise       = sqrt1 (q+1) n
```

Если переписать эту функцию на язык расширенного лямбда-исчисления, то получится что-то вроде

```
sqrt1 = λq.λn.IF (GT (MULT (SUCC q) (SUCC q)) n)
```

```
                q (sqrt1 (SUCC q) n)
sqrt = sqrt1 0
```

Здесь надо еще избавиться от рекурсии и «лишних» имен функций. Если избавиться от рекурсии с помощью Y-комбинатора и подставить полученную рекурсивную функцию в определение основной функции `sqrt`, то получится следующее выражение:

```
(Y λs.λq.λn.IF (GT (MULT (SUCC q) (SUCC q)) n)
               q (s (SUCC q) n)) 0
```

На самом деле все идентификаторы стандартных функций, встречающихся в этом выражении, были определены нами ранее в чистом лямбда-исчислении, так что можно сказать, что полученное выражение и является решением задачи. Можно попробовать перейти к чистому лямбда-исчислению без использования идентификаторов констант почти чисто механически, подставив вместо констант и стандартных функций их определения в чистом лямбда исчислении:

```
0      = λf.λx.x
1      = λf.λx.f x
SUCC   = λn.λf.λx.f (n f x)
PLUS   = λm.λn.m SUCC n
MULT   = λm.λn.m (PLUS n) 0
TRUE   = λx.λy.x
FALSE  = λx.λy.y
NOT    = λx.x FALSE TRUE
PRED   = λn.λf.λx.n (λg.λh.h (g f)) (λu.x) (λu.u)
MINUS  = λm.λn.n PRED m
EQ0    = λn.n (λx.FALSE) TRUE
LE     = λm.λn.EQ0 (MINUS n m)
GT     = λm.λn.NOT (LE m n)
Y      = λh.(λx.h (x x))(λx.h (x x))
IF     = λp.λq.λr.p q r
```

Правда, если делать подстановку действительно чисто механически, то выражение получится очень длинным. На самом деле многие из стандартных функций встречаются в нашей функции только в их применении к достаточно простым аргументам. В этом случае можно выполнить некоторые из редукций, подставив аргументы в тело функций. Так, например, выражение `SUCC q` превратится в `λf.λx.f (q f x)`, а идентификатор функции `IF` просто исчезнет. К сожалению, выражение все равно получается слишком длинным. Один из вариантов такого выражения представлен здесь, однако, форма представления выражения может быть разной.

```
((λh.(λx.h (x x))(λx.h (x x))) λs.λq.λn.((λm.λn.((n λn.λf.λx.n
(λg.λh.h (g f)) (λu.x) (λu.u) m) (λx.λx.λy.y) λx.λy.x) λx.λy.y
λx.λy.x) ((λf.λx.f (q f x)) (λn.(λf.λx.f (q f x) (λn.λf.λx.f
(n f x)) n) (λf.λx.x)) n) q (s (λf.λx.f (q f x)) n)) (λf.λx.x)
```

Задания для самостоятельной работы

Задание 1. Определите для чистого лямбда-исчисления логическую функцию `XOR` (исключающее ИЛИ).

Задание 2. В чистом лямбда-исчислении определите функцию вычисления остатка от деления одного неотрицательного целого числа на другое (положительное) число.

Задание 3. В чистом лямбда-исчислении определите функцию вычисления длины заданного списка.

Задание 4. Будем считать, что элементами некоторого списка могут быть только другие списки (возможно, пустые). Таким образом, список приобретает структуру дерева, листьями которого будут пустые списки, а непустые списки будут представлять собой вершины, для которых элементы списка будут поддеревьями. Определите в чистом лямбда-исчислении функцию вычисления высоты такого «дерева».

Глава 4. Системы исполнения функциональных программ

4.1. Промежуточный язык программирования

В этой главе мы построим несколько программ для исполнения кода, написанного на функциональных языках, для того, чтобы более тщательно исследовать вопрос о том, как исполняются функциональные программы. До сих пор мы лишь на словах описывали, как следует понимать те или иные конструкции языка *Haskell* или иного языка функционального программирования. Для того чтобы более точно описать семантику исполнения, следует написать какую-либо программу, способную исполнять предъявленный ей функциональный код. Разумеется, такие программы мы будем описывать на языке *Haskell*, этот язык достаточно удобен для того, чтобы на нем записывать сложные алгоритмы. Однако в качестве исходного, интерпретируемого языка язык *Haskell* не очень хорошо приспособлен для демонстрационных целей. В нем слишком много различных конструкций, многие из которых являются просто удобными синтаксическими сокращениями для других, может быть, более общих конструкций (это то, что системные программисты обычно называют «синтаксическим сахаром»). Чтобы не перегружать наши программы излишними деталями, мы возьмем в качестве исходного языка расширенное лямбда-исчисление, достаточно простое и в то же время достаточно мощное для того, чтобы можно было на нем выразить любые конструкции функциональных языков программирования.

Прежде всего, поскольку в наших построениях программы, записанные на языке расширенного лямбда-исчисления, являются предметом обработки других программ, следует подумать о том, как удобнее всего представлять конструкции расширенного лямбда-исчисления в наших программах.

Конечно, можно просто считать, что программа на языке расширенного лямбда-исчисления будет представлена строкой текста, однако такое представление, несмотря на его большую общность, является очень неудобным для обработки. Обычно перед тем, как начать исследовать или исполнять некую программу, ее переводят в некоторую промежуточную форму, представляющее собой *синтаксическое дерево* исходной программы. Мы не будем описывать процесс перевода текста программы в синтаксическое дерево, интересующиеся этим процессом студенты могут обратиться к многочисленной литературе по устройству и построению синтаксических анализаторов в компиляторах языков программирования. Мы же опишем только само синтаксическое дерево – результат синтаксического анализа выражения, записанного в расширенном лямбда-исчислении.

Описание конструкций синтаксического дерева можно сделать в виде описания нового типа данных языка *Haskell*, в котором определены все конструкции расширенного лямбда-исчисления. Поскольку эти конструкции определены рекурсивно, то и определяемый тип данных тоже, конечно, будет рекурсивным. Мы определим несколько конструкторов для этого типа данных в соответствии с тем, какие конструкции имеются в расширенном лямбда-исчислении. Сам определеяемый тип данных будет называться `Expr` – от *Expression* – выражение.

Внесем еще одно небольшое изменение в конструкции нашего расширенного лямбда-исчисления. Будем считать, что в блоках `let` и `letrec` может быть определено сразу несколько идентификаторов. Например, блок `let` будет иметь следующий общий вид:

```
let x1 = e1,
    x2 = e2,
    ...
    xk = ek in e
```

так что в блоке определяются сразу несколько идентификаторов `x1`, `x2`,... `xk` и связываются с выражениями `e1`, `e2`,... `ek` соответственно. Выражение `e` теперь может содержать любые из идентификаторов `x1`, `x2`,... `xk`, однако в самих выражениях `e1`, `e2`,... `ek` переменные `x1`, `x2`,... `xk` встречаться не могут. Аналогично будет выглядеть и рекурсивный блок `letrec` с той разницей, что в нем уже в выражениях `e1`, `e2`,... `ek` могут встречаться рекурсивные обращения к переменным `x1`, `x2`,... `xk`:

```
letrec x1 = e1,
       x2 = e2,
       ...
       xk = ek in e
```

Введенные изменения не сильно сказываются на сложности конструкций, однако, позволяют напрямую использовать косвенную рекурсию с помощью блока `letrec`, а также вводить сразу несколько определений переменных с помощью блока `let`.

Итак, теперь всего в расширенном лямбда-исчислении имеются 5 видов конструкций: константы (мы будем различать три вида констант – целые числа, логические значения и встроенные функции; следовало бы еще ввести константу для представления пустого списка, однако, мы для простоты не будем этого делать), переменные, лямбда-выражения, применения функции и блоки (опять, будем различать два вида блоков – рекурсивный, представленный конструкцией `letrec` и нерекурсивный, представленный конструкцией `let`). Соответственно, описание типа лямбда-выражений в виде синтаксического дерева будет содержать 8 конструкторов для описания каждого из типов и подтипов выражений.

```
data Expr =
     Integral Integer |  -- целые константы
     Logical Bool |      -- логические константы
     Function String |   -- идентификаторы примитивных функций
     Variable String  |  -- переменная
     Lambda   String Expr  |       -- лямбда-выражение
     Application Expr Expr |       -- применение функции
     Let [(String, Expr)] Expr |   -- простой блок
     Letrec [(String, Expr)] Expr  -- рекурсивный блок
```

Каждый из конструкторов имеет один или несколько аргументов в соответствии с тем, какие подконструкции содержатся в каждой из конструкций расширенного лямбда-исчисления, а для элементарных конструкций аргументы конструкторов содержат внешнее представление конструкции. Так, например, конструктор `Variable` имеет в качестве аргумента `String`, поскольку переменная представлена некторой строкой – идентификатором этой переменной, а конструктор `Application` имеет два аргумента, соответствующих своим подконструкциям – выражение, задающее вызываемую функцию, и выражение, задающее аргумент вызова. Приведем пример несложного лямбда-выражения и его представления в виде выражения типа `Expr` на языке *Haskell*. Пусть исходное лямбда-выражение имеет вид

```
let sqr = λx.* x x in sqr 5
```

Тогда соответствующее синтаксическое дерево может быть представлено следующим выражением на языке *Haskell*:

```
Let [("sqr",
      (Lambda "x"
              (Application (Application (Function "*")
                                        (Variable "x"))
                           (Variable "x"))))]
    (Application (Variable "sqr") (Integral 5))
```

Правила перевода с языка расширенного лямбда-исчисления в выражение типа `Expr` можно записать и более формально, это совсем простое упражнение. Более интересным является вопрос о переводе конструкций такого сложного языка функционального программирования, каким является язык *Haskell*, в более простой язык расширенного лямбда-исчисления. Мы не будем здесь приводить все детали такой трансформации, тем более, не будем пытаться построить программу, которая может осуществить такое преобразование, однако, наметим основные вехи на пути построения такого преобразования.

Прежде всего, давайте условимся, что мы будем рассматривать только правильные программы на языке *Haskell*, так что будем считать, что в программе не только нет грубых синтаксических ошибок, но и анализ типов данных уже произведен, и ошибок, связанных с неправильным употреблением типов, в программе тоже нет. Таким образом, мы сразу же

выводим из рассмотрения все конструкции, необходимые только для определения типов данных и описания типов. Мы будем считать, что нам не нужно рассматривать определения классов, конструкции, задающие описания типов тех или иных переменных, задание контекста, конструкцию `type` и пр. Конструкция `data` будет представлять для нас интерес только постольку, поскольку в ней определяются конструкторы данных, которые впоследствии могут использоваться для построения объектов и в конструкции сопоставления с образцом. Мы можем также проигнорировать модульную структуру программы, и считать, что программа на языке *Haskell* состоит просто из определений одного уровня.

Определения, из которых состоит программа на языке *Haskell*, – это определения функций и значений других типов. Можно считать, что исполнение программы состоит в вычислении некоторого выражения в контексте, заданном этими определениями. Таким образом, для того, чтобы выполнить программу, надо вычислить выражение вида

```
letrec x1 = e1,
       x2 = e2,
       ...
       xk = ek in e
```

где выписанный блок как раз и представляет собой набор, вообще говоря, рекурсивных определений переменных и некоторое выражение, которое необходимо вычислить. Правда, в языке *Haskell* определения функций могут быть записаны не только в виде `xi = ei`, но также и в виде набора уравнений, в которых встречаются сопоставления с различными образцами. Надо сказать, что сопоставление с образцом – это практически единственная конструкция в языке *Haskell*, которая не имеет прямого аналога в расширенном лямбда-исчислении и потому требует особого обсуждения. Перед тем, как показать, как именно сопоставление с образцом может быть переведено в конструкции расширенного лямбда-исчисления, коротко рассмотрим все остальные конструкции языка *Haskell*.

Выражения в *Haskell* строятся из простых значений, которыми являются числа, логические константы и изображения списков. Числа и логические значения мы считаем константами нашего расширенного лямбда-исчисления, так что они не требуют никакого перевода. Что же касается пустого списка и более сложных конструкций в изображении списков, таких как `[1, 2, 5]`, `[1..]` или даже `[x*x | x <- [1..10]]`, то их можно выразить с помощью применения стандартных конструкторов, стандартных функций или функций, которые легко определить самому. Поэтому можно считать, что такие изображения сложных списков являются частным случаем конструкций применения функций и конструкторов к более простым значениям.

К применению тех или иных функций сводятся и многие другие конструкции языка *Haskell*. Так, например, образование кортежа из `k` значений может быть выражено применением стандартной функции `TUPLE-k`, описанной нами в предыдущей главе, так что, например, изображение кортежа `(1, x, [])` может быть представлено в расширенном лямбда-исчислении в виде применения функции `TUPLE-3 1 x NIL`. Здесь `NIL` – это представление для пустого списка, чуть ниже мы рассмотрим более подробно это представление. Условное выражение `if b then e1 else e2` также представляется в виде применения стандартной функции `IF b e1 e2`. Несколько особое место занимает представление применения стандартных и определенных программистом конструкторов. С помощью конструкторов создаются новые значения, представляющие собой чисто механическое соединение нескольких значений определенного типа наподобие того, как это делается при образовании кортежа. Дополнительно к этому, конструктор определяет еще и *тэг* – имя конструктора, которое затем может быть использовано при сопоставлении с образцом. Будем считать, что при анализе определения типа данных все конструкторы получают целые номера, начинающиеся с нуля. Так, например, если тип данных *двоичное дерево* был определен с помощью описания

```
data Tree a = Null |
              Tree a (Tree a) (Tree a)
```

то конструктор `Null` получает номер 0, а конструктор `Tree` – номер 1. Аналогично и для стандартного конструктора списков, про который можно сказать, что его определение выглядит как

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

определены два конструктора: конструктор пустого списка `[]`, получающий номер 0, и конструктор `(:)`, получающий номер 1. Поскольку мы условились, что программа уже прошла анализ правильности, и ошибок в использовании типов данных в ней нет, то номера конструкторов могут быть локальными для каждого определения типа данных – путаницы, каким тегом какой конструктор помечен, не возникнет.

Теперь давайте считать, что применение любого конструктора эквивалентно образованию кортежа, в который входят как минимум тэг применяемого конструктора, а также все значения аргументов. Учитывая каррингговость конструкторов (как и прочих функций), можно сказать, что представлением любого конструктора языка *Haskell* в виде выражения расширенного лямбда-исчисления является выражение `TUPLE-k n`, где `k` - число аргументов конструктора, увеличенное на единицу, а `n` – номер (тэг) конструктора. Таким образом, выражение `[10, 15]`, которое может

быть записано с помощью применения стандартных конструкторов в виде `10:(15:[])`, будет представлено следующим выражением расширенного лямбда-исчисления:

```
TUPLE-3 1 10 (TUPLE-3 1 15 (TUPLE-1 0))
```

(напомним еще раз, что конструктор `(:)` имеет тэг 1 и два аргумента, а конструктор `[]` – тэг 0 и ни одного аргумента). Таким образом представление пустого списка, который мы несколькими строками выше записывали в виде `NIL`, теперь будет выглядеть как `(TUPLE-1 0)`.

В языке *Haskell* имеются также лямбда-выражения, которые несколько отличаются от лямбда-выражений в расширенном лямбда-исчислении. Однако отличия эти не очень существенны. Первое отличие состоит в том, что лямбда-выражения языка *Haskell* могут иметь более одного аргумента. Например, обычным представлением функции сложения в *Haskell* будет лямбда-выражение `\x y -> x+y`. Конечно же, такое лямбда-выражение можно легко переписать так, чтобы в лямбда-выражениях был только один аргумент: `\x -> \y -> x+y`. Второе отличие состоит в том, что в качестве формального аргумента в лямбда-выражении может использоваться не только имя переменной, но и образец, с которым производится сопоставление. Это отличие более существенно, и мы его рассмотрим вместе с конструкцией сопоставления с образцом в определении уравнений для некоторой функции.

Теперь осталось рассмотреть только одну конструкцию, которая еще не получила своего эквивалентного выражения в расширенном лямбда-исчислении. Это конструкция сопоставления с образцом в определении функции. Сначала надо понять, что, собственно, представляет собой сопоставление с образцом и как оно используется. На самом деле конструкция сопоставления с образцом несет на себе две основные нагрузки: выбор одного из уравнений при вызове функции и связывание частей фактического аргумента с переменными, входящими в состав образца. Пусть, например, имеются *n* уравнений, определяющих функцию `f`. Припишем каждому из уравнений некоторый номер, начиная с нуля и до (*n*-1). Не умаляя общности, можно считать, что функция имеет только один аргумент; если это не так, то определение можно разбить на несколько определений, в каждом из которых имеется только один аргумент. Таким образом, в самом общем виде мы можем записать следующие уравнения.

```
f <образец 0> | <условие 0> = <правая часть 0>
f <образец 1> | <условие 1> = <правая часть 1>
...
f <образец (n-1)> | <условие (n-1)> = <правая часть (n-1)>
```

Мы должны для функции `f` построить эквивалентное определение в расширенном лямбда-исчислении, которое будет иметь вид `f = λx.e`. Очевидно, что выражение для тела функции должно содержать в себе коды для проверки соответствия аргумента каждому из образцов уравнений (*код*

соответствия образцу), коды для связывания переменных образцов с нужными частями аргумента `x` (*связывающий код*), и выражения лямбда-исчисления, эквивалентные условиям `<условие i>` (*код условия*) и правым частям уравнений `<правая часть i>` (*код правой части*). Сначала свяжем с каждым из уравнений, определяющих функцию `f`, некоторую вспомогательную функцию, осуществляющую все действия по проверке соответствия с образцом, проверке условия и вычисления значения функции в соответствии с правой частью этого уравнения (*код уравнения*). После этого соединим полученные определения в одно определение функции.

Введем следующие имена вспомогательных функций для заданной функции `f`. Идентификаторами `f_0`, `f_1`,... `f_(n-1)` обозначим функции, соответствующие кодам каждого из уравнений исходной функции `f`. Аналогично, идентификаторы `f_i_pattern` будут обозначать коды проверки соответствия каждому из образцов уравнений, идентификаторы `f_i_bind` будут обозначать связывающие коды каждого из уравнений, `f_i_test` – коды проверки соответствующих условий и `f_i_right` – коды правых частей уравнений. Тогда в выражении `f = λx.e` код тела функции `e` будет представлять собой последовательную проверку аргумента `x` на соответствие образцам и условиям уравнений функции `f`, и его можно записать в расширенном лямбда-исчислении следующим образом.

```
f = λx.letrec f_0 = λx.if (f_0_pattern x)
                          (let <f_0_bind x> in
                               if (f_0_test x)
                                  (f_0_right x)
                                  (f_1 x)
                          )
                          (f_1 x),
              f_1 = λx.if (f_1_pattern x)
                          (let <f_1_bind x> in
                               if (f_1_test x)
                                  (f_1_right x)
                                  (f_2 x)
                          )
                          (f_2 x),
              ...
              f_(n-1) = λx.if (f_(n-1)_pattern x)
                              (let <f_(n-1)_bind x> in
                                   if (f_(n-1)_test x)
                                      (f_(n-1)_right x)
                                      error
                              )
                              error,
      in f_0(x)
```

Сделаем необходимые пояснения. Выполнение функции `f` начинается с того, что вызывается функция проверки соответствия аргумента первому уравнению `f_0`. Эта функция прежде всего проверит, соответствует ли аргумент первому образцу (вызов `f_0_pattern x`). Если аргумент не удовлетворяет образцу, то будет вызвана функция второго уравнения `f_1` и так далее, пока либо не будет обнаружено соответствие, либо ни одно из уравнений не будет признано подходящим, и в этом последнем случае будет сделано обращение к функции `error`, вызывающей аварийное окончание работы программы. Далее, если соответствие образцу обнаружено, то выполняется связывающий код, в котором каждой из переменных образца ставится в соответствие некоторая часть аргумента. Этот код представляет собой часть блока `let` и записан в нашем выражении в виде <`f_i_bind x`>. После выполнения связывания производится еще одна проверка – проверка на соответствие аргумента условию охраны, если, конечно, охрана имеется в уравнении (вызов `f_i_test x`). Это условие проверяется в ситуации, когда уже выполнен связывающий код, и поэтому находится внутри тела соответствующего блока. Если условие не выполняется, то будет продолжена проверка уравнений, если же условие выполнено, то исполняется код, соответствующий правой части соответствующего уравнения (вызов `f_i_right x`), и на этом работа функции будет завершена.

Теперь осталось понять, как должны быть написаны все коды, упомянутые, но не реализованные в нашем определении функции `f`: коды соответствия образцу `f_i_pattern`, связывающие коды `f_i_bind`, коды условий `f_i_test` и коды правых частей `f_i_right`. Что касается кодов условий и кодов правых частей, то здесь никаких особенностей в реализации нет. Соответствующие выражения из определяющих уравнений просто переводятся на язык расширенного лямбда-исчисления в соответствии с правилами перевода. Наиболее интересным является вопрос о том, как происходит проверка соответствия образцу в кодах `f_i_pattern` и как организовано связывание переменных в кодах `f_i_bind`.

Прежде всего, рассмотрим структуру образца. Напомним, что образцом может быть либо простая переменная (именованная или безымянная), либо константа, либо вызов конструктора, аргументами которого являются другие образцы, либо, наконец, кортеж из других образцов. На самом деле, константы также можно рассматривать как вызовы конструкторов без аргументов, так что фактически мы имеем только два вида образцов: образец-переменная и образец-вызов конструктора, однако, все-таки обычно константы представлены иначе, чем результаты применения функций-конструкторов, так что этот случай тоже лучше рассматривать отдельно. Если образцом является переменная,

то код соответствия образцу для любого аргумента выдает истинное значение, так как такому образцу соответствует любое значение аргумента. Связывающий код в этом случае также очень прост. Итак, если в некотором *i*-ом уравнении образцом является переменная `z`, то код соответствия образцу `f_i_pattern` выглядит как функция `λx.TRUE`, а связывающий код `f_i_bind` – как сопоставление `z = x`. Кстати, если переменная образца – безымянная, то связывающий код не нужен вовсе.

Если образцом является константа, то код сопоставления – это просто проверка того факта, что аргументом является то же самое значение, что и значение, представленное этой константой. В этом случае код соответствия образцу будет выглядеть как функция `f_i_pattern = λx.EQ c x`, где `c` - константа образца, а `EQ` – функция сравнения с константой соответствующего типа. Связывающий код для образца-константы не нужен, поскольку никаких переменных такой образец не содержит.

Код сопоставления с образцом в случае, когда образцом является кортеж, просто сводится к кодам сопоставления с образцами, составляющими кортеж. Просто надо перед всеми сопоставлениями с элементами кортежа выделить соответствующие части кортежа из фактического аргумента функции. Таким образом, и в коде проверки соответствия образцу, и в коде сопоставления просто появляется дополнительное действие – выделение элемента кортежа из аргумента с помощью применения стандартной функции `INDEX`.

Самым сложным случаем является случай образца-конструктора. Поскольку программа уже прошла проверку соответствия типам, то аргументом функции в этом случае может быть только значение, имеющее вид кортежа, первым элементом которого является номер конструктора. В коде сопоставления с образцом надо проверить, действительно ли первый элемент кортежа совпадает с номером конструктора из рассматриваемого образца, а если это действительно так, то далее надо проверить, что все остальные элементы кортежа соответствуют тем образцам, которые являются аргументами образца-конструктора. Поскольку элементы кортежа могут быть выбраны с помощью стандартной функции `INDEX`, то проверка соответствия номера конструктора первому элементу кортежа будет выглядеть следующим образом: `EQ c (INDEX 0 x)`, где `c` – номер конструктора в образце. Далее надо выбрать оставшиеся элементы кортежа с помощью вызовов `(INDEX i x)` при *i* равных 1, 2,... и применить коды сопоставления с соответствующими подобразцами для этих выбранных частей образца. Сам образец-конструктор содержит переменные только в своих подобразцах, так что связывающего кода порождать для образца в целом нет необходимости за исключением случая, когда некоторая переменная связывается с образцом в целом с помощью конструкции `z@pattern`, где `z` - имя переменной, а `pattern` -

образец. В этом случае в связывающий код надо поместить соответствующее уравнение `z = x`.

Приведем пример. Пусть одно из уравнений для функции `f` имеет вид

```
f lst@[t] = e
```

Прежде всего, перепишем образец таким образом, чтобы в нем в явном виде присутствовали все конструкторы:

```
f lst@(t : []) = e
```

Теперь понятно, что в коде сопоставления с образцом надо будет последовательно сделать две проверки: сначала проверить, что аргумент действительно сформирован с помощью конструктора `(:)`, а если это действительно так, то надо будет проверить, что третий элемент кортежа сформирован с помощью конструктора `[]`. Если считать, что номер конструктора `[]` – это ноль, а номер конструктора `(:)` – единица, то функция, реализующая код сопоставления, будет иметь вид:

```
λx.CAND (EQ 1 (INDEX 0 x))
        (let test1 = λx.(EQ 0 (INDEX 0 x)) in
             test1 (INDEX 2 x))
```

В этом примере функция состоит в применении операции условного логического «и» для проверки соответствия аргумента внешнему образцу и, если проверка будет закончена успешно, для проверки соответствия частей аргумента подобразцам. Функция `test1` вводится во внутреннем блоке для проверки соответствия части аргумента единственному подобразцу, содержащему конструктор `[]`. Эта функция применяется к части аргумента, выделенной с помощью применения стандартной функции `(INDEX 2 x)`.

Связывающий код для этого примера будет содержать связывание для двух переменных, содержащихся в образце. Переменная `lst` относится ко всему образцу, поэтому соответствующий связывающий код будет иметь вид `lst = x`. Переменная `t` соответствует части аргумента – второму элементу соответствующего кортежа, так что код связывания для нее имеет вид `t = INDEX 1 x`. Полностью заголовок блока, содержащий связывающий код, будет иметь вид

```
let l = x,
    t = INDEX 1 x
in ...
```

Итак, мы научились переводить любые конструкции программ на языке *Haskell* в выражения расширенного лямбда-исчисления. Конечно, множество деталей требуют более тщательной проработки, однако, все принципиальные вопросы решены. Теперь можно заняться исследованием программ, написанных на этом простом языке для того, чтобы понять точный смысл исполнения всех конструкций этого языка.

4.2. eval/apply интерпретатор

Прежде всего, попробуем выразить семантику языка расширенного лямбда-исчисления с помощью написания интерпретатора программ, написанных на этом языке. Для этого мы построим модель вычислений, то есть напишем программу, которая, получив в качестве исходных данных синтаксическое дерево выражения, записанного в расширенном лямбда-исчислении, произведет необходимые преобразования для того, чтобы привести это выражение к слабой заголовочной нормальной форме. Конечно, итоговая СЗНФ также будет представлена своим синтаксическим деревом. Писать такую программу мы будем на языке *Haskell*, и наша основная цель будет состоять в написании функции `interpret`, которая в качестве аргумента принимает объект типа `Expr`, и в качестве результата также выдает объект типа `Expr`, но уже представляющий выражение в СЗНФ, эквивалентное исходному. Итак, можно определить тип функции `interpret`:

```
interpret :: Expr -> Expr
```

Можно предложить различные способы определения такого интерпретатора, мы воспроизведем логику работу одного из самых старых (и достаточно простых) интерпретаторов такого рода – *eval / apply* интерпретатора, основанного на работе с контекстом и восходящего к первому интерпретатору языка *Lisp*, созданному Джоном МакКарти. Этот интерпретатор использует понятие *контекста* – списка, в котором идентификаторы переменных связаны со своими значениями – выражениями, находящимися в СЗНФ. Такого рода списки принято называть *ассоциативными*, так как они содержат *ассоциации* – пары, в которых одно значение (в данном случае – идентификатор переменной) связано с другим (в данном случае – выражением в СЗНФ). Определим соответствующий тип данных

```
type Context = [(String, Expr)]
```

Eval / apply интерпретатор МакКарти содержит две основные функции – функцию `eval`, вычисляющую значение выражения в заданном контексте переменных, и функцию `apply`, которая вычисляет результат применения некоторой функции к заданному значению аргумента. Кроме этих основных функций интерпретатор содержит несколько вспомогательных функций, в частности, функции для работы с ассоциативным списком, функции для определения результата применения стандартных функций и т.п. Типы основных функций `eval` и `apply` можно определить следующим образом:

```
eval  :: Context -> Expr -> Expr
apply :: Expr -> Expr -> Expr
```

Наиболее сложной из этих двух функций является функция `eval`, которая и выполняет всю основную работу. Вся функция интерпретации `interpret` выражается одним вызовом этой функции:

```
interpret = eval []
```

то есть интерпретация выражения – это его вычисление с помощью функции `eval` в пустом контексте. Прежде, чем рассматривать определение и работу основных функций интерпретатора, давайте рассмотрим несколько функций для работы с контекстом.

Контекст, как и любой ассоциативный список, предназначен для того, чтобы заносить в него ассоциативные пары и производить поиск в нем значения, связанного с заданным ключом. В данном случае ключом для поиска является идентификатор переменной, и основной функцией работы с контекстом будет функция ассоциативного поиска `assoc`. Две другие функции будут добавлять в контекст новую пару и объединять два контекста, добавляя в контекст сразу целый список пар. Эти две функции будут представлены в нашей программе операциями <+> и <++>. На самом деле первая из этих операций – <+> – это просто присоединение элемента к списку, конструктор `(:)`. Вторая операция – <++> – это просто соединение списков. Смысл введения новых идентификаторов для уже имеющихся операций состоит только в том, что новые операции работают со списками вполне определенного типа. Ниже приведены все определения для операций `assoc`, <+>, и <++>.

```
assoc :: String -> Context -> Expr
infixr 6 <+>
infixr 5 <++>
(<+>)  :: (String, Expr) -> Context -> Context
(<++>) :: Context -> Context -> Context

assoc x ((y, e):ctx) | x == y    = e
                     | otherwise = assoc x ctx
(<+>)  = (:)
(<++>) = (++)
```

Заметим еще, что функция `assoc` предназначена только для успешного поиска. Если искомого ключа в ассоциативном списке не окажется, то работа функции завершится аварийно. Мы, впрочем, всегда будем искать связанное с переменной значение в ситуации, когда в контексте эта переменная уже определена. Это соответствует тому факту, что в наших программах не будут использоваться переменные, не связанные никаким лямбда-выражением. Это, конечно, одно из условий правильности исходной программы.

Теперь попробуем написать уравнения для основной функции интерпретатора – `eval`. Прежде всего, обратим внимание на то, что некоторые выражения уже находятся в СЗНФ, так что результатом их

вычисления будет то же самое выражение. Так, в СЗНФ находятся все константы и лямбда-выражение. Тем не менее, лямбда-выражения и стандартные функции будут обрабатываться нами особо. Дело в том, что для того, чтобы правильно применить функцию, заданную лямбда-выражением, требуется, чтобы функциональное значение содержало бы в себе контекст глобальных переменных. Поскольку наши вычисления основаны на контексте переменных, то в тот момент, когда создается функциональное значение, мы будем запоминать текущий контекст переменных. Поэтому мы введем еще один тип выражения, функционально эквивалентный обычному лямбда-выражению, однако содержащий помимо связанной переменной и тела функции еще и контекст. Такое выражение будем называть *замыканием* лямбда-выражения, и оно и будет в нашей программе представлять СЗНФ для лямбда-выражения. Определение типа данных `Expr` придется расширить новым конструктором, который мы назовем `Closure`.

Аналогично, введем специальный тип выражения, представляющий СЗНФ для стандартных функций. Здесь уже не надо хранить в значении глобальный контекст, поскольку все стандартные функции выполняются одинаково независимо от контекста. Однако применение стандартной функции может быть частичным (в языке *Haskell* такая конструкция называется сечением), поэтому имеет смысл хранить вместе с некоторым представлением стандартной функции уже вычисленные значения аргументов, а также количество аргументов, которые еще требуется вычислить для того, чтобы функцию можно было бы применить. Для этого введем новый конструктор `Oper`, аргументами которого будут целое число – количество аргументов, которые еще необходимо вычислить для того, чтобы функцию можно было применить, идентификатор стандартной функции, а также список уже вычисленных значений аргументов. Итак, определение типа данных `Expr` принимает свой окончательный вид.

```
data Expr =
     Integral Integer |  -- целые константы
     Logical Bool |      -- логические константы
     Function String |   -- идентификаторы примитивных функций
     Variable String  |  -- переменная
     Lambda   String Expr  |         -- лямбда-выражение
     Application Expr Expr |         -- применение функции
     Let [(String, Expr)] Expr |     -- простой блок
     Letrec [(String, Expr)] Expr |  -- рекурсивный блок
     Closure  String Expr Context |  -- замыкание
     Oper     Int String [Expr]      -- сечение
```

Теперь мы уже готовы к тому, чтобы написать несколько простых уравнений для функции `eval`. Это будут уравнения для тех случаев, когда аргументом этой функции будет выражение, уже находящееся в СЗНФ или

выражения, СЗНФ для которых мы только что обсудили – лямбда-выражение и стандартная функция.

```
eval _   e@(Integral _)  = e
eval _   e@(Logical _)   = e
eval _   (Function f)    = Oper (arity f) f []
eval ctx (Lambda x e)    = Closure x e ctx
eval _ e@(Closure _ _ _) = e
eval _ e@(Oper _ _ _)    = e
```

На самом деле есть и еще одно очень простое уравнение. Если аргументом функции `eval` является переменная, то для того, чтобы вычислить ее значение, требуется просто найти это значение в текущем контексте.

```
eval ctx (Variable x)    = assoc x ctx
```

Остались самые сложные для реализации конструкции – блоки и применение функции. Надо сказать, что именно в этих конструкциях и содержится практически вся семантика программ. В главе 2 мы ввели понятие ленивых вычислений, при которых аргумент передается в функцию без вычисления его значения, вычисление происходит только в момент первого обращения к нему. В противоположность этому более простая схема энергичных вычислений предполагает, что перед началом работы функции ее аргументы вычисляются, а в функцию передаются уже вычисленные значения. При изучении лямбда-исчисления мы ввели также понятия различных порядков редукций, что примерно соответствовало схемам ленивых и энергичных вычислений. Сейчас, перед тем, как реализовывать вызов функции в нашем интерпретаторе, надо решить, какой схемы вычислений мы будем придерживаться.

Вообще говоря, в языке *Haskell* принята ленивая схема вычислений, так что если мы хотим написать интерпретатор, воспроизводящий семантику этого языка, то реализовать надо ленивые вычисления. Кроме того, как мы видели, при осуществлении текстовых подстановок в АПР могут встречаться ситуации, в которых необходимо производить переименование переменных. Однако мы начнем все же с реализации энергичной схемы просто потому, что эта схема проще. Кроме того, при реализации интерпретатора, основанного на контексте, на самом деле текстовые подстановки не производятся, так что трудность с необходимостью переименования переменных просто отсутствует. Наконец, язык *LISP*, для которого впервые был предложен *eval / apply* интерпретатор, реализует именно энергичную схему вычислений.

При энергичной схеме вычислений применение функции осуществляется в следующем порядке: сначала вычисляются выражения, задающие функцию и ее аргумент, а затем вычисленное значение аргумента передается функции для обработки. В нашем интерпретаторе это будет соответствовать тому, что для вычисления значения применения

функции (Application func arg) необходимо сначала вычислить выражения func и arg, а затем передать вычисленный аргумент в функцию для вычисления результата. Специально для второго действия – применения функции – предназначена вторая из основных функций интерпретатора – apply. Тогда уравнение для случая, когда самым внешним конструктором в вычисляемом выражении является конструктор Application, будет следующее уравнение:

```
eval ctx (Application func arg) =
                          apply (eval ctx func) (eval ctx arg)
```

Теперь настало время разобраться, как будет работать функция apply. Первым аргументом этой функции служит значение, представляющее вызываемую функцию, а вторам аргументом – вычисленное значение аргумента. Функцией может оказаться выражение только двух видов – замыкание (Closure) или сечение (Oper). Уравнения для функции apply надо будет выписать отдельно для этих двух случаев. Заметим, что применение функции apply соответствует редукции в нашем лямбда-исчислении. Применение замыкания лямбда-выражения соответствует β-редукции, а применение стандартной функции в случае, когда все аргументы для этой стандартной функции уже вычислены, – δ-редукции.

В случае замыкания вычисление значения функции происходит по правилам, заданным в теле функции, только в качестве значения аргумента функции подставляется фактическое значение аргумента, переданного вторым аргументом в функцию apply. Поскольку наши вычисления происходят в заданном контексте переменных (этот контекст хранится внутри замыкания в виде объекта типа Context), то надо извлечь этот контекст из замыкания, пополнить его новой парой, в которой формальный аргумент функции связывается с фактическим значением аргумента и вычислить тело функции в новом получившемся контексте. Получается следующее уравнение:

```
apply (Closure x body ctx) arg =
               eval newCtx body where newCtx = (x, arg) <+> ctx
```

В этом уравнении видно, что мы образуем новый контекст newCtx приписыванием к уже существующему контексту ctx новой пары (x, arg). Затем тело функции body вычисляется в этом новом контексте. Такое действие вполне соответствует определению β-редукции, так как при применении β-редукции происходит подстановка аргумента вместо формальной переменной лямбда-выражения (это соответствует образованию новой пары в контексте переменных) и вычисление тела лямбда-выражения.

Для случая, когда первым аргументом функции apply является значение, образованное конструктором Oper, алгоритм вычисления будет

различным в зависимости от того, является ли аргумент применения, переданный функции `apply`, последним аргументом стандартной функции. Если это еще не последний аргумент, то его значение просто присоединяется к списку уже вычисленных значений аргументов, а количество ожидаемых аргументов для завершения вычислений уменьшается на единицу. Если же аргумент - последний, то следует применить стандартную функцию (выполнить δ-редукцию), для чего потребуется еще одна вспомогательная функция интерпретатора `intrinsic`, осуществляющая работу всех стандартных алгоритмов. Итак, второе уравнение для функции `apply` будет выглядеть так:

```
apply (Oper nArgs func listArgs) arg
        | nArgs == 1  =  intrinsic func newListArgs
        | otherwise   =  Oper (nArgs-1) func newListArgs
              where newListArgs = listArgs ++ [arg]
```

Теперь мы уже можем исполнять любые программы, не содержащие блоков. Что касается нерекурсивного блока `let`, то его реализация сравнительно несложна. По существу, такой блок ничем не отличается от вызова безымянной функции с несколькими аргументами, причем тело блока является телом этой функции, формальные аргументы – это переменные, определяемые в заголовке блока, а фактическими аргументами вызова будут выражения, находящиеся в правых частях связывающих уравнений в заголовке блока. Собственно, можно было бы перевести нерекурсивный блок на язык лямбда-выражений, однако, проще и естественнее будет записать уравнения для функции `eval` в этом случае непосредственно. Дело еще и в том, что в данном случае нам совершенно не нужна сложная работа с контекстами, которая была проделана, когда мы реализовывали в нашем интерпретаторе лямбда-выражения. Выражения в заголовке блока вычисляются в исходном контексте, а для вычисления тела блока контекст просто пополняется новыми связями.

Уравнение для случая нерекурсивного блока получается довольно длинным, но по существу, ничего сложного в нем нет. Прежде всего, для того, чтобы получить список новых связей переменных с их вычисленными значениями к списку пар (переменная, выражение) применяется поэлементно функция, которая, оставляя первый элемент пары без изменения, производит вычисление второго элемента пары – выражения – в заданном контексте с помощью функции `eval`. Эта вспомогательная функция задана в уравнении с помощью лямбда-выражения `(\(x, e) -> (x, eval ctx e))`. Потом получившийся список пар соединяется с контекстом `ctx`, и тело блока вычисляется в новом, пополненном контексте `newCtx`.

```
eval ctx (Let pairs body) = eval newCtx body where
    newCtx = (map (\(x, e) -> (x, eval ctx e)) pairs) <++> ctx
```

Почти таким же образом можно реализовать и рекурсивный блок. Разница состоит в том, что в рекурсивном блоке все выражения, содержащиеся в правых частях равенств в заголовке блока, вычисляются не в старом контексте `ctx`, а в новом уже пополненном контексте `newCtx`. Конечно, может показаться странным, что мы используем новый контекст в конструкции, где этот новый контекст только строится, однако, не забывайте, что *Haskell* – это ленивый язык, поэтому уравнение для реализации рекурсивного блока `Letrec` выглядит практически так же, как и для нерекурсивного блока.

```
eval ctx (Letrec pairs body) = eval newCtx body where
 newCtx = (map (\(x, e) -> (x, eval newCtx e)) pairs) <++> ctx
```

Теперь мы уже готовы исполнять практически любые программы с помощью построенного интерпретатора. Осталось не так уж много проблем. Во-первых, у нас не реализованы никакие встроенные функции, а без них написать сколько-нибудь полезную программу не удастся (если только не промоделировать все стандартные значения и функции так, как это мы проделывали в определении чистого лямбда-исчисления). Это, впрочем, проблема небольшая. Для примера возьмем следующий набор "стандартных" функций:

`ADD` - функция сложения целых;

`SUB` - функция вычитания целых;

`MULT` - функция умножения целых;

`DIV` - функция деления целых с отбрасыванием отстатка;

`EQ0` - функция сравнения целого числа с нулем;

`SUCC` - функция увеличения целого числа на единицу;

`PRED` - функция уменьшения целого числа на единицу

Этот набор элементарных функций уже позволит нам писать простые программы для целочисленных вычислений, таких, как вычисление факториала числа или проверка простоты заданного натурального числа. Для того, чтобы можно было в программе использовать эти функции, для каждой из них надо задать число требующихся для ее выполнения аргументов («арность» функции), задаваемую с помощью функции интерпретатора `arity` и алгоритм выполнения операции, задаваемый в интерпретаторе с помощью функции `intrinsic`. Соответствующие уравнения могут выглядеть следующим образом:

```
arity "ADD" = 2
arity "SUB" = 2
arity "MULT" = 2
arity "DIV" = 2
arity "EQ0" = 1
arity "SUCC" = 1
arity "PRED" = 1
```

```
intrinsic "ADD" [Integral(a), Integral(b)] = Integral (a+b)
intrinsic "SUB" [Integral(a), Integral(b)] = Integral (a-b)
intrinsic "MULT" [Integral(a), Integral(b)] = Integral (a*b)
intrinsic "DIV" [Integral(a), Integral(b)] =
                                        Integral (a `div` b)
intrinsic "EQ0" [Integral(a)] = Logical (a==0)
intrinsic "SUCC" [Integral(a)] = Integral (a+1)
intrinsic "PRED" [Integral(a)] = Integral (a-1)
```

Вторая проблема состоит в организации условных вычислений. Дело в том, что реализовать условные вычисления так, как это предполагалось раньше – с помощью реализации встроенной функции `IF` – не удастся. Проблема состоит в том, что мы реализовали энергичную схему вычислений, а функция `IF` требует, чтобы по крайней мере два ее последних аргумента не вычислялись прежде, чем будет получено значение первого аргумента – условия. Проще всего считать конструкцию для условного вычисления одним из специальных видов выражений и реализовать эту конструкцию отдельно. Добавим к типу данных `Expr` еще один конструктор для представления условного выражения:

```
data Expr = ...
        If Expr Expr Expr |  -- условное выражение
           ...
```

и допишем еще одно уравнения для функции `eval`:

```
eval ctx (If cond t e) =
 eval ctx (if (eval ctx cond) == (Logical True) then t else e)
```

В этом уравнении явно написано, что функция `eval` вычисляется два раза. Сначала она вычисляет значение условия `cond` в заданном контексте, а затем, в зависимости от того, какой результат получился при вычислении, вычисляется одно из двух выражений `t` или `e`. В этом уравнении предполагается, что объекты типа `Expr` можно сравнивать между собой с помощью операции сравнения на равенство `==`. Этого можно достичь, если включить тип данных `Expr` в класс `Eq` явно или неявно с помощью конструкции языка `deriving`.

Теперь наш интерпретатор написан полностью. Он получился довольно компактным, так что его можно привести целиком в листинге 3.1.

Листинг 3.1. Энергичный eval / apply интерпретатор

```
-- Основной тип данных для представления выражений
-- расширенного лямбда-исчисления
data Expr =
     Integral Integer |  -- целые константы
     Logical Bool |      -- логические константы
     Function String |   -- идентификаторы примитивных функций
     Variable String  |  -- переменная
```

```
      Lambda   String Expr  |          -- лямбда-выражение
      If Expr Expr Expr |             -- условное выражение
      Application Expr Expr |         -- применение функции
      Let [(String, Expr)] Expr |     -- простой блок
      Letrec [(String, Expr)] Expr |  -- рекурсивный блок
      Closure  String Expr Context |  -- замыкание
      Oper     Int String [Expr]      -- сечение
            deriving (Show, Eq)

-- Контекст вычислений представлен ассоциативным списком
type Context = [(String, Expr)]

-- Определения типов основных и вспомогательных функций
---- Интерпретатор:
interpret  :: Expr -> Expr

---- Основные функции eval / apply
eval       :: Context -> Expr -> Expr
apply      :: Expr -> Expr -> Expr

---- Функции для применения стандартных операций
infixr 6 <+>
infixr 5 <++>
(<+>)      :: (String, Expr) -> Context -> Context
(<++>)     :: Context -> Context -> Context
assoc      :: String -> Context -> Expr
arity      :: String -> Int
intrinsic  :: String -> [Expr] -> Expr

-- Определения уравнений для вспомогательных функций
---- Поиск по контексту:
assoc x ((y, e):ctx) | x == y    = e
                     | otherwise = assoc x ctx
---- Добавление в контекст
(<+>)  = (:)
(<++>) = (++)

---- Определение числа операндов стандартных операций
arity "ADD" = 2
arity "SUB" = 2
arity "MULT" = 2
arity "DIV" = 2
arity "EQ0" = 1
arity "SUCC" = 1
arity "PRED" = 1

---- Выполнение стандартных операций над списком аргументов
intrinsic "ADD" [Integral(a), Integral(b)] = Integral (a+b)
intrinsic "SUB" [Integral(a), Integral(b)] = Integral (a-b)
intrinsic "MULT" [Integral(a), Integral(b)] = Integral (a*b)
```

```
intrinsic "DIV" [Integral(a), Integral(b)] =
                                        Integral (a `div` b)
intrinsic "EQ0" [Integral(a)] = Logical (a==0)
intrinsic "SUCC" [Integral(a)] = Integral (a+1)
intrinsic "PRED" [Integral(a)] = Integral (a-1)

-- Определения уравнений для основных функций
---- Интерпретатор:
interpret = eval []

---- Eval - вычисление значения выражения в контексте
(приведение к СЗНФ)
eval _   e@(Integral _)  = e
eval _   e@(Logical _)   = e
eval _   (Function f)    = Oper (arity f) f []
eval ctx (Lambda x e)    = Closure x e ctx
eval _ e@(Closure _ _ _) = e
eval _ e@(Oper _ _ _)    = e
eval ctx (Variable x)    = assoc x ctx
eval ctx (Application func arg) =
                         apply (eval ctx func) (eval ctx arg)
eval ctx (If cond t e) =
 eval ctx (if (eval ctx cond) == (Logical True) then t else e)
eval ctx (Let pairs body) = eval newCtx body where
 newCtx = (map (\(x, e) -> (x, eval ctx e)) pairs) <++> ctx
eval ctx (Letrec pairs body) = eval newCtx body where
 newCtx = (map (\(x, e) -> (x, eval newCtx e)) pairs) <++> ctx

---- Apply - вычисление результата применения функции к
аргументу
apply (Closure x body ctx) arg =
              eval newCtx body where newCtx = (x, arg) <+> ctx
apply (Oper nArgs func listArgs) arg
      | nArgs == 1 = intrinsic func newListArgs
      | otherwise  = Oper (nArgs-1) func newListArgs
          where newListArgs = listArgs ++ [arg]
```

Интерпретатор полностью работоспособен. Давайте с его помощью вычислим значение факториала числа 6. Для этого мы определим программу, в которой описана функция `fact` для вычисления факториала, и затем эта функция применяется к числу 6. Сначала мы напишем нашу программу на языке расширенного лямбда-исчисления. Затем, записав соответствующее выражение в виде значения типа `Expr`, мы сможем применить к нему функцию `interpret` и получить искомое значение.

Итак, программа на языке расширенного лямбда-исчисления будет выглядеть так:

```
letrec fact =
       λn.IF (EQ0 n) 1 (MULT n (fact (PRED n))) in fact 6
```

Соответствующий объект типа `Expr` будет записан с помощью более громоздкой конструкции, хотя, по существу, она является простым переписыванием только что приведенного выражения.

```
Letrec
   [("fact",
     (Lambda
         "n"
         (If
            (Application
               (Function "EQ0")
               (Variable "n"))
            (Integral 1)
            (Application
               (Application
                  (Function "MULT")
                  (Variable "n"))
               (Application
                  (Variable "fact")
                  (Application
                     (Function "PRED")
                     (Variable "n"))))
         )))]
   (Application (Variable "fact") (Integral 6))
```

При применении к только что выписанному выражению функции `interpret` будет получен ожидаемый результат `Integral 720`.

Написание интерпретатора для функционального языка само по себе довольно интересно, однако, надо сказать, что оно помогло нам еще и уточнить в виде программного кода ряд деталей, которые мы раньше могли описать только словесно. Например, семантика выполнения условного выражения определена в интерпретаторе очень четко: для его исполнения требуется вычислить значения двух подвыражений – условия и одной из двух альтернатив. Выбор альтернативы определяется результатом вычисления условия. Однако некоторые детали семантики все же остались определенными недостаточно точно. Например, семантику условного выражения можно было бы выразить и по-другому, записав соответствующее уравнение для функции `eval` в виде

```
eval ctx (If cond t e) =
    if (eval ctx cond) == (Logical True)
        then (eval ctx t) else (eval ctx e)
```

Эта запись работает точно так же, как и предыдущая, однако для того, чтобы понять, что только одно из двух подвыражений `t` и `e` вычисляется, надо понимать, что именно так работает конструкция `if` в языке *Haskell*. Получается замкнутый круг: семантика условного выражения объясняется с помощью ссылок на семантику условного выражения! Еще хуже обстоит дело с семантикой энергичного

вычисления. На протяжении всего раздела мы утверждали, что нами реализована схема энергичных вычислений. Однако если принять во внимание, что язык *Haskell* работает в соответствии с ленивой схемой, то окажется, что фактически вычисление значений аргумента будет отложено до момента первого обращения к ним, то есть на самом деле вычисления происходят по ленивой схеме! Опять оказалось, что семантику вычислений невозможно понять, если не знать заранее, по какой схеме и как выполняется программа.

Возникает несколько вопросов: что надо сделать с программой, чтобы она и в самом деле реализовывала семантику энергичных вычислений? как ясно выразить тот факт, что семантика вычислений все-таки ленивая? можно ли использовать какой-нибудь «энергичный» язык функционального программирования (скажем, *LISP*) для реализации ленивой семантики? На некоторые из этих вопросов можно дать сравнительно простой ответ, на другие удовлетворительного ответа найти не удается.

Наверное, самым простым вопросом является первый из приведенных: как исправить программу, чтобы семантика вычислений и вправду была энергичной. Дело в том, что хотя язык *Haskell* и является, вообще говоря, «ленивым», его все-таки можно заставить выполнить передачу аргументов, характерную для энергичной схемы вычислений. Для этого используется встроенная операция `$!` – операция «энергичного» применения функции. Так, если `f` – это некоторая функция, а `a` – ее аргумент, представленный некоторым выражением, то обычный вызов функции, записанный в виде `(f a)` будет выполняться по ленивой схеме вычислений. Чтобы применить функцию к аргументу по энергичной схеме, то есть заставить исполнительную систему вычислить аргумент до его передачи в функцию, надо записать вызов в виде применения упомянутой операции: `(f $! a)`. Заметим, кстати, что в языке есть и еще одна операция применения функции ствндартным для *Haskell* ленивым способом – операция «применить». Она реализована в виде бинарной операции `$`, и используется обычно для образования сечений.

В нашем интерпретаторе операция энергичного вызова может быть использована в ситуации, где выражения-аргументы передаются в функцию, то есть при вызове функции интерпретатора `apply`. Это потребует изменения единственного уравнения в определении функции `eval`.

```
eval ctx (Application func arg) =
                  (apply $! (eval ctx func)) $! (eval ctx arg)
```

Чтобы проверить, что мы действительно имеем два интерпретатора с разным поведением, попробуем применить функцию `interpret` к выражению, которое должно выдавать разный результат в зависимости от применяемой схемы вычислений. Точнее, попробуем определить

выражение, которое выдает определенный результат при ленивой схеме вычислений, и приводит к ошибке при энергичной схеме вычислений. Наверное, самым простым примером такого рода выражения будет применение функции, отбрасывающей свой аргумент, к ошибочному аргументу: `(λx.1)(DIV 1 0)`. При ленивой схеме вычислений выражение должно приводиться к значению `1`, в то время как при энергичной схеме вычислений программа должна быть прервана из-за невозможности вычисления результата деления на ноль `(DIV 1 0)`. Перепишем нашу исходную программу в виде

```
test = Application
  (Lambda "x" (Integral 1))
  (Application
     (Application (Function "DIV") (Integral 1)) (Integral 0))
```

и попробуем применить к этому выражению функцию интерпретатора `interpret test`. Как и следовало ожидать, первая версия нашего «энергичного» интерпретатора выдаст результат `(Integral 1)`, что подтверждает нашу мысль о том, что на самом деле нами был определен все же, скорее, «ленивый» интерпретатор. Однако, вторая – «исправленная» версия интерпретатора выдаст тот же самый результат! Причину этого можно обнаружить, если рассмотреть разницу между выражениями `(1 `div` 0)` и `(Integral (1 `div` 0))`. Во втором случае применение конструктора, как и все вызовы в языке *Haskell*, будет ленивым, так что деление на ноль будет происходить только тогда, когда аргумент конструктора будет нужно вывести, сравнить с образцом или использовать в качестве операнда встроенной примитивной функции. Для того, чтобы аргументы конструктора вычислялись бы «в принудительном порядке», то есть согласно энергичной схеме вычислений, требуется при описании конструктора пометить соответствующие аргументы символом `!`. Например, для того, чтобы аргумент конструктора `Integral` всегда вычислялся в момент применения конструктора, в описании типа данных `Expr` требуется описать этот конструктор следующим образом:

```
data Expr = Integral !Integer |
            ...
```

Вот теперь, когда это изменение также будет внесено в нашу программу, интерпретатор при попытке вычисления выражения `test` выдаст сообщение об ошибке при попытке деления на ноль.

Из этого обсуждения видно, насколько непросто оказалось выразить семантику энергичного и ленивого вычисления с помощью описанного нами интерпретатора. К аналогичным проблемам приводит попытка написать «ленивый» интерпретатор в «энергичном» языке функционального программирования, подобном языку *LISP*. Строго говоря, полностью выразить семантику ленивых вычислений, используя только чисто функциональные средства энергичного языка, вообще не

удается. В следующем разделе мы рассмотрим еще один способ описания семантики языка, который дает гораздо более выразительные и очевидные средства для описания поведения программ. При этом нам уже не понадобится применять трюки с использованием энергичных вызовов в ленивом языке.

Примеры решения задач

Во всех примерах и заданиях для самостоятельного решения в этом разделе используется представление выражений расширенного лямбда-исчисления, похожее на то, что использовалось при описании *eval / apply* интерпретатора.

```
data Expr =
     Integral Integer |  -- целые константы
     Function String  |  -- идентификаторы примитивных функций
     Variable String  |  -- переменная
     Lambda   String Expr  |         -- лямбда-выражение
     Application Expr Expr           -- применение функции
```

По сравнению с описанием выражения в *eval / apply* интерпретаторе это представление несколько упрощено, в частности, в нем нет логических значений и блоков.

Задание 1. Напишите функцию, выдающую для заданного выражения список всех встречающихся в нем свободных переменных.

Решение. Аргументом проектируемой функции должно быть значение типа `Expr`, а результатом – список строк. Свободные переменные – это переменные, появляющиеся в конструкции `Variable v`, которые не находятся в области действия лямюда-выражения `Lambda v e`. Собственно, задачу решает очень простая функция:

```
free :: Expr -> [String]
free (Variable v) = [v]
free (Lambda v e) = (free e) <-> v
free (Application e1 e2) = (free e1) ++ (free e2)
free _ = []
```

где операция `<->` удаляет строку из списка строк. К сожалению, эта простая функция не препятствует созданию повторяющихся элементов в списке строк. На самом деле, здесь вместо списков лучше было бы использовать множества без повторяющихся элементов. Можно воспользоваться стандартной библиотекой языка и операциями над множествами, определенными в ней. Однако, давайте вместо этого реализуем необходимые нам действия непосредственно в самой программе. В качестве представления множества выберем упорядоченный список строк без повторяющихся элементов, а поиск в нем для удаления элемента будем производить с помощью алгоритма двоичного поиска. В случае, если список строк очень велик, это не будет самым эффективным

способом представления, однако, он вполне приемлем для случая, когда число строк в списке не превосходит нескольких десятков.

Функцию `merge` слияния двух упорядоченных списков в один мы уже определяли ранее в листинге У2.6 в примерах решения задач к главе 2.2. Теперь определим операцию удаления элемента из упорядоченного списка с двоичным поиском.

```
(<->) :: Ord a => [a] -> a -> [a]
[] <-> _ = []
s <-> e | e < m = (left <-> e) ++ (m:right)
        | e > m = left ++ (m:(right <-> e))
        | e == m = left ++ right
            where i = (length s) `div` 2
                  m = s !! i
                  left = take i s
                  right = drop (i+1) s
```

Эта функция действительно реализует алгоритм двоичного поиска, но поскольку поиск осуществляется не в массиве, а в списке, то ее эффективность на самом деле зависит от того, насколько быстро работают встроенные функции `take`, `drop` и операция взятия элемента по номеру `(!!)`. Если есть сомнения в эффективности работы этих операций, то лучше вместо двоичного поиска использовать обычный линейный поиск по списку, и тогда функция удаления элемента будет определена следующим образом:

```
(<->) :: Ord a => [a] -> a -> [a]
[] <-> _ = []
s@(first:tail) <-> e | e < first = s
                     | e > first = first:(tail <-> e)
                     | e == first = tail
```

Полностью программа приведена ниже в листинге У4.1.

Листинг У4.1. Нахождение списка свободных переменных.

```
-- Тип данных для представления выражений
data Expr =
     Integral Integer |  -- целые константы
     Function String  |  -- идентификаторы примитивных функций
     Variable String  |  -- переменная
     Lambda   String Expr  |         -- лямбда-выражение
     Application Expr Expr           -- применение функции

-- Функция слияния двух упорядоченных списков
-- (без повторений элементов).
-- Аргументы - два упорядоченных списка.
-- Результат - список, объединяющий элементы исходных списков
--             без повторений.
merge :: Ord a => [a] -> [a] -> [a]
merge [] l = l
```

```
merge l [] = l
merge lst1@(x1:l1) lst2@(x2:l2)
                   | x1 < x2   = x1 : merge l1 lst2
                   | x1 == x2  = x1 : merge l1 l2
                   | otherwise = x2 : merge lst1 l2

-- Функция удаления элемента из упорядоченного списка.
-- Список просматривается способом линейного поиска.
-- Аргументы: список и удаляемый элемент.
-- Результат: список без удаленного элемента
(<->) :: Ord a => [a] -> a -> [a]
[] <-> _ = []
s@(first:tail) <-> e | e < first = s
                     | e > first = first:(tail <-> e)
                     | e == first = tail

-- Основная функция для решения задачи.
-- Аргумент: представление выражения лямбда-исчисления.
-- Результат: упорядоченный список свободных переменных.
free :: Expr -> [String]
free (Variable v) = [v]
free (Lambda v e) = (free e) <-> v
free (Application e1 e2) = merge (free e1) (free e2)
free _ = []
```

Тестовые примеры для этой программы напишите самостоятельно.

Задание 2. Напишите функцию, которая проверяет, есть ли в заданном выражении β-редекс.

Решение. Для выражений заданного вида β-редекс – это подвыражение вида `(Application (Lambda x e1) e2)`, так что задача сводится к определению того, если ли в заданном выражении подвыражение такого вида. В языке *Haskell* такие задачи решаются очень просто с помощью конструкции сопоставления с образцом.

```
beta :: Expr -> Bool
beta (Application (Lambda _ _) _) = True
beta (Lambda _ e) = beta e
beta (Application e1 e2) = beta e1 || beta e2
beta _ = False
```

Задание 3. Напишите функцию, которая определяет, находится ли заданное выражение в нормальной форме.

Решение. Выражение находится в нормальной форме, если в нем нет δ- и β-редексов. Определить, есть ли в выражении β-редекс мы можем очень легко, это мы сделали в предыдущем задании. Определение δ-редекса, к сожалению, не сводится к простой проверке формы выражения, так как дополнительно необходимо проверять, имеется ли для встроенной функции достаточное число аргументов. Это число, конечно, зависит от самой вызываемой функции. Кроме того, надо проверять, что аргументы встроенной функции являются константами.

Чтобы решить задачу в достаточно общем виде, надо иметь функцию, которая определяет «арность» каждой встроенной функции. Давайте считать, что такая функция у нас есть:

```
arity :: String -> Int
```

Тогда можно написать функцию, которая проверяет, является ли заданное выражение δ-редексом для функции, имеющей *n* аргументов:

```
deltaN :: Int -> Expr -> Bool
deltaN n (Function f)        = arity f == n
deltaN n (Application f e) | isConstant e
                             = deltaN (n+1) f && nf e
deltaN _ _                   = False
```

Здесь идентификатором `isConstant` обозначена функция проверки того, является ли выражение константой. Конечно, константой является целое число, то есть выражение вида `(Integral _)`, но константами также следует считать и идентификаторы встроенных функций, то есть выражения вида `(Function _)`.

Теперь можно немного модифицировать функцию `beta`, которая теперь будет просто проверять, является ли выражение β-редексом и написать функцию `delta`, проверяющую, является ли выражение δ-редексом. Полностью решение приведено в листинге У4.2. В нем функция `arity` определена для некоторых встроенных операций, причем считается, что если арность функции не задана, то она равна двум.

Листинг У4.2. Определение, находится ли выражение в нормальной форме.

```
-- Тип данных для представления выражений.
data Expr =
     Integral Integer |  -- целые константы
     Function String  |  -- идентификаторы примитивных функций
     Variable String  |  -- переменная
     Lambda   String Expr  |         -- лямбда-выражение
     Application Expr Expr           -- применение функции

-- Функция проверки, является ли выражение beta-редексом.
-- Аргумент: исходное выражение.
-- Результат: True, если аргумент является beta-редексом,
--            False в противном случае.
beta :: Expr -> Bool
beta (Application (Lambda _ _) _) = True
beta _ = False

-- Функция проверки, является ли выражение delta-редексом для
-- встроенной функции заданного числа аргументов.
-- Аргументы: число аргументов встроенной функции;
--            проверяемое выражение.
-- Результат: True, если аргумент является delta-редексом,
--            False в противном случае.
```

```
deltaN :: Int -> Expr -> Bool
deltaN n (Function f)      = arity f == n
deltaN n (Application f e) | isConstant e
                             = deltaN (n+1) f && nf e
deltaN _ _                 = False

-- Функция проверки, является ли выражение delta-редексом для
-- некоторой встроенной функции.
-- Аргумент: проверяемое выражение.
-- Результат: True, если аргумент является delta-редексом,
--            False в противном случае.
delta :: Expr -> Bool
delta = deltaN 0

-- Функция проверки, является ли выражение beta-
-- или delta-редексом.
-- Аргумент: проверяемое выражение.
-- Результат: True, если аргумент является редексом,
--            False в противном случае.
redex :: Expr -> Bool
redex e = delta e || beta e

-- Функция определения арности встроенной функции.
-- Определение этой функции дано просто для примера,
-- набор встроенных функций далеко не полон.
-- Аргумент: имя встроенной функции.
-- Результат: арность функции. Полагаем арность равной двум
--            для всех "неизвестных" функций.
arity :: String -> Int
arity "IF" = 3
arity "CAR" = 1
arity "CDR" = 1
arity "minus" = 1
arity _ = 2

-- Функция проверки того, является ли выражение констрантой.
-- Аргумент: исходное выражение.
-- Результат: True, если выражение является константой,
--            False в противном случае.
isConstant :: Expr -> Bool
isConstant (Integral _) = True
isConstant (Function _) = True
isConstant _            = False

-- Основная функция определения того, находится ли выражение
-- в нормальной форме.
-- Аргумент: исходное выражение.
-- Результат: True, если аргумент находится в нормальной
--            форме, False в противном случае.
nf :: Expr -> Bool
```

```
nf e | redex e          = False
nf (Lambda _ e)         = nf e
nf (Application e1 e2) = nf e1 && nf e2
nf _                    = True
```

Заметим, что приведенное в листинге решение довольно просто, но неэффективно: одни и те же подвыражения просматриваются в нем несколько раз. Конечно, можно написать и более эффективно работающую программу, такое упражнение предоставляется читателям для самостоятельной работы.

Еще одно замечание: программа хорошо работает только для «правильных» выражений, в которых нет свободных переменных, а все функции применяются к выражениям «правильного» типа. Если выражению не удается придать какой-либо смысл, то и определение того, находится ли оно в нормальной форме, может выдавать ошибочный результат. Так, например, для выражения

```
Lambda "x" (Application (Function "minus") (Function "+"))
```

программа выдаст результат False (выражение не находится в нормальной форме), однако интерпретировать такой результат довольно затруднительно – как применить функцию “minus” к функции “+” непонятно.

Задания для самостоятельной работы

Задание 1. Напишите функцию, которая по заданному выражению выдает список имен всех встроенных (константных) функций, встречающихся в нем.

Задание 2. Напишите функцию, которая по имени и телу некоторой рекурсивной функции строит с помощью Y-комбинатора эквивалентную ей нерекурсивную функцию.

Задание 3. Напишите функцию, которая проверяет, есть ли в заданном выражении идентификаторы, представляющие как свободные переменные, так и связанные в некотором его подвыражении переменные. Например, в выражении `(λx.y (λy.y))` идентификатор `y` представляет как свободную переменную (первое вхождение идентификатора в выражении), так и связанную (второе вхождение).

4.3. Компиляция в SECD-машину

Семантику исполнения программы лучше всего выражать не через исполнение таких же или очень похожих конструкций на те, семантику которых мы и пытаемся определить, а с помощью более простых конструкций, исполнение которых настолько ясно и однозначно, что не требует никаких дополнительных пояснений. Например, если мы построим модель абстрактной машины, которая исполняет простые команды последовательно одна за одной, то можно будет считать, что семантика

работы этой простой машины нам абсолютно ясна. Тогда, если удастся перевести (скомпилировать) выражение, записанное на языке расширенного лямбда-исчисления, в последовательность команд такой машины, так что эффект вычисления выражения будет эквивалентен эффекту выполнения получившейся последовательности команд, то можно считать, что задача определения операционной семантики расширенного лямбда-исчисления (или в более общем виде – программ, написанных на функциональных языках) будет решена.

Здесь мы впервые употребили термин *операционная семантика*. Этот термин означает любую семантику, выраженную в терминах последовательного исполнения некоторых простых команд (операций). Например, когда мы задавали смысл конструкций чистого лямбда-исчисления в терминах операций преобразования этих выражений в другие путем применения к ним β-редукций, мы задавали тем самым операционную семантику выражений чистого лямбда-исчисления. Однако заданная нами в терминах преобразования выражений операционная семантика нас удовлетворить не может. Дело в том, что отдельные операции при таком задании семантики – это операции текстового преобразования. Для человеческого восприятия операции преобразования текста привычны и удобны, однако они требуют постоянного переписывания одних и тех же значительных фрагментов текста. Хотелось бы иметь возможность выполнять операции над более удобными для представления в компьютере объектами. Кроме того, в текстовых преобразованиях так и не удалось достаточно точно представить семантику ленивых вычислений – введение блока `let` и специальных правил его обработки – это не очень изящное и не очень хорошо формализуемое решение.

Давайте рассмотрим одну из простых машин, предназначенных для исполнения функциональных программ – SECD-машину Лэндина (Peter J. Landin). Мы рассмотрим две разновидности этой машины: сначала высокоуровневую, в которой командами являются сами выражения расширенного лямбда-исчисления, а затем разновидность той же машины более низкого уровня, в которой команды – это специально созданные простые объекты. Высокоуровневая SECD-машина хороша тем, что не требует никакой трансляции выражений на язык простых команд, давая, тем не менее, возможность достаточно ясно выразить семантику преобразования выражений. Однако, высокоуровневая машина не позволяет достаточно точно и ясно выразить понятия рекурсии и рекурсивного блока. Низкоуровневая машина позволяет решить эту проблему с помощью введения специальных команд для работы с рекурсивным контекстом.

SECD-машина состоит из четырех *регистров*, в каждом из которых содержатся списки объектов характерного для каждого из регистров типа.

Эти четыре регистра - это регистр стека (**S**tack), регистр окружения или контекста (**E**nvironment), регистр управления или команд (**C**ontrol) и регистр хранения состояний (**D**ump). Четыре первых буквы названий регистров и образуют название машины – SECD. Регистр стека служит для хранения вычисленных значений выражений – выражений в слабой заголовочной нормальной форме. Регистр контекста служит для хранения выражений, связанных с переменными программы. Как правило – это тоже выражения в слабой заголовочной нормальной форме. Регистр управления содержит последовательность команд, которые необходимо исполнить. Работа машины заканчивается, когда исчерпывается список команд для исполнения. Для высокоуровневой SECD-машины этот регистр содержит выражения, значения которых необходимо выполнить; для низкоуровневой – последовательность элементарных команд. Последний, четвертый регистр – регистр хранения состояний – содержит запомненные для дальнейшего применения состояния регистров машины. При входах в функции в этом регистре запоминается текущее состояние SECD-машины для того, чтобы можно было вернуться к нему после того, как работа функции будет закончена.

Мы рассмотрим сначала высокоуровневую SECD-машину. Работу этой машины мы представим в виде функциональной программы на языке *Haskell*, моделирующей последовательное выполнение команд. К сожалению, моделирование последовательного исполнения команд в функциональном языке имеет один существенный недостаток – невозможность полного и адекватного выражения понятия разрушающего присваивания, тем не менее, по крайней мере, частично, такая модель дает хорошее представление о работе SECD-машины.

Мы по-прежнему будем считать, что основным объектом обработки для нас является выражение расширенного лямбда-исчисления, представленное объектами типа `Expr`, однако, давайте сейчас разделим все выражения на те, которые представляют собой исходные выражения, подлежащие обработке – преобразованию их в слабую заголовочную нормальную форму («вычислению»), и уже «вычисленные» выражения, находящиеся в СЗНФ. Выражения первого типа (исходные) будем называть *командами*, поскольку именно под управлением этих выражений происходит работа всей машины. Кроме исходных выражений, представляющих программы на нашем языке программирования - расширенном лямбда-исчислении, в число команд будут входить и вспомогательные объекты, нужные только для правильного функционирования SECD-машины. Тип объектов-команд обозначим идентификатором `Command`. Выражения в СЗНФ будут представлены объектами типа `WHNF` (от *Weakest Header Normal Form* – слабая заголовочная нормальная форма). Некоторые выражения в исходной форме, такие, как целые числа и логические значения, фактически уже

находятся в СЗНФ. Для таких выражений у нас будут различные конструкторы для представления их в виде объектов типа `Command` и в виде объектов типа `WHNF`.

Если считать, что типы `Command` и `WHNF` уже определены, то для описания состояния SECD-машины в целом, нам достаточно описать типы, представляющие объекты, хранящиеся в четырех регистрах SECD-машины. В стеке машины хранится последовательность (список) уже вычисленных выражений в СЗНФ, так что тип этого регистра можно описать как `[WHNF]`. Регистр контекста хранит контекст в том самом виде, как мы его использовали при описании *eval/apply*-интерпретатора. Регистр команд содержит последовательность исходных выражений-команд, так что его тип может быть выражен описателем `[Command]`. Наконец, регистр хранения состояний содержит последовательность троек, описывающих содержимое трех остальных регистров. Таким образом, получаем следующее описание SECD-машины и всех ее регистров.

```
type Stack = [WHNF]
     Environment = Context
     Control = [Command]
     Dump = [(Stack, Environment, Control)]
     SECD = (Stack, Environment, Control, Dump)
```

Работа SECD-машины выполняется по шагам, при этом каждый шаг работы определяется содержимым первой команды из регистра команд. Исполнение команды обычно приводит к удалению ее из регистра управления и модификации содержимого остальных регистров машины. Во всех случаях исполнение команды-выражения приводит к вычислению этого выражения и загрузке результата вычисления на вершину стека результатов. Вообще, конечная цель работы машины состоит в исполнении всех содержащихся в регистре управления команд и получению результата в качестве первого элемента стека. Один шаг работы SECD-машины можно записать в виде `SECD -> SECD'`, где `SECD` и `SECD'` – состояния SECD-машины до и после выполнения шага соответственно. Тогда всю работу SECD-машины можно записать в виде последовательности состояний, причем последним состоянием будет такое, в котором регистр управления пуст. Мы будем записывать состояния SECD-машины более подробно в виде `(s, e, c, d)`, где *s*, *e*, *c* и *d* описывают содержимое отдельных регистров машины. В начальный момент работы все регистры пусты, только в регистре управления содержится исходное выражение для вычисления. Такое состояние можно записать в виде `([], [], [expr], [])`, где `expr` – исходное выражение типа `Command`. Если программа нормально закончит работу, то в конечном состоянии результат вычислений будет находиться в стеке, так что состояние машины можно описать в виде `([res], [], [], [])`, где `res` – результат вычислений типа `WHNF` – выражение в СЗНФ.

Мы будем описывать работу SECD-машины в виде уравнений функции `evaluate`, которая и определяет алгоритм работы машины. Эта функция – рекурсивная с концевой рекурсией; практически все ее уравнения имеют вид

```
evaluate (s, e, c, d) = evaluate (s', e', c', d')
```

Ясно, что такая форма записи уравнений функции – это просто способ записи, в котором указано, как состояние машины `(s, e, c, d)` переходит в новое состояние – `(s', e', c', d')`. Исключение составляет единственное уравнение, описывающее результат работы машины – конечное состояние. Удобно считать, что в этом случае результатом применения функции `evaluate` является само это конечное состояние:

```
evaluate secd@(s, e, [], []) = secd
```

Интерпретация выражения состоит в том, что выражение загружается в SECD-машину в качестве первой и единственной команды в регистре управления, а затем к полученной машине применяется функция `evaluate`, которая осуществляет переходы машины из состояния в состояние до тех пор, пока не будет достигнуто конечное состояние. Результат вычислений будет находиться в стеке в этом конечном состоянии. Таким образом, функцию интерпретации с помощью SECD-машины можно записать в виде следующего описания и уравнения.

```
interpret :: Command -> WHNF
interpret com = res where (res:_, _, _, _) =
                                  evaluate ([], [], [com], [])
```

Рассмотрим теперь более подробно структуру команд и промежуточных результатов – объектов типа `WHNF`. Основная часть команд содержит просто все основные типы выражений исходной программы, соответствующие конструкторы уже были представлены нами ранее в *eval/apply*-интерпретаторе в виде конструкторов типа `Expr`. Некоторые другие команды, необходимые для функционирования SECD-машины, будем представлять по мере необходимости.

```
data Command =
    Integral Integer | Logical Bool | Function String |
                                        -- константы
    Variable String  |                  -- переменная
    Lambda   String Command  |          -- лямбда-выражение
    Application Command Command |       -- применение функции
    If Command Command Command |        -- условное выражение
    Let [(String, Command)] Command |   -- простой блок
    Letrec [(String, Command)] Command | -- рекурсивный блок
```

Результатами вычисления выражений служат также уже знакомые нам типы значений: целые и логические константы, замыкания, представляющие лямбда-выражения, и сечения, представляющие

стандартные функции с недостаточным числом переданных аргументов. Таким образом, описание типа данных WHNF будет выглядеть следующим образом:

```
data WHNF =
    Numeric Integer | Boolean Bool |       -- константы
    Closure String Command Context |       -- замыкание
    Oper Int String [WHNF]                 -- сечение
```

Здесь конструкторы Numeric и Boolean – это просто другие формы представления выражений, представленных конструкторами Integral и Logical, Closure – конструктор для представления замыкания, и Oper – конструктор для представления сечения стандартной функции вместе с уже вычисленными значениями аргументов. Теперь будем записывать уравнения для функции evaluate, определяя тем самым алгоритм работы SECD-машины.

Как и раньше, если исходными выражениями являются целые или логические константы, то результат вычисления выражения – это сами эти константы. Они переносятся на стек результатов, изменяя только форму представления. Больше ничего в состоянии SECD-машины менять не надо. Аналогично, если исходное выражение – это лямбда-выражение, то оно переносится на стек в виде замыкания (в замыкании как и раньше запоминается текущий контекст), а если исходное выражение – это стандартная функция, то она переносится на стек в виде сечения. Если в качестве команды выступает переменная, то ее значение ищется в текущем контексте. Отсюда имеем пять простых уравнений для функции evaluate.

```
evaluate (s, e, (Integral n):c, d) =
            evaluate ((Numeric n):s, e, c, d)
evaluate (s, e, (Logical b):c, d) =
            evaluate ((Boolean b):s, e, c, d)
evaluate (s, e, (Lambda x body):c, d) =
            evaluate ((Closure x body e):s, e, c, d)
evaluate (s, e, (Function f):c, d) =
            evaluate ((Oper (arity f) f []):s, e, c, d)
evaluate (s, e, (Variable x):c, d) =
            evaluate ((assoc x e):s, e, c, d)
```

Для того, чтобы вычислить условное выражение вида (If cond t e) надо прежде всего вычислить значение условия cond. Если результатом окажется константа (Boolean True), то в дальнейшем надо будет вычислять выражение t, а иначе - выражение e. Вычислить значение выражения можно, просто поместив его в регистр управления, так что выражение cond можно будет просто вытащить из конструкции (If cond the els) и поместить его в регистр управления. Для того же, чтобы осуществить выбор из двух альтернатив, введем еще одну вспомогательную команду Select, которая и будет

осуществлять выбор в зависимости от того, какое значение лежит на вершине стека результатов. Добавим новый конструктор для описания команды `Select` в описание типа данных `Command` и напишем уравнения, определяющие семантику вычисления условного выражения.

```
data Command =
        ... |
    Select Command Command |          -- выбор из альтернатив

evaluate (s, env, (If cond t e):c, d) =
             evaluate (s, env, cond:(Select t e):c, d)
evaluate ((Boolean True):s, env, (Select t e):c, d) =
             evaluate (s, env, t:c, d)
evaluate ((Boolean False):s, env, (Select t e):c, d) =
             evaluate (s, env, e:c, d)
```

Посмотрим, как будет меняться состояние SECD-машины при вычислении условного выражения, скажем

```
(If (Logical True) (Integral 2) (Integral 3))
```

Начальное состояние машины будет следующим:

```
([], [], [(If (Logical True) (Integral 2) (Integral 3))], [])
```

После первого шага условие будет вынесено в качестве первой команды, и состояние SECD-машины станет таким:

```
([],[],[(Logical True),(Select (Integral 2)(Integral 3))],[])
```

На следующем шаге значение условия будет вычислено и помещено на стек:

```
([(Boolean True)],[],[(Select (Integral 2)(Integral 3))],[])
```

Теперь команда `Select` выберет первый из своих аргументов, поскольку на стеке находится значение «истина». На следующем шаге значение этого аргумента будет вычислено и помещено на стек. На этом работа машины будет закончена.

```
([], [], [(Integral 2)], [])
([(Numeric 2)], [], [], [])
```

Мы рассмотрели самые простые команды. Теперь можно перейти к более сложным командам применения функции `Application` и блокам `Let` и `Letrec`. Для того, чтобы выполнить команду применения функции к аргументу, в энергичной схеме вычисления требуется прежде всего вычислить как саму функцию, так и ее аргумент. После этого можно выполнять собственно применение функции. Поэтому для реализации применения функции придется ввести еще одну вспомогательную команду – команду «применить», которая будет выполняться в условиях, когда функция и ее аргумент уже вычислены и находятся на вершине стека результатов. Назовем новую команду `Apply` и доопределим тип данных `Command`, чтобы ввести новый конструктор для этой команды. Тогда уравнение для команды `Application` будет выглядеть так:

```
data Command =
      ... |
    Apply |              -- команда "применить"

evaluate (s, e, (Application f a):c, d) =
                           evaluate (s, e, a:f:Apply:c, d)
```

Обратите внимание, что для вычисления значения функции и ее аргумента выражения `f` и `a` просто выкладываются в регистр управления, так что при выполнении последующих шагов вычисляются их значения, и вычисленные выражения в СЗНФ попадают на вершину стека результатов. Таким образом, перед выполнением команды `Apply` состояние SECD-машины можно описать следующим образом: `(func:arg:s, e, Apply:c, d)`, где `arg` и `func` – вычисленные значения аргумента функции и самой функции. Обратите внимание, что, поскольку сначала будет вычислено значение аргумента, а потом – значение функции, то и в стек результатов сначала попадет значение аргумента функции, а уже сверху него окажется лежащим значение самой функции. На самом деле вычисления можно было бы производить в любом порядке, однако описанный здесь порядок вычислений, хотя и кажется несколько неестественным, окажется более удобным впоследствии при реализации ленивых вычислений. Команда `Apply` будет выполняться по-разному в зависимости от того, как представлена применяемая функция `func`. Если функция – это сечение, то применение ее к последнему аргументу вызовет вычисление результата с помощью функции `intrinsic`. Если аргумент – не последний, то он просто добавляется к списку уже вычисленных аргументов. Отсюда имеем два следующих уравнения.

```
evaluate ((Oper n f args):arg:s, e, Apply:c, d)
    | n == 1    = evaluate ((intrinsic f newArgs):s, e, c, d)
    | otherwise = evaluate ((Oper (n-1) f newArgs):s, e, c, d)
        where newArgs = args ++ [arg]
```

Применение функции, представленной замыканием, гораздо сложнее. Для ее вычисления обычно формируют новую SECD-машину, в которой тело лямбда-выражения служит основной командой, контекст переменных пополняется новой связью формального аргумента с фактическим значением, а старое состояние машины запоминается в регистре хранения состояний. После того, как результат работы будет вычислен, старое состояние SECD-машины восстанавливается. Отсюда имеем следующие уравнения для функции `evaluate`.

```
evaluate ((Closure x body env):a:s, e, Apply:c, d) =
         evaluate ([], (x, a) <+> env, [body], (s, e, c):d)
evaluate (x:_, _, [], (s, e, c):d) = evaluate (x:s, e, c, d)
```

Первое из этих уравнений описывает процесс вызова функции, а второе – процесс возврата из функции. В первом уравнении происходит

формирование новой SECD-машины с пустым стеком, регистром контекста, в котором старый контекст пополнен новой связью и регистром управления, содержащим только тело вычисляемой функции. Старое содержимое регистров запоминается в регистре сохранения состояний. Второе уравнение описывает ситуацию, когда команд для выполнения уже нет, однако в регистре сохранения состояний имеется сохраненное предыдущее состояние, а в регистре стека на вершине находится результат работы функции. В этом случае происходит возврат к сохраненному состоянию машины, причем вычисленное значение переносится на вершину «старого» стека.

Сходным образом можно реализовать и вычисление нерекурсивного блока `Let`. Как и в случае вызова функции, надо сначала вычислить в исходном контексте значения «аргументов» блока, а затем организовать вычисление тела блока в новом контексте, пополненном новыми связями переменных блока с вычисленными значениями «аргументов». В отличие от применения лямбда-выражения нам не нужно организовывать замыкание, поскольку тело блока исполняется в том же самом контексте глобальных переменных, что и выражения, сопоставляемые с переменными блока (если не считать самих этих переменных). Второе отличие состоит в том, что у лямбда-выражения только один аргумент, а в блоке определяются сразу несколько переменных, так что команда `Apply` здесь не сработает. Вместо нее добавим в список команд еще одну специальную команду `LetApply`, в которой, помимо самой команды, запомним имена всех переменных блока и тело блока. Таким образом, команда `LetApply` будет представлять собой нечто среднее между командой `Apply` для применения функции и представлением замыкания функции, в котором хранятся имена «формальных аргументов» и «тело функции», но, в отличие от обычного замыкания для функции, нет представления контекста. Сразу же добавим и еще одну команду – `LetrecApply` – для вычисления рекурсивного блока.

```
data Command =
        ... |
    LetApply [String] Command |    -- команда "вычислить блок"
    LetrecApply [String] Command   -- команда "выч. рек. блок"
```

Далее вычисление значения простого блока происходит по той же схеме, что и вычисление значения функции. Прежде всего, в регистр управления выкладываются выражения аргументов для их последовательного вычисления в текущем контексте. После того, как все они будут вычислены, а вычисленные значения будут положены в регистр стека, команда `LetApply` сформирует новый контекст и создаст новую SECD-машину, запомнив состояние старой машины в регистре сохранения состояний. Возврат из блока будет происходить точно так же, как и для обычной функции.

```
evaluate (s, e, (Let pairs body):c, d) =
   evaluate (s, e,
             (reverse values) ++ ((LetApply names body):c), d)
      where (names, values) = unzip pairs
evaluate (s, e, (LetApply names body):c, d) =
   evaluate ([], head <++> e, [body], (tail, e, c):d)
      where (head, tail) = makepairs names s
```

Первое из двух написанных уравнений описывает выполнение команды `Let`, в котором машина подготавливается для выполнения входа в блок. Прежде всего, список пар (*имя*, *выражение*) разбивается на два списка – список имен `names` и список выражений `values` – с помощью стандартной функции обработки списков `unzip`. Это действие описано в предложении `where` в первом уравнении. Затем список выражений «переворачивается» с помощью функции `reverse` и помещается в начало списка команд в регистре управления. «Переворачивание» списка нужно для того, чтобы в регистре стека результаты сформировались таким образом, что первое из выражений находится на вершине стека, а уже за ним следуют все остальные значения аргументов по порядку. Наконец, формируется команда `LetApply` для завершения вычисления блока. Второе уравнение описывает выполнение команды `LetApply`. В ней происходит формирование пополнения контекста из имен, после чего старое состояние SECD-машины запоминается в регистре сохранения состояний, и формируется новая машина. Рассмотрим этот процесс чуть подробнее.

Непосредственно перед выполнением команды `LetApply` в регистре стека оказываются значения выражений, которые необходимо связать с переменными исходного блока `Let`. Если этот блок имел вид

```
let x1 = e1,
    x2 = e2,
    ...
    xn = en
in e
```

то состояние SECD-машины перед началом выполнения команды `LetApply` будет следующим:

```
((e1':e2':...:en':tail), ctx,
                        (LetApply ["x1", "x2",... "xn"] e):c, d)
```

где `e1'`, `e2'`,... `en'` – вычисленные значения выражений `e1`, `e2`,... `en`. Это состояние следует преобразовать в состояние новой SECD-машины, а текущее состояние регистров стека, контекста и управления сохранить в регистре хранения состояний, то есть надо перейти к состоянию

```
([], newCtx, [e], (tail, ctx, c):d)
```

где `newCtx` - это новый контекст, пополненный связями переменных `x1`, `x2`,... `xn` со значениями `e1'`, `e2'`,... `en'`, то есть

```
newCtx = ("x1", e1'):("x2":e2'):...:("xn", en'):ctx
```

Таким образом, функция `makepairs` должна преобразовать аргументы `["x1", "x2",... "xn"]` и `(e1':e2':...:en':tail)` в необходимые списки `[("x1", e1'), ("x2":e2'),... ("xn", en')]` и остаток второго аргумента `tail`. Тогда соответствующее уравнение будет действительно работать так, как описано. Уравнения для функции `makepairs` можно теперь описать следующим образом:

```
makepairs [] s = ([], s)
makepairs (x:names) (e:exprs) = ((x, e):head, tail)
    where (head, tail) = makepairs names exprs
```

Итак, мы описали поведение SECD-машины практически для всех команд, кроме рекурсивного блока `Letrec`. К сожалению, для рекурсивного блока поведение SECD-машины описать в терминах функционального программирования не удается без использования специальных функций, нарушающих основные принципы функционального программирования. Дело в том, что, несмотря на внешнюю схожесть нерекурсивного и рекурсивного блоков, их реализация имеет одно существенное отличие: контекст для вычисления выражений `e1`, `e2`,... `en` должен быть тем же самым, что и новый, пополненный контекст, используемый для вычисления тела блока `e`. Есть, правда, одна существенная разница: при энергичной схеме вычислений контекст при вычислении выражений `e1`, `e2`,... `en` впрямую не используется (то есть, значения переменных `x1`, `x2`,... `xn` не выбираются при вычислениях), так что первоначально, до начала вычисления тела блока можно в этом контексте связать с переменными `x1`, `x2`,... `xn` произвольные значения. Однако, после того, как значения выражений `e1'`, `e2'`,... `en'` будут вычислены, их надо «вставить» в уже сформированный контекст вместо начальных значений, то есть совершить что-то вроде разрушающего присваивания. Если допустить, что такая функция, выполняющая присваивание, есть, то исполнение рекурсивного блока можно будет описать с помощью следующих уравнений.

```
evaluate (s, e, (Letrec pairs body):c, d) =
  evaluate (s, (map (\(x,y) -> (x,(Numeric 0))) pairs) <++> e,
          (reverse values) ++ ((LetrecApply names body):c), d)
     where (names, values) = unzip pairs
evaluate (s, e, (LetrecApply names body):c, d) =
  evaluate ([], newCtx, [body],
            (tail, (drop (length names) e), c):d)
     where (newCtx, tail) = replaceAll names s e
```

Здесь в первом уравнении контекст модифицируется для вычисления значений выражений `e1`, `e2`,... `en`. Для этого берутся все имена `x1`, `x2`,... `xn` из заголовка блока и связываются с произвольно выбранным значением

(Numeric 0). После этого в новом контексте, образованном из старого добавлением новых только что образованных связей, вычисляются значения выражений e1, e2,... en. Далее команда LetrecApply исполняется практически так же, как и команда LetApply, но вместо образования нового контекста при ее исполнении происходит модификация уже имеющегося контекста с помощью функции replaceAll. Эта функция должна заменить в заданном контексте e значения, связанные с именами из списка names, на вычисленные значения, находящиеся в начале списка s. Другими словами, если структура аргументов при вызове будет такой, как показано ниже

```
replaceAll [x1, x2,... xn]
           [e1', e2',...en',s1,...]
           [(x1, ?), (x2, ?),... (xn, ?), (y1, v1),...]
```

то в результате применения функции будет выдан **тот же** контекст, что и значение последнего аргумента функции, только с **замененными** значениями переменных x1, x2,... xn. Результат применения можно будет описать в виде пары следующих списков:

```
([(x1, e1'), (x2, e2'),... (xn, en'), (y1, v1),...], [s1,...])
```

Нетрудно написать подобную функцию, которая будет формировать действительно новое значение контекста, но в функциональном программировании невозможно подменить в списке одни значения другими. Это, однако, является существенным элементом реализации, поскольку контекст, содержащий неопределенные значения переменных x1, x2,... xn, может быть сохранен при вычислении значений выражений e1, e2,... en (скорее всего, это обязательно случится, поскольку в выражениях e1, e2,... en наверняка будут встречаться определения рекурсивных функций, работающих в этом контексте), а значит, нужно на самом деле произвести замену значений так, чтобы при использовании этого контекста переменные после первого же обращения к ним получили бы уже новые – вычисленные – значения и впоследствии имели бы связи уже с этими новыми значениями.

Если мы все же попробуем реализовать функцию replaceAll в чисто функциональном стиле, то есть в виде функции, создающей и возвращающей не старый контекст с измененными значениями переменных, а вновь построенный контекст, то впоследствии каждое рекурсивное обращение изнутри этого контекста будет приводить к ошибкам, так как для таких обращений будет фактически использоваться старый контекст с неопределенными связями.

Конечно, при реализации SECD-машины в виде обычной последовательно исполняющейся (императивной) программы никаких проблем с формированием «зацикливающегося» контекста не будет.

Основным преимуществом описания семантики исполнения программ с помощью SECD-машины перед описанием семантики с

помощью интерпретатора является простота исполнения и независимость этой семантики от интерпретирующей системы самой машины. Действительно, поведение нашей SECD-машины при исполнении ею программы почти не зависит от того, как реализован язык *Haskell*. Мы реализовали энергичную SECD-машину, и вправе ожидать, что при попытке вычислить значение выражения

```
test =
   Application
      (Lambda "x" (Integral 1))
      (Application (Application
            (Function "DIV") (Integral 1)) (Integral 0))
```

будет зафиксирована ошибка при делении на ноль, несмотря на то, что аргумент, при вычислении которого и происходит эта ошибка, не используется в вызванной функции `(λx.1)`. Такое поведение характерно для энергичной схемы вычислений. Однако, если мы попробуем пропустить такой тест, то мы увидим, что результатом работы интерпретатора все-таки будет целое число 1. Это происходит потому, что ленивый инструментальный язык *Haskell* опять будет задерживать вычисление значения аргумента функции до первого обращения к нему, а этого обращения как раз и не произойдет. Чтобы реализовать полностью энергичную модель вычисления, пришлось бы внести в программу довольно много мелких изменений, связанных с принудительным вычислением аргументов функций при вызовах.

На самом деле это свидетельствует о том, что функциональный язык программирования *Haskell* плохо подходит для реализации последовательного исполнения команд, как это предписывается архитектурой SECD-машины. Для полноты все же приведем в листинге 3.2 полный текст программы, реализующей нашу SECD-машину. В этом тексте приводится определение даже такой «неправильной» функции, как `replaceAll` (она определена в ней с помощью более простой функции работы с контекстом `replace`). Однако эта функция не работает так, как это было описано, поэтому на самом деле описанная нами программа не будет правильно обрабатывать рекурсивные блоки, если в них действительно содержатся рекурсивные обращения к определяемым переменным, так что программа приводится не для практического исполнения, а, скорее, просто для того, чтобы иметь представление об общем объеме и некоторых деталях работы SECD-машины.

Листинг 3.2. Реализация энергичной SECD-машины

```
-- Основной тип данных для представления выражений
-- расширенного лямбда-исчисления
-- "команды" машины
data Command =
    Integral Integer | Logical Bool | Function String |
```

```
                                        -- константы
    Variable String |                   -- переменная
    Lambda String Command |             -- лямбда-выражение
    Application Command Command |       -- применение функции
    If Command Command Command |        -- условное выражение
    Let [(String, Command)] Command |   -- нерекурсивный блок
    Letrec [(String, Command)] Command | -- рекурсивный блок
    Select Command Command |      -- выбор из альтернатив
    Apply |                       -- применить функцию
    LetApply [String] Command |   -- вычислить значение блока
    LetrecApply [String] Command  -- выч. значение рек. блока
         deriving (Show, Eq)

-- "результаты вычислений"
data WHNF =
    Numeric Integer | Boolean Bool |  -- константы
    Closure String Command Context |  -- замыкание
    Oper Int String [WHNF]            -- сечение
         deriving (Show, Eq)

-- Контекст вычислений представлен ассоциативным списком
type Context = [(String, WHNF)]

-- Типы, определеяющие высокоуровневую SECD-машину
type Stack = [WHNF]
type Environment = Context
type Control = [Command]
type Dump = [(Stack, Environment, Control)]
type SECD = (Stack, Environment, Control, Dump)

-- Определения типов основных и вспомогательных функций
---- Интерпретатор SECD-машины:
interpret :: Command -> WHNF
evaluate  :: SECD -> SECD

---- Функции для применения стандартных операций
infixr 6 <+>
infixr 5 <++>
(<+>)      :: (String, WHNF) -> Context -> Context
(<++>)     :: Context -> Context -> Context
assoc      :: String -> Context -> WHNF
replace    :: String -> WHNF -> Context -> Context
arity      :: String -> Int
intrinsic  :: String -> [WHNF] -> WHNF

-- Определения уравнений для вспомогательных функций
---- Поиск по контексту:
assoc x ((y, e):ctx) | x == y = e
                     | otherwise = assoc x ctx
---- Добавление в контекст
```

```
(<+>) = (:)
(<++>) = (++)
---- Замена в контексте:
----  НЕ ВЫПОЛНЯЕТ НЕОБХОДИМОЕ ЗДЕСЬ РАЗРУШАЮЩЕЕ ПРИСВАИВАНИЕ!
replace x e ((y, e'):ctx)
                      | x == y    = ((y, e):ctx)
                      | otherwise = (y, e'):(replace x e ctx)

---- Определение числа операндов стандартных операций
arity "ADD" = 2
arity "SUB" = 2
arity "MULT" = 2
arity "DIV" = 2
arity "EQ0" = 1
arity "SUCC" = 1
arity "PRED" = 1

---- Выполнение стандартных операций над списком аргументов
intrinsic "ADD" [Numeric(a), Numeric(b)] = Numeric (a+b)
intrinsic "SUB" [Numeric(a), Numeric(b)] = Numeric (a-b)
intrinsic "MULT" [Numeric(a), Numeric(b)] = Numeric (a*b)
intrinsic "DIV" [Numeric(a), Numeric(b)] = Numeric (a `div` b)
intrinsic "EQ0" [Numeric(a)] = Boolean (a==0)
intrinsic "SUCC" [Numeric(a)] = Numeric (a+1)
intrinsic "PRED" [Numeric(a)] = Numeric (a-1)

-- Определения уравнений для основных функций
---- Интерпретатор:
interpret e = e' where (e':_, _, _, _) =
                  evaluate ([], [], [e], [])

evaluate secd@(_, _, [], [])      = secd
evaluate (x, _, [], (s, e, c):d)  = evaluate (x ++ s, e, c, d)
evaluate (s, e, (Integral n):c, d) =
                  evaluate ((Numeric n):s, e, c, d)
evaluate (s, e, (Logical b):c, d) =
                  evaluate ((Boolean b):s, e, c, d)
evaluate (s, e, (Function f):c, d) =
                  evaluate ((Oper (arity f) f []):s, e, c, d)
evaluate (s, e, (Variable x):c, d) =
                  evaluate ((assoc x e):s, e, c, d)
evaluate (s, e, (Lambda x body):c, d) =
                  evaluate ((Closure x body e):s, e, c, d)
evaluate (s, e, (If cond th el):c, d) =
                  evaluate (s, e, cond:(Select th el):c, d)
evaluate ((Boolean True):s, e, (Select th el):c, d)  =
                  evaluate (s, e, th:c, d)
evaluate ((Boolean False):s, e, (Select th el):c, d) =
                  evaluate (s, e, el:c, d)
evaluate (s, e, (Application f a):c, d) =
```

```
                evaluate (s, e, a:f:Apply:c, d)
evaluate ((Closure x body env):a:s, e, Apply:c, d) =
          evaluate ([], (x, a) <+> env, [body], (s, e, c):d)
evaluate ((Oper n f args):a:s, e, Apply:c, d)
   | n == 1    = evaluate ((intrinsic f newArgs):s, e, c, d)
   | otherwise = evaluate ((Oper (n-1) f newArgs):s, e, c, d)
       where newArgs = args ++ [a]
evaluate (s, e, (Let pairs body):c, d) =
   evaluate (s, e,
             (reverse values) ++ ((LetApply names body):c), d)
                 where (names, values) = unzip pairs
evaluate (s, e, (LetApply names body):c, d) =
          evaluate ([], head <++> e, [body], (tail, e, c):d)
              where (head, tail) = makepairs names s
evaluate (s, e, (Letrec pairs body):c, d) =
  evaluate (s, (map (\(x,y) -> (x,(Numeric 0))) pairs) <++> e,
          (reverse values) ++ ((LetrecApply names body):c), d)
              where (names, values) = unzip pairs
evaluate (s, e, (LetrecApply names body):c, d) =
  evaluate ([], newCtx, [body],
            (tail, (drop (length names) e), c):d)
              where (newCtx, tail) = replaceAll names s e

-- makepairs [x1,... xn] [e1,... en, s1,...] =
--                          ([(x1, e1),... (xn, en)], [s1,...])
makepairs [] s = ([], s)
makepairs (x:names) (e:exprs) = ((x, e):head, tail)
    where (head, tail) = makepairs names exprs

-- replaceAll [x1, x2,... xn] [e1', e2',...en',s1,...]
--            [(x1, ?), (x2, ?),... (xn, ?), (y1, v1),...] =
--         ([(x1, e1'),... (xn, en'), (y1, v1),...], [s1,...])
replaceAll [] s e = (e, s)
replaceAll (x:nms) (e:exprs) ctx =
                        replaceAll nms exprs (replace x e ctx)
```

Мы реализовали энергичную SECD-машину. Возникает естественный вопрос: что нужно изменить в нашей программе, чтобы поведение SECD-машины изменилось и стало бы соответствовать ленивой схеме вычисления выражений? Очевидно, что для этого, прежде всего, надо изменить реализацию вызова функции, чтобы вычислять значение ее аргумента лишь при первом обращении к нему. Конечно, такие же изменения следует внести и в реализацию вычисления блоков. Поскольку все наши вычисления основаны на хранении значений в контексте, то все, что для этого требуется – это задержать вычисление значения при помещении его в контекст. Теперь в тот момент, когда значение, хранящееся в контексте, понадобится в качестве аргумента строгой функции (первое обращение!), нужно будет произвести вычисление и заменить в контексте значение переменной вычисленным значением. Для

этого опять потребуется применение функции, подобной функциям `replace` или `replaceAll`.

Мы введем понятие *задержки* – это новый тип выражения, который содержит команду (выражение, подлежащее вычислению) и контекст, в котором это вычисление следует производить. Заметим, что задержка по своей структуре напоминает *замыкание*, с помощью которого ранее мы представляли функциональные значения. Такое сходство на самом деле не случайно, поскольку фактически задержки представляют собой функции без аргументов, которые требуется вычислять при обращении к ним. В нашем описании типа `WHNF` появится определение нового конструктора для реализации понятия задержки:

```
data WHNF =
        ... |
    Delay Command Context
```

а команда применения функции будет теперь выполняться таким образом, чтобы задержать вычисление аргумента:

```
evaluate (s, e, (Application f a):c, d) =
                      evaluate ((Delay a e):s, e, f:Apply:c, d)
```

Команда применения функции `Apply`, как и раньше, приведет к тому, что если функция представлена своим замыканием, то задержанный аргумент будет добавлен в контекст переменных для вычисления значения функции. Таким образом задержанные выражения смогут попасть в контекст переменных.

В дальнейшем задержанное выражение может быть вычислено в тот момент, когда это значение понадобится, фактически, при обращении к аргументу примитивной функции (напомним, что сопоставление с образцом у нас также реализуется с помощью вызова примитивной функции – функции взятия элемента кортежа).

Но чтобы реализовать механизм ленивых вычислений адекватно, необходимо, чтобы вычисление выражения происходило не более одного раза. С помощью чисто функциональных средств легко можно реализовать работу SECD-машины, при которой задержанное выражение вычисляется каждый раз, когда происходит обращение к нему, но вычисление выражения один раз при первом обращении требует хранения результата вычислений на том же месте, которое раньше занимала задержка. Таким образом, необходимо выполнять присваивания, которые в чисто функциональном виде представить невозможно. В первом случае можно говорить не о ленивых вычислениях, а, скорее, о передаче аргументов «по наименованию», как это описывалось в начале раздела «ленивые вычисления» главы 2.

Для реализации передачи аргументов «по наименованию» достаточно организовать вычисление задержанных выражений всякий раз перед тем, как происходит обращение к функции для вычисления

результата работы примитивной функции, строгой по своим аргументам. Если считать, что все примитивные функции строгие по всем своим аргументам, то изменить потребуется уравнение, в котором описывается команда `Apply` для случая применения примитивной функции. Это та пара уравнений, которая раньше выглядела следующим образом:

```
evaluate ((Oper n f args):a:s, e, Apply:c, d)
   | n == 1     = evaluate ((intrinsic f newArgs):s, e, c, d)
   | otherwise  = evaluate ((Oper (n-1) f newArgs):s, e, c, d)
      where newArgs = args ++ [a]
```

Здесь потребуется изменить работу в том случае, когда аргумент функции представлен задержкой: перед применением функции задержку необходимо «инициировать», организовав вычисление задержанного выражения. Это можно сделать практически так же, как и для случая вычисления значения обычной функции или простого блока, инициировав «вход в функцию». Различие проявляется при выходе из функции, так как вычисленное значение требуется поместить не на вершину стека, а на то же место, которое занимал аргумент примитивной функции перед началом вычисления. Для этого можно ввести специальную команду `DelayRet`, которая осуществит «специальный» выход из функции. Эту команду следует поместить после команды для вычисления задержанного выражения.

Итак, представим необходимые изменения. Во-первых, добавим новую команду:

```
data Command =
        ... |
    DelayRet       -- команда "возврат из вычисления задержки"
```

Во-вторых, добавим уравнения, которые организуют вычисление задержки перед применением примитивной функции к аргументу:

```
evaluate (prim@(Oper n f args):(Delay com env):s,e,Apply:c,d)=
         evaluate ([], env, [com, DelayRet], (prim:s, e, c):d)
```

Наконец, реализуем выполнение команды `DelayRet`:

```
evaluate ([x], _, [DelayRet], (prim:s, e, c):d)   =
         evaluate (prim:x:s, e, Apply:c, d)
```

Это все, что нужно сделать для реализации передачи аргументов «по наименованию». Заметим, что, вообще говоря, после возврата из функции для вычисления задержки может оказаться, что вычисленный результат в свою очередь окажется задержкой. В этом случае придется снова «входить в функцию» для вычисления этой новой задержки и повторять этот цикл до тех пор, пока вычисленное значение окажется чем-нибудь, отличным от задержки.

Чтобы реализовать ленивые вычисления в полном объеме, необходимо обеспечить механизм, который позволил бы один раз вычисленное значение задержки в дальнейшем использовать, не проводя

те же вычисления повторно. Для этого средств функционального программирования недостаточно, так как требуется выполнять «присваивание» вычисленного значения на место, где до этого хранилась задержка.

Можно представлять себе, что фактически задержки помещаются в некоторую специально выделенную область памяти, а в регистрах SECD-машины содержатся ссылки на эти задержки. При вычислении значения задержки в эту область памяти помещается вычисленное выражение, так что при последующих обращениях к ним новых вычислений делать уже не нужно.

Такой подход к реализации приводит нас к концепции *помеченного выражения* – выражения в СЗНФ, которое содержит ссылку на область памяти, содержащую либо задержку, либо уже вычисленное выражение, которое задержкой не является. Такое помеченное выражение играет роль обычного выражения в СЗНФ, но в тех случаях, когда требуется его значение, фактически происходит обращение к значению в той области памяти, на которое указывает ссылка. В том случае, когда значение требуется для вычисления значения строгой примитивной функции, может происходить вычисление задержанного выражения, при этом результат вычисления помещается на место задержки, так что при последующих обращениях вычисления заново не производятся.

Мы не будем изменять нашу реализацию в соответствии с описанным поведением, поскольку все равно в чисто функциональном виде этого сделать не удастся.

Еще одно замечание: мы не описывали поведение ленивой SECD-машины при вычислении ею блоков `Let` и `Letrec`. Во-первых, заметим, что при ленивых вычислениях фактически нет разницы между этими двумя видами блоков, поскольку вычисление значений определяемых переменных все равно задерживается до момента первого обращения к ним из тела блока. Во-вторых, реализация блоков происходит примерно так же, как мы это делали для команды применения функции: вычисление значений переменных задерживается и образованные задержки помещаются в создаваемый контекст. Впоследствии при вычислении тела блока задержанные вычисления будут, как и раньше, производиться при первом обращении к соответствующим переменным из строгой примитивной функции.

Наконец, заметим, что если реализуются ленивые вычисления (или хотя бы передача аргументов в функцию «по наименованию»), то нет необходимости во введении специальной конструкции для условного вычисления – команды `If`. Вместо этого можно просто считать, что есть примитивная функция `IF`, которая имеет один строгий и два нестрогих аргумента, так что фактически в момент работы этой функции вычисляется только условие, а два других аргумента задерживаются. В результате

работы функции один из этих аргументов отбрасывается, а второй будет являться результатом работы функции. Конечно, этот результат обязательно окажется «задержкой».

Мы описали архитектуру и поведение высокоуровневой SECD-машины. Однако, на самом деле более традиционным способом описания SECD-машины является описание низкоуровневой SECD-машины, при котором выполняемыми командами служат не исходные выражения, как это было у нас на протяжении всего раздела, но сравнительно короткие специально спроектированные команды. Обращения к переменным и их имена также заменяются более простыми конструкциями – так называемыми адресными парами, которые лишь указывают местоположение в контексте нужного значения. Исходное выражение при таком подходе компилируется в последовательность команд, которые могут исполняться SECD-машиной, так что весь процесс вычислений разбивается на два этапа: компиляцию исходного выражения и интерпретацию полученной последовательности команд SECD-машиной.

Чтобы проиллюстрировать этот подход, рассмотрим наш простой язык расширенного лямбда-исчисления и опишем процесс компиляции этого языка в последовательность команд SECD-машины, а также опишем процесс исполнения этих команд. Для этого прежде всего опишем процесс преобразования имен переменных в адресные пары.

В каждый момент времени при исполнении функциональной программы имеется контекст переменных, составленный из имен формальных аргументов исполняемых функций и переменных, полученных при их описании в `let-` и `letrec-`блоках. Так, например, при вычислении значения выражения

```
letrec x = 2,
    f = λy.* x y
in f 3
```

в тот момент, когда вычисляется произведение двух чисел с помощью стандартной функции умножения целых, контекст переменных состоит из аргумента функции `y` (со значением `3`) и двух переменных `x` и `f`, определенных в блоке `letrec`. Этот контекст состоит на самом деле из двух «уровней» определения переменных – уровня описания переменных в блоке и вложенного в него уровня описания формального аргумента функции. Уровни принято нумеровать целыми числами, начиная от нуля, при этом самый внутренний блок получает номер ноль, а каждый из охватывающих его уровней получает номер, на единицу больший него. Внутри каждого уровня переменные также нумеруются числами, начиная с нуля. Таким образом, каждая переменная будет характеризоваться двумя числами – номером уровня и номером переменной внутри этого уровня. Эти два числа и образуют так называемую адресную пару переменной.

В приведенном выше примере в точке выполнения умножения переменная `y` имеет адресную пару (0,0), поскольку находится на нулевом уровне и является единственной переменной на этом уровне. Переменная `x` будет иметь адресную пару (1,0), поскольку это первая переменная на уровне 1, а переменная `f` будет однозначно характеризоваться своей адресной парой (1,1). Заметим, что адресная пара одной и той же переменной будет разной в зависимости от того места в программе, в котором осуществляется доступ к этой переменной. Так, например, в момент вызова функции `f` в том же самом примере переменная `x` будет характеризоваться адресной парой (0,0), а переменная `y` в этот момент вообще недоступна и, следовательно, в адресной паре не нуждается.

Если раньше мы использовали контекст переменных в виде ассоциативного списка, в котором переменные хранились вместе со своими значениями, то теперь в регистре контекста SECD-машины будут храниться только значения, доступ к которым будет осуществляться с помощью адресных пар. Содержимым E-регистра будет список элементов, каждый из которых представляет собой список выражений, являющихся значениями переменных одного уровня. Например, для приведенной выше программы в момент выполнения команды умножения содержимое регистра контекста будет выглядеть следующим образом:

```
[[3], [2, f']]
```

где через `f'` обозначено замыкание, представляющее лямбда-выражение `(λy.* x y)`.

Несложно определить функцию, которая по заданной адресной паре `(i, j)` будет выдавать значение из так организованного контекста:

```
getValue (level, number) context = context !! level !! number
```

Если SECD-машина будет иметь в регистре контекста указанный выше список, то программа умножения будет выполнена с помощью последовательного исполнения нескольких команд, первые из которых загружают на вершину стека значения переменных `x` и `y`, а также знак операции умножения `*`, а следующие две команды выполняют собственно умножение. Поскольку обращение к переменным происходит с помощью адресных пар, то выглядеть эти команды будут примерно следующим образом:

```
Load (0, 0);    -- загрузка переменной y
Load (1, 0);    -- загрузка переменной x
LoadFunc "*";   -- загрузка функции умножения
Apply;          -- применение умножения к первому операнду
Apply;          -- применение результата ко второму операнду
```

Для того, чтобы перевести (скомпилировать) программу на исходном языке расширенного лямбда-исчисления в эту последовательность команд, надо в момент компиляции знать, каким переменным какие адресные пары

будут соответствовать в каждый момент выполнения программы. Поэтому при компиляции надо иметь список, в котором хранятся переменные и соответствующие им адресные пары. Проще всего организовать этот список в точности так же, как и список адресных пар, тогда адресная пара будет просто вычисляться по положению переменной в списке. Например, в момент компиляции выражения `(* x y)` в вышеприведенном примере надо, чтобы список активных переменных имел структуру

```
[["y"], ["x", "f"]]
```

Приведем определение функции `addr`, которая по заданному списку переменных и переменной из этого списка вычисляет адресную пару этой переменной. Для этого сначала определим функцию `pos`, которая ищет переменную в простом списке имен и выдает позицию переменной в этом списке, если она там есть, и число -1, если переменной в списке нет.

```
pos :: String -> [String] -> Int
pos var (head:tail)
          | var == head  = 0  -- переменная первая в списке
          | otherwise    = if k == -1 then -1 else 1 + k
              where k = pos var tail
pos _ [] = -1
```

теперь функция `addr` определяется просто:

```
addr :: String -> [[String]] -> (Int, Int)
addr x (level:list) =
    if j >= 0 then (0, j)
              else (let (p, q) = addr x list in (p+1, q))
      where j = pos x level
```

Теперь, когда мы умеем вычислять адресные пары переменных, можно приступить к процессу компиляции программы в последовательность команд нашей SECD-машины низкого уровня. Однако, перед тем, как это делать, надо рассмотреть подробнее, как работает эта машина.

Структура регистров машины низкого уровня такая же, как и у машины высокого уровня, только в регистре контекста хранится не ассоциативный список переменных с их значениями, а двухуровневый список значений, которые можно получать с помощью адресных пар. Таким образом, как и раньше, имеем следующее описание SECD-машины:

```
type Context = [[WHNF]]
type Stack = [WHNF]
type Environment = Context
type Control = [Command]
type Dump = [(Stack, Environment, Control)]
type SECD = (Stack, Environment, Control, Dump)
```

Основное отличие этой машины от машины высокого уровня состоит в наборе команд и правилах их выполнения. Команды представляют собой не сложные выражения исходного языка, как это было

раньше, а простые инструкции, содержащие в большинстве случаев только простые операнды или вообще не содержащие операндов. Мы все же будем допускать, чтобы некоторые команды содержали внутри себя последовательности других команд. На самом деле, это делает наши команды командами высокого уровня, имеющими структуру дерева. Мы, однако, надеемся, что читатели при некотором желании смогут понять, как можно добавить еще некоторые команды назкого уровня для того, чтобы избавиться полностью от команд, имеющих структуру дерева.

Мы будем использовать составные команды для организации управления. Примерами составных команд могут служить такие команды как команда выбора из альтернатив `If` или команда образования замыкания для функции `LoadFunc`.

В список команд SECD-машины низкого уровня часто включают и команды для выполнения примитивных функций расширенного лямбда-исчисления, таких как сложение и другие арифметические операции, сравнения, логические операции, команды образования списков `CONS` и кортежей `TUPLE-k` и другие. Мы не будем этого делать, считая, что примитивные операции, как и раньше, будут исполняться некоторой интерпретирующей системой, представленной функцией `intrinsic`, получающей в качестве аргументов идентификатор операции и список ее операндов. Для загрузки значений на стек будут также использоваться команды загрузки констант (`LoadConst`), имеющие в качестве аргумента простое значение загружаемой константы. Конечно, будет также использоваться команда загрузки значения из контекста по адресной паре (`Load`), команда загрузки замыкания для функции (`LoadFunc`), операндом которой будет список команд, представляющих тело функции, команда загрузки знака примитивной операции (`LoadOp`). Кроме этого потребуются также команды применения функции `Apply` и еще несколько вспомогательных команд, смысл которых будет разъяснен позже.

Программа, записанная на исходном языке расширенного лямбда-исчисления, сначала переводится (компилируется) в последовательность команд SECD-машины низкого уровня. Далее, если требуется исполнить эту программу, запускается интерпретатор команд SECD-машины. Компиляция исходного текста – это обычная функция, которая получает в качестве аргумента исходную программу и выдает в качестве результат список команд. Интерпретатор команд SECD-машины можно тоже описать в функциональном виде примерно так же, как мы это делали для SECD-машины высокого уровня, однако, как и раньше, такое описание будет недостаточным из-за того, что потребуется выполнять команды, которые невозможно описать в чисто функциональных терминах, так как они включают в себя изменение уже имеющихся значений (присваивание). Как и раньше, для энергичной SECD-машины это касается только вычисления значения рекурсивного блока, а для ленивой машины такое присваивание

потребуется еще и для реализации повторных обращений к аргументам функции, вычисление которых было задержано до момента первого обращения к ним.

Мы не будем писать программу на языке *Haskell* для реализации компилятора и интерпретатора SECD-машины низкого уровня. Вместо этого мы сделаем следующее. Во-первых, определим типы данных для представления исходных программ, результатов работы (значения в виде СЗНФ) и команд SECD-машины. После этого мы опишем компиляцию в виде уравнений для функции компиляции `comp`. Уравнения будем записывать неформально, однако переписать их на языке *Haskell* большого труда не представляет. Наконец, опишем работу интерпретатора в терминах переходов из одного состояния SECD-машины в другое, имеющих вид

`(s, e, c, d)` $\Rightarrow$ `(s', e', c', d')`

где `(s, e, c, d)` представляет состояние SECD-машины до выполнения команды, а `(s', e', c', d')` – после ее выполнения. Конечно, правила такого перехода в основном зависят от первой из команд, лежащих в регистре управления.

Итак, начнем с описания выражений исходного языка программирования. На самом деле, это описание практически полностью совпадает с описанием типа `Expr`, как оно появилось у нас в первый раз при описании интерпретатора. Добавим только конструкцию для представления пустого списка `Nil` и снова введем конструкцию для явного представления условного выражения, которое невозможно выразить в виде встроенной функции при энергичных вычислениях. У нас получится следующее описание.

```
data Expr =
    Integral Integer             | -- целая константа
    Logical Bool                 | -- логическая константа
    Nil                          | -- константа - пустой список
    Function String              | -- встроенная функция
    Variable String              | -- переменная
    Lambda   String Expr         | -- лямбда-выражение
    Application Expr Expr        | -- применение функции
    If Expr Expr Expr            | -- условное выражение
    Let [(String, Expr)] Expr    | -- нерекурсивный блок
    Letrec [(String, Expr)] Expr  -- рекурсивный блок
```

Выражения в слабой заголовочной нормальной форме будут представлены значениями, которые можно описать в виде следующего описания типа и его конструкторов:

```
data WHNF =
    Numeric Integer          | -- целая константа
    Boolean Bool             | -- логическая константа
    List [WHNF]              | -- список (в том числе - пустой)
```

```
    Closure [Command] Context  |   -- замыкание для функции
    Op String Int [WHNF]  -- встроенная функция или ее сечение
```

В этом описании, как и в предыдущем, нет ничего нового, кроме представления списка значений. Отметим, что список может содержать любые значения, так что, в отличие от языка *Haskell*, в нашем расширенном лямбда-исчислении список можно образовать из значений разных типов, соединяя в одном списке, например, целые числа, вложенные списки и замыкания, представляющие функции.

Теперь опишем набор команд SECD-машины. В большинстве случаев смысл команд понятен интуитивно или ясен из предыдущих абзацев. Смысл некоторых команд будет понятен при рассмотрении работы SECD-машины.

```
data Command =
    Load (Int, Int)    | -- загрузка значения по адресной паре
    LoadConst WHNF     | -- загрузка констант
    LoadFunc [Command] | -- загрузка замыкания
    LoadOp String      | -- загрузка встроенной операции
    Select [Command] [Command] |  -- выбор альтернативы
    Apply              | -- применение функции
    Return             | -- возврат из функции
    LetApply           | -- вход в нерекурсивный блок
    Dummy              | -- образование фиктивного контекста
    RecApply           | -- вход в рекурсивный блок
    Stop                 -- остановка работы
```

Работу компилятора мы будем описывать в виде уравнений функции `comp`, в левой части которых находится некоторая конструкция языка – выражение типа `Expr`, а в правой части – результирующая последовательность команд. Кроме исходного выражения аргументом функции `comp` является также контекст переменных, так что компиляция выражения всегда производится в некотором контексте переменных. Для упрощения записи будем считать, что функция `comp` представлена бинарной операцией `**`, так что наши уравнения будут иметь вид

```
expr ** context  =  [список команд]
```

Так, например, выражение `(Integral n)` компилируется в команду загрузки целой константы, так что соответствующее выражение будет выглядеть следующим образом:

```
(Integral n) ** context  =  [LoadConst (Numeric n)]
```

Так же просто описываются и уравнения для компиляции еще некоторых простых типов выражения:

```
(Logical b)  ** context  =  [LoadConst (Boolean b)]
(Nil)        ** context  =  [LoadConst (List [])]
(Function f) ** context  =  [LoadOp f]
(Variable x) ** context  =  [Load (addr x context)]
```

Чтобы скомпилировать условное выражение, надо сначала скомпилировать команды для вычисления условия, а также для вычисления обеих альтернатив условного выражения. После этого команды расставляются таким образом, чтобы сначала было вычислено условие, а потом исполнилась бы команда `Select`, которая уже и произведет выбор, какую из двух оставшихся последовательностей команд надо выполнять. Таким образом, соответствующее уравнение будет записано так:

```
(If cond eThen eElse) ** context  =  (cond ** context) ++
                [Select (eThen ** context) (eElse ** context)]
```

Разумеется, компиляция всех подвыражений этого выражения производится в одном и том же контексте.

Чуть-чуть сложнее выглядит команда компиляции лямбда-выражения. Сложность заключается в том, что команды, составляющие тело лямбда-выражения, должны компилироваться не в исходном контексте, а в контексте, в котором добавлен новый уровень переменных, содержащий единственную переменную – формальный аргумент лямбда-выражения. Поскольку контекст представляет собой список уровней, то добавление к нему нового уровня, содержащего переменную `x`, можно описать с помощью выражения `[x]:context`. Тогда все уравнение для представления результата компиляции лямбда-выражения, можно записать так:

```
(Lambda x body) ** context  =
                [LoadFunc ((body ** ([x]:context)) ++ [Return]))
```

то есть тело лямбда-выражения компилируется в расширенном контексте, к полученной последовательности команд добавляется еще команда возврата из функции `Return`, и вся эта скомпилированная последовательность команд является единственным операндом команды `LoadFunc`.

Компиляция выражения, представляющего собой применение функции к аргументу, выполняется совершенно естественным при энергичных вычислениях способом, а именно: сначала должны выполняться команды вычисления значения аргумента, затем команды, вычисляющие функцию (в простейшем случае это будет единственная команда загрузки функции), и, наконец, должна выполниться команда `Apply`, которая и применяет вычисленную функцию к вычисленному аргументу. Все это можно записать в виде следующего уравнения:

```
(Application func arg) ** context  =
                (arg ** context) ++ (func ** context) ++ [Apply]
```

Осталось написать два самых сложных уравнения для компиляции блоков `Let` и `Letrec`. Для компиляции блока `Let` воспользуемся тем, что вычисление этого блока эквивалентно применению функции, у которой тело лямбда-выражения совпадает с телом блока, а аргументом

являются выражения, вычисленные в заголовке блока. Единственное отличие состоит в том, что функция всегда применяется только к одному аргументу, а в блоке определяются и используются, вообще говоря, несколько аргументов, и поэтому вычисленные значения аргументов необходимо собрать в единый список, который и присоединяется к контексту при входе в тело блока.

Итак, тело блока скомпилируем так же, как и тело обычного лямбда-выражения; контекст для компиляции получен соединением списка переменных блока с текущим контекстом. Выражения, входящие в заголовок блока, компилируются в текущем контексте, при этом получившиеся значения надо записать в список, для чего можно воспользоваться последовательным применением встроенной функции `cons`. Наконец, выполняется вход в функцию с помощью команды применения функции, работа которой, правда, немного отличается от работы обычной команды `Apply`: отличие состоит в том, что аргументы входа уже упакованы в список. Так что введем специальную команду входа в нерекурсивный блок - `LetApply`.

```
(Let [(x1,e1),...,(xn,en)] body) ** context  =
        [LoadConst (List [])] ++
        (en ** context) ++ [LoadOp "cons", Apply, Apply] ++
        ...
        (e1 ** context) ++ [LoadOp "cons", Apply, Apply] ++
        [LoadFunc (body ** ([x1,...,xn]:context)), LetApply]
```

При исполнении сгенерированной последовательности команд сначала в стек будет загружен пустой список, к которому будут последовательно присоединяться значения вычисленных «аргументов» блока, причем вычисления производятся от последних аргументов к первым. Каждое вычисленное значение присоединяется к списку значений с помощью команд загрузки операции `cons` и двух команд применения функции `Apply` (две команды необходимы потому, что функция `cons` имеет два аргумента). После этого в стек будет загружено тело блока в виде замыкания функции, построенного для текущего контекста значений и, наконец, с помощью команды `LetApply` будет построен новый контекст, и произойдет вход в блок.

Компиляция рекурсивного блока производится точно так же. Имеются лишь три небольших отличия. Во-первых, компиляция выражений, входящих в заголовок блока, производится в том же контексте, что и компиляция тела блока, поскольку в этих выражениях допустимо обращение к переменным заголовка. Во-вторых, для того, чтобы вычисления происходили в расширенном контексте, необходимо сформировать этот расширенный контекст еще **до** начала вычислений. Действительного обращения к значениям из этого контекста при вычислении выражений происходить не будет (напомним, что речь идет об

энергичных вычислениях, так что любое прямое обращение к переменным блока из выражений заголовка блока немедленно привело бы к зацикливанию программы; обычно переменные содержатся в этих выражениях только внутри тел лямбда-выражений), так что контекст пока можно сформировать фиктивный, не содержащий действительных значений. Для формирования такого фиктивного контекста введем специальную команду SECD-машины – `Dummy`. В третьих, вход в рекурсивный блок происходит в ситуации, когда текущим контекстом является фиктивный контекст, сформированный командой `Dummy`, поэтому при входе надо заменить часть этого контекста списком вычисленных значений. Это будет делать команда входа в рекурсивный блок - `RecApply`. Итак,

```
(Letrec [(x1,e1),...,(xn,en)] body) ** context  =
         [Dummy, LoadConst (List [])] ++
         (en ** ([x1,...,xn]:context)) ++
         [LoadOp "cons", Apply, Apply] ++
         ...
         (e1 ** ([x1,...,xn]:context)) ++
         [LoadOp "cons", Apply, Apply] ++
         [LoadFunc (body ** ([x1,...,xn]:context)), RecApply]
```

Программирование компилятора закончено. Заметим, что, несмотря на то, что уравнения для компилятора были даны в неформальном виде, их несложно перевести в чисто функциональную программу. Для полноты изложения приведем эту программу в листинге 3.3 целиком.

Листинг 3.3. Компиляция выражений расширенного лямбда-исчисления в низкоуровневую SECD-машину

```
-- Основной тип данных для представления выражений
-- расширенного лямбда-исчисления и "команд" SECD-машины
data Expr =
    Integral Integer            | -- целая константа
    Logical Bool                | -- логическая константа
    Nil                         | -- константа - пустой список
    Function String             | -- встроенная функция
    Variable String             | -- переменная
    Lambda   String Expr        | -- лямбда-выражение
    Application Expr Expr       | -- применение функции
    If Expr Expr Expr           | -- условное выражение
    Let [(String, Expr)] Expr   | -- нерекурсивный блок
    Letrec [(String, Expr)] Expr  -- рекурсивный блок
             deriving (Show, Eq)
data Command =
    Load (Int, Int)    | -- загрузка значения по адресной паре
    LoadConst WHNF     | -- загрузка константы
    LoadFunc [Command] | -- загрузка замыкания
    LoadOp String      | -- загрузка встроенной операции
    Select [Command] [Command] |  -- выбор альтернативы
```

```
    Apply               | -- применение функции
    Return              | -- возврат из функции
    LetApply            | -- вход в нерекурсивный блок
    Dummy               | -- образование фиктивного контекста
    RecApply            | -- вход в рекурсивный блок
    Stop                  -- остановка работы
             deriving (Show, Eq)

-- "результаты вычислений"
data WHNF =
    Numeric Integer            |
    Boolean Bool               |
    List [WHNF]                |
    Closure [Command] Context  |
    Op String Int [WHNF]
            deriving (Show, Eq)

-- Определения типов основных и вспомогательных функций
---- Компилятор:
compile    :: Expr -> [Command]
comp       :: [[String]] -> Expr -> [Command] -> [Command]
compile   e = comp [] e [Stop]

-- Вычисление адресной пары переменной в заданном контексте
имен
pos       :: String -> [String] -> Int
pos x (y:l) | x == y = 0
            | otherwise = if k == -1 then -1 else 1 + k
      where k = pos x l
pos x [] = -1

addr       :: String -> [[String]] -> (Int, Int)
addr x (lev:list) =
    if j >= 0 then (0, j)
              else (let (p, q) = addr x list in (p+1, q))
                       where j = pos x lev

comp env (Integral n) lc = (LoadConst (Numeric n)) : lc
comp env (Logical b) lc = (LoadConst (Boolean b)) : lc
comp env Nil lc = (LoadConst (List [])) : lc
comp env (Function f) lc = (LoadOp f) : lc
comp env (Variable x) lc = (Load (addr x env)) : lc
comp env (Lambda var e) lc =
         (LoadFunc (comp ([var]:env) e [Return])) : lc
comp env (Application func arg) lc =
         ((comp env arg) . (comp env func)) (Apply:lc)
comp env (If cond th el) lc =
         comp env cond
              ((Select (comp env th []) (comp env el [])):lc)
comp env (Let pairs body) lc =
```

```
        (LoadConst (List [])) :
        (foldr (comp' env)
               ((LoadFunc (comp (names:env) body [Return])):
                LetApply:lc) (reverse values))
    where (names, values) = unzip pairs
          comp' e ex ls   =
                comp e ex ((LoadOp "cons"):Apply:Apply:ls)
comp env (Letrec pairs body) lc =
    Dummy : (LoadConst (List [])) :
    (foldr (comp' env)
        ((LoadFunc (comp (names:env) body [Return])):
                    RecApply:lc) (reverse values))
  where (names, values) = unzip pairs
    comp' e ex ls = comp e ex ((LoadOp "cons"):Apply:Apply:ls)
```

В этой программе основная работа по компиляции выражения выполняется функцией comp, которая для повышения эффективности имеет дополнительный накапливающий параметр – уже сформированный список команд. Это позволяет полностью избавиться от неэффективного соединения списков команд с помощью операции ++, которую мы активно использовали в нашем неформальном описании компилятора.

Теперь перейдем к описанию работы SECD-машины низкого уровня. Ее работу мы тоже будем описывать неформально в терминах последовательного исполнения *переходов*, изменяющих состояние регистров машины. Каждый переход происходит под воздействием одной команды из регистра управления. Последней исполняемой командой будет команда остановки машины – `Stop`.

Практически всю работу SECD-машины низкого уровня тоже можно описать в виде чисто функциональной программы. Исключение составляет команда `RecApply`, которая производит присваивание в контексте – подмену фиктивного пустого списка значений на вычисленный список. Здесь снова существенно, что происходит именно замена значения, а не формирование нового списка, как это происходит в функциональных программах, поскольку этот фиктивный контекст уже мог быть запомнен в замыканиях для лямбда-выражений, встречающихся в заголовке блока. Замена значения должна привести к тому, что все ссылки на этот контекст будут теперь указывать на измененное значение. Фактически исполнение команды `RecApply` приведет к образованию циклического контекста, в котором некоторые элементы содержат ссылку на себя.

Сначала опишем, как исполняются простые команды, загружающие в стек результатов те или иные значения.

```
(s, ctx, (LoadConst val):c, d)  ⇒  (val:s, ctx, c, d)
(s, ctx, (Load addr):c, d)  ⇒  ((getValue addr ctx):s,ctx,c,d)
(s,ctx,(LoadFunc coms):c,d) ⇒  ((Closure coms ctx):s,ctx,c,d)
(s, ctx, (LoadOp f):c, d)   ⇒  ((Op f (arity f) []):s,ctx,c,d)
```

Все эти переходы совершенно понятны – значение из команды или контекста просто переносится на стек результатов, возможно, несколько видоизменяясь. Так, при исполнении команды `LoadFunc` загрузки функции в стек помещается замыкание, образованное из команд, реализующих работу этой функции, и текущего контекста. При исполнении команды загрузки встроенной функции `LoadOp` формируется пустой список аргументов, и с помощью функции `arity` вычисляется количество требуемых аргументов для того, чтобы функция могла выполниться.

Команда `Select` тоже работает достаточно просто, однако результат ее работы зависит от значения, находящегося на вершине стека. Исполнение этой команды мы опишем с помощью двух переходов, соответствующих двум возможным значениям, которые будут находиться на вершине стека в момент ее исполнения.

```
((Boolean True):s,ctx,(Select th el):c,d) ⇒ (s,ctx,th ++ c,d)
((Boolean False):s,ctx,(Select th el):c,d) ⇒ (s,ctx,el ++ c,d)
```

В результате к текущему списку команд регистра управления добавляется один из двух списков команд из исполняющейся команды `Select`. Теперь опишем пару команд для исполнения входа в функцию и выхода из нее – команд `Apply` и `Return`. В случае, когда функция, в которую происходит вход, представлена замыканием, команда `Apply` просто сохраняет текущее состояние SECD-машины в регистре хранения состояний, формирует новый контекст и подготавливает машину к выполнению команд вызываемой функции. Такое поведение можно описать с помощью следующего правила перехода:

```
((Closure coms env):arg:s, ctx, Apply:c, d)  ⇒
                          ([], [arg]:env, coms, (s, ctx, c):d)
```

Работа команды `Apply` в ситуации, когда исполняемая функция представлена сечением зависит от того, является ли аргумент функции последним требуемым аргументом. Если да, то результат вычисляется с помощью функции `intrinsic`, которая, как и раньше, выполняет действия по исполнению встроенной функции. Если нет, то очередной аргумент просто добавляется в список аргументов сечения. Таким образом, работа этой команды может быть описана с помощью следующих двух правил перехода (во втором из этих правил подразумевается, что значение `n` больше единицы).

```
((Op f 1 list):arg:s, ctx, Apply:c, d)  ⇒
                  ((intrinsic f (list ++ [arg])):s, ctx, c, d)
((Op f n list):arg:s, ctx, Apply:c, d)  ⇒
                  ((Op f (n-1) (list ++ [arg])):s, ctx, c, d)
```

Команда выхода из функции просто восстанавливает сохраненное ранее состояние SECD-машины, перенося в стек вычисленное функцией значение.

```
(val:s', ctx', Return:c', (s, ctx, c):d) ⇒ (val:s, ctx, c, d)
```

Команда входа в нерекурсивный блок `LetApply` исполняется практически так же, как и команда входа в функцию `Apply`. Единственное отличие состоит в том, что при ее исполнении уже сформирован список аргументов применяемой функции. Кроме того, можно заметить, что текущий контекст `ctx` обязательно совпадает с контекстом, сохраненным в замыкании. Фактически, этот контекст даже и не надо было бы сохранять.

```
((Closure coms env):(List args):s, ctx, LetApply:c, d)  ⇒
                              ([], args:env, coms, (s, ctx, c):d)
```

Самой сложной является команда входа в рекурсивный блок. При вычислении рекурсивного блока, прежде всего, с помощью команды `Dummy` образуется фиктивный контекст. При исполнении команды `RecApply` этот контекст является текущим, но он же мог быть сохранен во всех замыканиях, образованных при вычислении заголовка блока. Поэтому команда `RecApply` подменяет во всех экземплярах этого контекста первый элемент на список вычисленных значений. Будем считать, что это проделывает специальная функция `replace`, так что в результате вызова `(replace args ([]:ctx))` не просто образуется новый контекст `(args:ctx)`, но фактически то же самое значение получают и все ранее сохраненные экземпляры этого контекста. В остальном работа этой команды похожа на работу команды `LetApply`.

```
(s, ctx, Dummy:c, d)  ⇒  (s, []:ctx, c, d)
((Closure coms ([]:ctx)):(List args):s,[]:ctx,RecApply:c,d) ⇒
            ([], (replace args ([]:ctx)), coms, (s, ctx, c):d)
```

Для полноты изложения осталось привести только правило для исполнения команды остановки `Stop`. Формально его можно записать следующим образом:

```
(s, ctx, Stop:c, d)  ⇒  (s, ctx, c, d)
```

Приведем два примера работы нашей SECD-машины. В первом примере определяется лямбда-выражение, представляющее функцию удвоения аргумента. Это лямбда-выражение применяется к аргументу 2. Конечно, в результате должно получиться значение 4. Исходная программа может быть записана в расширенном лямбда-исчислении следующим образом:

```
(λn.+ n n) 2
```

Эта программа может быть представлена в виде следующего значения типа `Expr`:

```
(Application
   (Lambda "n" (Application
                  (Application (Function "+")
                               (Variable "n"))
                  (Variable "n")))
   (Integral 2))
```

При компиляции этой программы в последовательность команд SECD-машины получится следующая последовательность команд SECD-машины:

```
[LoadConst (Numeric 2),
 LoadFunc [Load (0,0),
           Load (0,0),
           LoadOp "+",
           Apply, Apply,
           Return],
 Apply,
 Stop]
```

Поместим эту последовательность команд в регистр управления SECD-машины и инициируем ее работу. Последовательность состояний проиллюстрируем с помощью следующей таблицы:

Таблица 1. Последовательность состояний SECD-машины при исполнении команд

s	**e**	**c**	**d**
[]	[]	[LoadConst (Numeric 2), LoadFunc [Load (0,0), Load (0,0), LoadOp "+", Apply, Apply, Return], Apply, Stop]	[]
[(Numeric 2)]	[]	[LoadFunc [Load (0,0), Load (0,0), LoadOp "+", Apply, Apply, Return], Apply, Stop]	[]
[(Closure [Load (0,0), Load (0,0), LoadOp "+", Apply, Apply, Return] []), (Numeric 2)]	[]	[Apply, Stop]	[]
[]	[[(Numeric 2)]]	[Load (0,0), Load (0,0), LoadOp "+", Apply, Apply, Return]	[([], [], [Stop])]
[(Numeric 2)]	[[(Numeric 2)]]	[Load (0,0), LoadOp "+", Apply, Apply, Return]	[([], [], [Stop])]
[(Numeric 2), (Numeric 2)]	[[(Numeric 2)]]	[LoadOp "+", Apply, Apply, Return]	[([], [], [Stop])]
[(Op "+", 2, []), (Numeric 2), (Numeric 2)]	[[(Numeric 2)]]	[Apply, Apply, Return]	[([], [], [Stop])]
[(Op "+", 1, [(Numeric 2)]), (Numeric 2)]	[[(Numeric 2)]]	[Apply, Return]	[([], [], [Stop])]
[(Numeric 4)]	[[(Numeric 2)]]	[Return]	[([], [], [Stop])]
[(Numeric 4)]	[]	[Stop]	[]
[(Numeric 4)]	[]	[]	[]

Теперь проиллюстрируем работу SECD-машины в случае, когда программа содержит рекурсивную функцию. Пусть исходная программа имеет следующий вид:

```
letrec sum =
      λn.if (= n 0) then 0 else (+ n (sum (- n 1))) in (sum 2)
```

Эта программа представляет собой рекурсивный блок, в котором определена простая функция суммирования натуральных чисел от единицы до заданного значения аргумента. Функция определена рекурсивно и вызывается в теле блока с аргументом 2. В результате должно получиться число 3, поскольку 1+2=3. Представление этой программы в виде выражения более сложно, а получившаяся в результате трансляции программа получается более длинной. Вот как может выглядеть исходное выражение:

```
(Letrec [("sum", (Lambda "n"
                    (If (Application
                            (Application
                                (Function "=")
                                (Variable "n"))
                            (Integral 0))
                        (Integral 0)
                        (Application
                            (Application
                                (Function "+")
                                (Variable "n"))
                            (Application
                                (Variable "sum")
                                (Application
                                    (Application
                                        (Function "-")
                                        (Variable "n"))
                                    (Integral 1)))))))]
        (Application (Variable "sum") (Integral 2)))
```

После его компиляции получится следующая последовательность команд:

```
[Dummy, LoadConst (List []),
 LoadFunc [LoadConst (Numeric 0), Load (0,0), LoadOp "=",
Apply, Apply,
         Select [LoadConst (Numeric 0)]
                [LoadConst (Numeric 1), Load (0,0),
                 LoadOp "-", Apply, Apply, Load (1,0),
                 Apply, Load (0,0), LoadOp "+", Apply, Apply],
           Return],
 LoadOp "cons", Apply, Apply,
 LoadFunc [LoadConst (Numeric 2), Load (0,0), Apply, Return],
 RecApply, Stop]
```

Мы снова приведем таблицу, в которой будет показано, как эти команды исполняются при работе SECD-машины, однако, чтобы не загромождать таблицу постоянно повторяющимися фрагментами одних и тех же команд, введем следующие обозначения.

Обозначим идентификатором `Then` последовательность из единственной команды `[LoadConst (Numeric 0)]`, представляющей первую составную часть команды `Select`; вторую составную часть этой команды - последовательность команд `[LoadConst (Numeric 1), Load (0,0), LoadOp "-", Apply, Apply, Load (1,0), Apply, Load (0,0), LoadOp "+", Apply, Apply]`, обозначим идентификатором `Else`; далее, обозначим идентификатором `Sum` последовательность команд `[LoadConst (Numeric 0), Load (0,0), LoadOp "=", Apply, Apply, Select Then Else, Return]`, составляющих тело функции `sum` исходной программы; наконец, обозначим идентификатором `Body` последовательность команд `[LoadConst (Numeric 2), Load (0,0), Apply, Return]`, полученных компиляцией тела рекурсивного блока. Тогда вся последовательность команд, полученная компиляцией исходной программы, может быть записана следующим образом:

```
[Dummy, LoadConst (List []), LoadFunc Sum, LoadOp "cons",
 Apply, Apply, LoadFunc Body, RecApply, Stop].
```

Кроме того, если содержимое некоторой клетки в таблице в точности повторяет содержимое вышележащей клетки, то в клетке будем писать обозначение -"-. Получившаяся таблица, демонстрирующая последовательность исполнения этих команд, все равно получается достаточно длинной, и, может быть, не имеет смысл подробно изучать исполнение каждой из исполняющихся команд, однако, следует рассмотреть подробно команды, имеющие отношение к наиболее тонкому фрагменту исполнения – командам формирования фиктивного контекста, а также командам замены части этого фиктивного контекста на вычисленный новый контекст при входе в рекурсивный блок. Так, например, в результате исполнения команды `RecApply` будет образован контекст, содержащий ссылку на себя. В таблице такой контекст будет обозначен идентификатором `ctx`, а внутри контекста этот идентификатор будет обозначать циклическую ссылку на себя.

Таблица 2. Последовательность состояний SECD-машины при исполнении более сложной программы

s	**e**	**c**	**d**
[]	[]	[Dummy, LoadConst (List []), LoadFunc Sum, LoadOp "cons", Apply, Apply, LoadFunc Body, RecApply, Stop]	[]

[]	[[]]	[LoadConst (List []), LoadFunc Sum, LoadOp "cons", Apply, Apply, LoadFunc Body, RecApply, Stop]	[]
[List []]	[[]]	[LoadFunc Sum, LoadOp "cons", Apply, Apply, LoadFunc Body, RecApply, Stop]	[]
[(Closure Sum [[]]), List []]	[[]]	[LoadOp "cons", Apply, Apply, LoadFunc Body, RecApply, Stop]	[]
[(Op "cons" 2 []), (Closure Sum [[]]), List []]	[[]]	[Apply, Apply, LoadFunc Body, RecApply, Stop]	[]
[(Op "cons" 1 [(Closure Sum [[]])]), List []]	[[]]	[Apply, LoadFunc Body, RecApply, Stop]	[]
[List [(Closure Sum [[]])]]	[[]]	[LoadFunc Body, RecApply, Stop]	[]
[(Closure Body [[]]), List [(Closure Sum [[]])]]	[[]]	[RecApply, Stop]	[]
[]	ctx:[[(Closure Sum ctx)]]	Body:[LoadConst (Numeric 2), Load (0,0), Apply, Return]	[([], [], [Stop])]
[(Numeric 2)]	-"-	[Load (0,0), Apply, Return]	-"-
[(Closure Sum ctx), (Numeric 2)]	-"-	[Apply, Return]	-"-
[]	[[(Numeric 2)], [(Closure Sum ctx)]]	Sum:[LoadConst (Numeric 0), Load (0,0), LoadOp "=", Apply, Apply, Select Then Else, Return]	[([], ctx, [Return]), ([], [], [Stop])]
[(Numeric 0)]	-"-	[Load (0,0), LoadOp "=", Apply, Apply, Select Then Else, Return]	-"-
[(Numeric 2), (Numeric 0)]	-"-	[LoadOp "=", Apply, Apply, Select Then Else, Return]	-"-
[(Op "=" 2 []), (Numeric 2), (Numeric 0)]	-"-	[Apply, Apply, Select Then Else, Return]	-"-
[(Op "=" 1 [(Numeric 2)]), (Numeric 0)]	-"-	[Apply, Select Then Else, Return]	-"-
[(Boolean False)]	-"-	[Select Then Else, Return]	-"-
[]	-"-	Else:[LoadConst (Numeric 1), Load (0,0), LoadOp "-", Apply, Apply, Load (1,0), Apply, Load (0,0), LoadOp "+", Apply, Apply] ++ [Return]	-"-
[(Numeric 1)]	-"-	[Load (0,0), LoadOp "-", Apply, Apply, Load (1,0), Apply, Load (0,0), LoadOp "+", Apply, Apply, Return]	-"-
[(Numeric 2), (Numeric 1)]	-"-	[LoadOp "-", Apply, Apply, Load (1,0), Apply, Load (0,0), LoadOp "+", Apply, Apply, Return]	-"-
[(Op "-" 2 []), (Numeric 2),	-"-	[Apply, Apply, Load (1,0), Apply, Load (0,0), LoadOp "+", Apply, Apply,	-"-

(Numeric 1)]		Return]	
[(Op "-" 1 [(Numeric 2)]), (Numeric 1)]	-"-	[Apply, Load (1,0), Apply, Load (0,0), LoadOp "+", Apply, Apply, Return]	-"-
[(Numeric 1)]	-"-	[Load (1,0), Apply, Load (0,0), LoadOp "+", Apply, Apply, Return]	-"-
[(Closure Sum ctx), (Numeric 1)]	-"-	[Apply, Load (0,0), LoadOp "+", Apply, Apply, Return]	-"-
[]	[[(Numeric 1)], [(Closure Sum ctx)]]	Sum:[LoadConst (Numeric 0), Load (0,0), LoadOp "=", Apply, Apply, Select Then Else, Return]	[([], [[(Numeric 2)], [(Closure Sum ctx)]], [Load (0,0), LoadOp "+", Apply, Apply, Return]), ([], ctx, [Return]), ([], [], [Stop])]
[(Numeric 0)]	-"-	[Load (0,0), LoadOp "=", Apply, Apply, Select Then Else, Return]	-"-
[(Numeric 1), (Numeric 0)]	-"-	[LoadOp "=", Apply, Apply, Select Then Else, Return]	-"-
[(Op "=" 2 []), (Numeric 1), (Numeric 0)]	-"-	[Apply, Apply, Select Then Else, Return]	-"-
[(Op "=" 1 [(Numeric 1)]), (Numeric 0)]	-"-	[Apply, Select Then Else, Return]	-"-
[(Boolean False)]	-"-	[Select Then Else, Return]	-"-
[]	-"-	Else:[LoadConst (Numeric 1), Load (0,0), LoadOp "-", Apply, Apply, Load (1,0), Apply, Load (0,0), LoadOp "+", Apply, Apply] ++ [Return]	-"-
[(Numeric 1)]	-"-	[Load (0,0), LoadOp "-", Apply, Apply, Load (1,0), Apply, Load (0,0), LoadOp "+", Apply, Apply, Return]	-"-
[(Numeric 1), (Numeric 1)]	-"-	[LoadOp "-", Apply, Apply, Load (1,0), Apply, Load (0,0), LoadOp "+", Apply, Apply, Return]	-"-
[(Op "-" 2 []), (Numeric 1), (Numeric 1)]	-"-	[Apply, Apply, Load (1,0), Apply, Load (0,0), LoadOp "+", Apply, Apply, Return]	-"-
[(Op "-" 1 [(Numeric 1)]), (Numeric 1)]	-"-	[Apply, Load (1,0), Apply, Load (0,0), LoadOp "+", Apply, Apply, Return]	-"-
[(Numeric 0)]	-"-	[Load (1,0), Apply, Load (0,0), LoadOp "+", Apply, Apply, Return]	-"-
[(Closure Sum ctx), (Numeric 0)]	-"-	[Apply, Load (0,0), LoadOp "+", Apply, Apply, Return]	-"-
[]	[[(Numeric 0)],	Sum:[LoadConst (Numeric 0), Load (0,0),	[([],

	[(Closure Sum ctx)]]	LoadOp "=", Apply, Apply, Select Then Else, Return]	[[(Numeric 1)], [(Closure Sum ctx)]], [Load (0,0), LoadOp "+", Apply, Apply, Return]), ([], [[(Numeric 2)], [(Closure Sum ctx)]], [Load (0,0), LoadOp "+", Apply, Apply, Return]), ([], ctx, [Return]), ([], [], [Stop])]
[(Numeric 0)]	-"-	[Load (0,0), LoadOp "=", Apply, Apply, Select Then Else, Return]	-"-
[(Numeric 0), (Numeric 0)]	-"-	[LoadOp "=", Apply, Apply, Select Then Else, Return]	-"-
[(Op "=" 2 []), (Numeric 0), (Numeric 0)]	[[(Numeric 0)], [(Closure Sum ctx)]]	[Apply, Apply, Select Then Else, Return]	[([], [[(Numeric 1)], [(Closure Sum ctx)]], [Load (0,0), LoadOp "+", Apply, Apply, Return]), ([], [[(Numeric 2)], [(Closure Sum ctx)]], [Load (0,0), LoadOp "+", Apply, Apply, Return]), ([], ctx, [Return]), ([], [], [Stop])]
[(Op "=" 1 [(Numeric 0)]), (Numeric 0)]	-"-	[Apply, Select Then Else, Return]	-"-
[(Boolean True)]	-"-	[Select Then Else, Return]	-"-
[]	-"-	[Then: LoadConst (Numeric 0), Return]	-"-
[(Numeric 0)]	-"-	[Return]	-"-
[(Numeric 0)]	[[(Numeric 1)], [(Closure Sum ctx)]]	[Load (0,0), LoadOp "+", Apply, Apply, Return]	[([], [[(Numeric 2)], [(Closure Sum ctx)]], [Load (0,0), LoadOp "+", Apply, Apply, Return]), ([], ctx, [Return]), ([], [], [Stop])]

[(Numeric 1), (Numeric 0)]	-"-	[LoadOp "+", Apply, Apply, Return]	-"-
[(Op "+" 2 []), (Numeric 1), (Numeric 0)]	-"-	[Apply, Apply, Return]	-"-
[(Op "+" 1 [(Numeric 1)]), (Numeric 0)]	-"-	[Apply, Return]	-"-
[(Numeric 1)]	-"-	[Return]	-"-
[(Numeric 1)]	[[(Numeric 2)], [(Closure Sum ctx)]]	[Load (0,0), LoadOp "+", Apply, Apply, Return]	[([], ctx, [Return]), ([], [], [Stop])]
[(Numeric 2), (Numeric 1)]	-"-	[LoadOp "+", Apply, Apply, Return]	-"-
[(Op "+" 2 []), (Numeric 2), (Numeric 1)]	-"-	[Apply, Apply, Return]	-"-
[(Op "+" 1 [(Numeric 2)]), (Numeric 1)]	-"-	[Apply, Return]	-"-
[(Numeric 3)]	-"-	[Return]	-"-
[(Numeric 3)]	ctx:[[(Closure Sum ctx)]]	[Return]	[([], [], [Stop])]
[(Numeric 3)]	[]	[Stop]	[]
[(Numeric 3)]	[]	[]	[]

Итак, мы рассмотрели различные способы реализации функциональных программ, в том числе построение по заданной функциональной программе ее эквивалента в некотором императивном языке программирования (языке, построенном на последовательном исполнении команд), исполняемом последовательной SECD-машиной. Тем самым мы установили, что функциональные языки программирования выразимы с помощью обычных средств последовательного исполнения. На самом деле, произвольную последовательно исполняющуюся программу тоже можно выразить с помощью эквивалентной ей функциональной программы, и в следующем разделе мы займемся построением функциональной программы по заданной программе, написанной на привычном императивном языке программирования.

4.4. Функциональные эквиваленты императивных программ

В качестве императивного языка программирования построим простой язык программирования с *Java*-подобным синтаксисом, который содержит лишь небольшое число конструкций и способов представления данных. В нашем языке будут только простые переменные целого типа и операции над ними, будут присваивания и простые управляющие конструкции – условные операторы и операторы цикла, но все более сложные конструкции (массивы, функции, классы и т.п.) мы рассматривать

не будем. На самом деле принципиальных сложностей в рассмотрении этих конструкций нет. Труднее всего поддаются выражению в функциональном виде конструкции структурных выходов из циклов и механизм обработки исключений. Поскольку у нас будут переменные только одного типа, то нам не понадобятся описания переменных, таким образом, вся программа будет представлять собой один исполняемый блок.

Поскольку нашей целью является представление работы последовательной программы в виде функции, то глобальное поведение такой программы должно быть «функциональным», то есть задача каждой программы сводится к вычислению некоторых результирующих значений по заданным в начале работы исходным данным. Поэтому в наших программах не будет также процедур ввода/вывода, вместо этого мы будем считать, что перед началом работы задано "начальное состояние" переменных, используемых в программе, а в конце работы результат можно прочитать, исследуя заключительное состояние переменных. Например, программу вычисления факториала заданного целого числа можно будет представить в следующем виде:

```
f = 1;
while (n > 0) {
    f = f * n;
    n = n - 1;
}
```

Предполагается, что если задано начальное значение переменной `n`, то результат работы можно будет прочитать, исследуя переменную `f`. Итак, рассмотрим все конструкции используемого нами языка.

Во-первых, переменные и целые числа будут использоваться для построения *выражений*. Точных правил для выражений мы задавать не будем, однако, будем считать, что выражения составляются с помощью унарных и бинарных операций над значениями и скобок таким образом, чтобы всегда можно было определить, как и в какой последовательности надо выполнять операции, чтобы получить требуемый результат. В программах не будут использоваться явно логические значения и логические переменные, однако, выражения с логическим результатом все же будут рассматриваться как часть управляющих конструкций для организации ветвления и повторения (циклов). Мы будем полагать, что целое значение, равное нулю, будет представлять также логическое значение «ложь», а значение, равное единице – «истину». Теперь можно написать синтаксические правила для построения выражений приблизительно следующим образом:

выражение → *переменная* | *константа* | *унарная-операция выражение* | *выражение бинарная-операция выражение*

Данный синтаксис не задает однозначных правил построения и анализа выражений, однако, мы будем считать, что в любом случае мы

сможем разобраться, в каком порядке должны выполняться те или иные операции.

Теперь зададим правила, определяющие структуру операторов программы.

оператор → *пустой-оператор* | *присваивание* | `{` *последовательность* `}` | *оператор-выбора* | *цикл*

пустой-оператор → `;`

присваивание → *переменная* = *выражение* `;`

последовательность → *оператор* | *оператор последовательность*

оператор-выбора → `if (` *выражение* `)` *оператор* `else` *оператор* | `if (` *выражение* `)` *оператор*

цикл → `while (` *выражение* `)` *оператор*

Для того чтобы определить функцию, эквивалентную любой программе, проще всего написать программу перевода (компилятор), которая будет получать исходную программу, написанную на нашем императивном языке программирования, в качестве аргумента, и будет выдавать функцию в качестве результата. Для того, чтобы сделать это, нам понадобится, во-первых, определить, как представлена исходная программа, а во-вторых, определить, какую функцию мы будем считать эквивалентной нашей исходной программе.

Первый вопрос мы решим так же, как мы это уже делали для функциональных программ: определим новый тип данных, значениями которого будут конструкции программ императивного языка программирования. Два основных типа, которые при этом удобно определить, – это тип `Expression` для представления выражений и тип `Operator` для представления операторов исходной программы. Конструкторы значений этих типов буду представлять все способы задания выражений и операторов программы.

```
data Expression = Variable String |
                  Constant Integer |
                  Unary String Expression |
                  Binary Expression String Expression

data Operator = Skip |
                Assignment String Expression |
                Sequence [Operator] |
                Select Expression Operator Operator |
                Loop Expression Operator
```

По сравнению с исходным определением языка в приведенных определениях типов `Expression` и `Operator` сделано несколько небольших изменений. Во-первых, операторы, задающие последовательность других, более простых, операторов представлены списком, в то время как в исходном определении последовательность – это

пара из одного простого оператора и последовательности всех остальных операторов. Во-вторых, в операторе выбора всегда присутствует часть `else` (понятно, что случай ее отсутствия можно легко выразить с помощью пустого оператора в части `else`). В остальном определения типов `Expression` и `Operator` просто повторяют синтаксические правила, задающие наш язык. Мы будем предполагать, что исходная программа сразу задана в виде выражения типа `Operator`, то есть уже выполнен синтаксический анализ программы, анализатор убедился в ее синтаксической правильности и построил соответствующее выражение. Так, например, приведенная в начале раздела программа вычисления факториала заданного числа после обработки и построения соответствующего выражения будет выглядеть так:

```
Sequence [(Assignment "f" (Constant 1)),
          (Loop (Binary (Variable "n") ">" (Constant 0))
                (Sequence [(Assignment "f"
                  (Binary (Variable "f") "*" (Variable "n"))),
                           (Assignment "n"
                  (Binary (Variable "n") "-" (Constant 1)))]
         ))]
```

Теперь надо решить второй вопрос – договориться о том, что представляет собой исполнение программы, как задать исходные данные для нее, и как интерпретировать полученные результаты ее исполнения.

При последовательном исполнении операторов программы меняются значения составляющих контекст программы переменных. Можно считать, что с самого начала вся доступная программе память представлена набором программных переменных, таким образом, каждый шаг при исполнении программы, вообще говоря, приводит к изменению значений этих переменных – мы будем говорить об изменении *состояния среды*. Весь эффект исполнения программы, таким образом, состоит в изменении состояния среды от некоторого начального состояния, содержащего в том числе исходные данные для работы программы, до конечного состояния, содержащего результаты. В начальном состоянии нас интересуют только значения переменных, заданных в качестве исходных данных (в примере с факториалом – значение переменной `n`); в конечном состоянии, напротив, нас интересуют значения переменных, представляющих результат работы программы (в примере с факториалом – значение переменной `f`). Если, как и раньше, представлять контекст переменных списком пар из имен и значений переменных,

```
type Environment = [(String, Expression)]
```

то начальное состояние среды при запуске программы вычисления факториала числа 5 может быть представлено в виде

```
env_initial = [("f", (Constant 0)), ("n", (Constant 5))]
```

(предполагается, что начальное значение переменной `f` равно нулю). После исполнения программы значение переменной `n` уменьшится до нуля, а значением переменной `f` окажется вычисленное значение факториала – 120. Таким образом, конечное состояние среды можно будет описать так:

```
env-final = [("f", (Constant 120)), ("n", (Constant 0))]
```

Исполнение программы может быть задано функцией, преобразующей начальное состояние среды `env-initial` в конечное состояние – `env-final`. Таким образом, наша задача состоит в том, чтобы по заданной программе (выражению типа `Operator`) построить функцию, аргументом и результатом которой служит среда (выражение типа `Environment`). Мы, однако, начнем с более простой программы-интерпретатора, которая будет имитировать последовательное исполнение операторов программы. Такой интерпретатор получает в качестве аргументов программу и начальное состояние среды, а в результате работы выдает конечное состояние среды. Тип функции-интерпретатора можно задать с помощью следующего описания типа на языке *Haskell*:

```
interpreter :: Operator -> Environment -> Environment
```

Заметим, что если рассматривать эту функцию не как функцию двух аргументов, которая по программе и начальному состоянию переменных выдает конечное состояние переменных после выполнения программы, а как функцию одного аргумента (свойство карринга!), то в результате как раз и получится нужная нам функция преобразования программы в эквивалентную ей функцию, переводящую начальное состояние переменных в конечное.

Для определения искомой функции сначала требуется написать функцию, вычисляющую значение выражения в заданной среде. Такую функцию мы уже писали неоднократно, так что здесь можно просто повторить определение такой функции почти без дополнительных пояснений. Функция должна преобразовать сложное выражение, содержащее переменные, константы и знаки операций в простое выражение-константу. Как и раньше будем предполагать, что значение всех переменных задано в среде и может быть найдено с помощью функции

```
assoc :: String -> Environment -> Expression
```

Определение функций `assoc` и `evaluate` следует ниже.

```
evaluate :: Expression -> Environment -> Expression

assoc x ((y, e):tail) | x == y     = e
                      | otherwise  = assoc x tail
evaluate (Variable x) env      = assoc x env
evaluate e@(Constant n) _      = e
evaluate (Unary op expr) env   =
```

```
          intrinsic op [evaluate expr env]
evaluate (Binary e1 op e2) env =
          intrinsic op [(evaluate e1 env), (evaluate e2 env)]
```

В приведенном определении предполагается, что функция `intrinsic` выполняет встроенные функции, заданные в унарной или бинарной операции, над константами. Так, например, если считать, что встроенными являются арифметические бинарные операции `+`, `-`, `*`, `/` и `%` (последние две операции представляют целочисленное деление и операцию получения остатка от целочисленного деления), а также унарная операция арифметического отрицания `neg`, то уравнения, определяющие результат выполнения арифметических операций, будут выглядеть следующим образом:

```
intrinsic "+" [(Constant a), (Constant b)] = Constant (a + b)
intrinsic "-" [(Constant a), (Constant b)] = Constant (a - b)
intrinsic "*" [(Constant a), (Constant b)] = Constant (a * b)
intrinsic "/" [(Constant a), (Constant b)] =
                                        Constant (a `div` b)
intrinsic "%" [(Constant a), (Constant b)] =
                                        Constant (a `mod` b)
intrinsic "neg" [(Constant a)] = Constant (0 - a)
```

Поскольку требуется определить еще и операции с логическим результатом, то можно точно также добавить несколько уравнений, определяющих результат применения функций сравнения и логических операций к константам:

```
intrinsic ">" [(Constant a), (Constant b)] =
                        Constant (if a > b then 1 else 0)
intrinsic "<" [(Constant a), (Constant b)] =
                        Constant (if a < b then 1 else 0)
intrinsic "=" [(Constant a), (Constant b)] =
                        Constant (if a == b then 1 else 0)
intrinsic "and" [(Constant a), (Constant b)] =
                        Constant (if a + b == 2 then 1 else 0)
intrinsic "or" [(Constant a), (Constant b)] =
                        Constant (if a + b >= 1 then 1 else 0)
```

Разумеется, аналогичным образом можно определить и другие операции.

Теперь уже можно выписать и определяющие уравнения основной функции `interpret`. Эта функция просто следует правилам исполнения программы, заданным неформально для каждого из возможных типов операторов программы.

```
interpreter Skip env = env
interpreter (Assignment x e) env =
                  replace x (evaluate e env) env
interpreter (Sequence []) env = env
interpreter (Sequence (s1:others)) env =
    interpreter (Sequence others) (interpreter s1 env)
```

```
interpreter (Select cond s1 s2) env =
    interpreter iff env
    where iff =
      (if (evaluate cond env) == (Constant 1) then s1 else s2)
interpreter oper@(Loop cond s) env =
    if (evaluate cond env) == (Constant 0)
       then env
       else interpreter oper (interpreter s env)
```

Эти уравнения необходимо объяснить. Первое уравнение, впрочем, очевидно – оно отражает тот факт, что выполнение пустого оператора не изменяет среду. Второе уравнение использует функцию `replace`, которая заменяет в заданной среде значение одной переменной на заданное значение. Точнее, функция должна построить новую среду, отличающуюся от исходной значением только одной переменной. Вот как может выглядеть определение этой функции.

```
replace :: String -> Expression -> Environment -> Environment
replace x v ((first@(y, _)):others)
             | x == y    = (y, v) : others
             | otherwise = first:(replace x v others)
```

Третье и четвертое уравнения функции `interpret` задают семантику выполнения последовательности операторов следующим образом. Последовательность из нулевого числа операторов не меняет среду, а если последовательность не пуста, то надо сначала выполнить первый из операторов, а оставшиеся выполнить в среде, которая получится после исполнения первого оператора.

Пятое уравнение предписывает для исполнения условного оператора сначала вычислить значение условия, и в зависимости от результата исполнять первый или второй из внутренних операторов.

Самое сложное – последнее уравнение, описывающее семантику выполнения оператора цикла. В нем предписывается вычислить условие цикла в исходном состоянии, а затем, в зависимости от результата либо оставить среду неизменной (исполнение цикла завершено), либо исполнить тело цикла в исходной среде, а затем повторить исполнение того же оператора в новой среде, полученной после однократного выполнения тела цикла

Можно проверить, что наш интерпретатор действительно работает. Если вызвать функцию `interpreter` с параметрами, задающими нашу исходную программу вычисления факториала и с начальной средой `env_initial`, то результатом вычисления будет новая среда

```
env_final = [("f", (Constant 120)), ("n", (Constant 0))]
```

В данном случае, правда, важно не столько то, что мы можем с помощью функции `interpreter` получать результаты исполнения программы, сколько то, что мы можем с ее помощью по любой программе получать эквивалентную этой программе функцию. Ранее, в этой главе мы

также установили, что по каждой функции мы можем построить эквивалентную ей императивную программу. В совокупности с только что полученным результатом это означает, что выразительная сила у функционального и императивного программирования одинакова, то есть средства программирования в обоих случаях эквивалентны. Правда, в последнем разделе мы рассматривали только очень простые программы, так что осталось не вполне ясным, можно ли будет построить функциональную программу по более сложной императивной программе, такой, которая будет содержать не только циклы, присваивания и условные операторы, но также функции, обработку исключительных ситуаций, сложные структуры данных и другие конструкции.

На самом деле, эквивалентность функционального и императивного программирования установлена и для этих, более сложных, случаев, хотя доказательство этого довольно сложно. Самым сложным оказывается установление функционального эквивалента для конструкций структурных переходов и обработки исключительных ситуаций. Для выражения понятия перехода были даже введены специальные средства в функциональные языки - функции с продолжением.

На этом мы заканчиваем наш обзор функциональных средств программирования.

Примеры решения задач

Во всех примерах и заданиях для самостоятельного решения в этом разделе исследуются программы, написанные на некотором императивном языке программирования, содержащем переменные и константы целого типа, присваивания, условные операторы и последовательности операторов. Программы представлениы с помощью значений типа `Operator`, а промежуточные выражения представлены значениями типа `Expression`. Определения этих типов использовались нами в главе 4.4, однако по сравнению с основным текстом пособия определения этих типов незначительно упрощены (убраны циклы).

```
data Expression = Variable String |
                  Constant Integer |
                  Unary String Expression |
                  Binary Expression String Expression

data Operator = Skip |
                Assignment String Expression |
                Sequence [Operator] |
                Select Expression Operator Operator
```

Задание 1. Напишите функцию, получающую список всех переменных программы, которые не встречаются в левых частях присваиваний.

Решение. Для решения задачи необходимо из списка всех использующихся в программе переменных удалить те из них, которые используются в левых частях операторов присваивания. Элементарными операциями здесь будут добавление переменной (строки) в список и вычисление разности списков (как множеств строк). Подобные функции уже не раз были представлены в этом пособии. Для решения этой задачи будем использовать упорядоченные списки строк.

Листинг У4.3. Список неинициализированных переменных.

```
data Expression = Variable String |
                  Constant Integer |
                  Unary String Expression |
                  Binary Expression String Expression

data Operator = Skip |
                Assignment String Expression |
                Sequence [Operator] |
                Select Expression Operator Operator

-- Добавление строки в список.
add :: String -> [String] -> [String]
add s [] = [s]
add s list@(head:tail) | s < head  = s:list
                       | s == head = list
                       | otherwise = head:(add s tail)

-- Вычисление разности множеств, представленных упорядоченными
-- списками.
diff :: [String] -> [String] -> [String]
diff list1 [] = list1
diff [] list2 = []
diff l1@(h1:t1) l2@(h2:t2) | h1 < h2 = h1:(diff t1 l2)
                           | h1 > h2 = diff l1 t2
                           | otherwise = diff t1 t2

-- Основная функция для решения задачи.
-- Аргумент: анализируемый оператор (программа).
-- Результат: список переменных программы, не встречающихся
--            в левых частях операторов присваивания.
setvars :: Operator -> [String]
setvars op = diff right left
                   where (left, right) = evalsets op ([], [])

-- Вспомогательная функция, выполняющая всю основную работу.
-- Имеет накапливающий аргумент для повышения эффективности.
-- Аргументы: исследуемый оператор;
--            пара множеств переменных, встретившихся ранее.
-- Результат: дополненная переменными оператора пара множеств.
evalsets :: Operator -> ([String], [String]) ->
```

```
                        ([String], [String])
evalsets Skip pair = pair
evalsets (Assignment var expr) (left, right) =
        (add var left, addvars expr right)
evalsets (Select expr op1 op2) (left, right) =
        evalsets op1 (evalsets op2 (left, addvars expr right))
evalsets (Sequence list) pair = foldr evalsets pair list

addvars :: Expression -> [String] -> [String]
addvars (Variable v) = add v
addvars (Unary _ e) = addvars e
addvars (Binary e1 _ e2) = (addvars e1) . (addvars e2)
```

Задание 2. Напишите функцию, удаляющую из программы все «несущественные» пустые операторы. Пустой оператор будет несущественным, если он является элементом последовательности операторов (`Sequence`). Заметим также, что пустая последовательность операторов эквивалентна пустому оператору и должна трактоваться так же.

Решение. Для решения задачи напишем функцию, которая преобразует любой оператор в «каноническую» форму. Будем считать, что оператор находится в канонической форме, если в нем нет операторов `Skip` внутри оператора `Sequence` и нет пустых последовательностей операторов. Собственно, решение задачи и состоит в приведении оператора к канонической форме.

Вообще говоря, решение довольно прямолинейно, единственная сложность состоит в том, что уже после того, как из последовательности операторов убраны все пустые операторы, последовательность сама может оказаться пустой, и в этом случае ее тоже необходимо преобразовать в оператор `Skip`. Эту последнюю проверку выполняет в программе функция `check`.

Листинг У4.4. Приведение программы к каноническому виду.

```
data Expression = Variable String |
                  Constant Integer |
                  Unary String Expression |
                  Binary Expression String Expression
                      deriving (Eq, Show)

data Operator = Skip |
                Assignment String Expression |
                Sequence [Operator] |
                Select Expression Operator Operator
                    deriving (Eq, Show)

-- Основная функция для решения задачи.
-- Аргумент: программа (оператор);
-- Результат: программа, приведенная к каноническому виду.
```

```
toCanonical :: Operator -> Operator
toCanonical (Sequence list) =
    check (Sequence (filter (/= Skip) (map toCanonical list)))
toCanonical (Select expr op1 op2) =
    Select expr (toCanonical op1) (toCanonical op2)
toCanonical op = op

-- Вспомогательная функция для последней проверки оператора.
-- Превращает пустую последовательность операторов в Skip.
check :: Operator -> Operator
check (Sequence []) = Skip
check op = op
```

Задания для самостоятельной работы

Задание 1. Напишите функцию, которая выдает список всех переменных программы, которым в программе делаются присваивания, но которые не используются ни в одном из выражений в программе.

Задание 2. Напишите функцию, которая выводит программу в виде, удобном для чтения (то есть когда присваивания имеют вид `v = e;`, оператор выбора имеет вид `if (condition) operator; else operator;` и т.д.).

Задание 3. Напишите функцию, которая меняет в программе все переменные с заданным именем на переменные с другим именем.

Задание 4. Напишите функцию, которая удаляет из программы все присваивания переменным, если эти переменные не используются в других операторах программы. Учтите, что если переменная используется как в левой, так и в правой части одного и того же оператора присваивания, то такое использование переменной «не считается».

Задание 5. Напишите функцию, которая находит в программе все переменные, которые встречаются одновременно в левых и правых частях одних и тех же операторов присваивания.

Приложение 1. Система функционального программирования WinHugs

Для решения учебных задач по программированию на языке *Haskell* в рамках операционной системы *Windows* (имеются версии и для других операционных систем) рекомендуется установить систему программирования *WinHugs*, которую можно скопировать и установить с сайта http://www.haskell.org/hugs. Система программирования *WinHugs* является некоммерческой свободно распространяемой системой программирования на языке *Haskell*. Последняя известная на момент написания настоящего текста версия датирована сентябрем 2006 года. Для установки системы необходимо запустить установочный пакет *WinHugs-Sep2006.exe* и следовать указаниям запущенного приложения. После установки системы запуск программы осуществляется обычным способом.

Программная оболочка системы *WinHugs* представляет собой визуализированную среду программирования для редактирования текстов программ на языке *Haskell* и запуска интерпретатора (см. рисунок 1.1). Поддерживается версия языка *Haskell*, называемая *Hugs'98*.

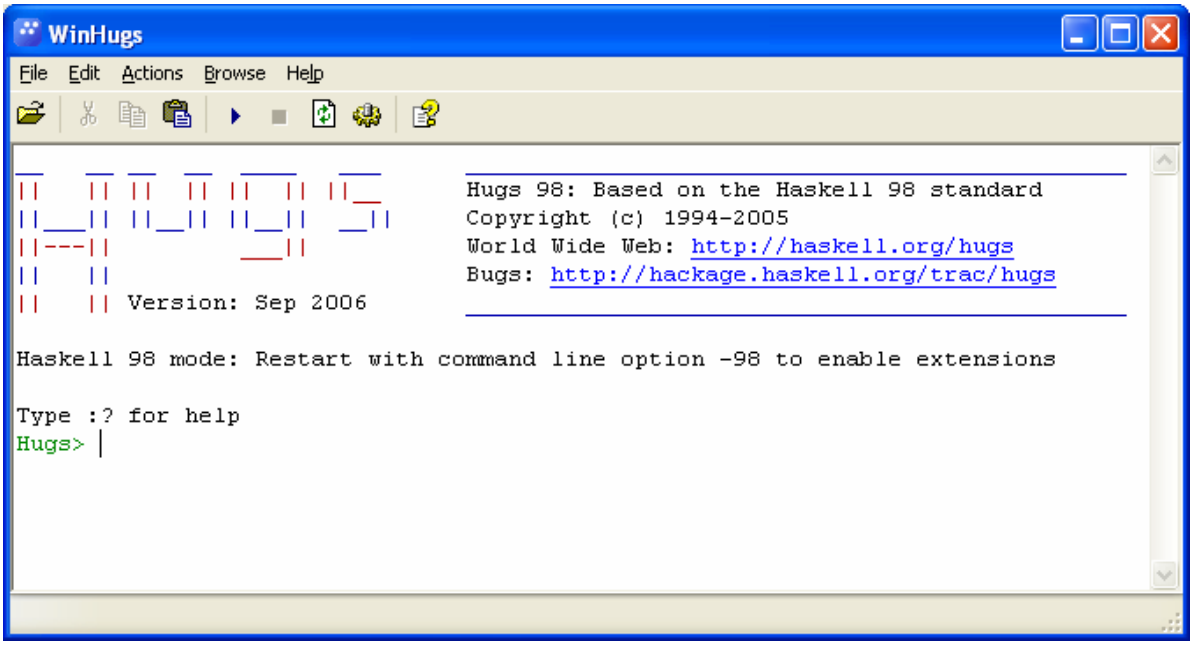

Рис. П1.1. Окно работающей системы WinHugs

Работа в системе состоит из сессий, включающих в себя создание и редактирование файлов программы, отладку и запуск тестовых примеров. Далее приводятся рекомендации по работе в системе.

П1.1. Работа в системе WinHugs

Окно работающего приложения представляет собой командную среду для выполнения команд по редактированию и запуску программ. В

это же окно выводятся результаты работы программы. В дальнейшем все примеры исполнения команд будут приводится в следующем формате:

```
mod> команда
mod> результат
```

где `команда` - это одна из управляющих команд или команда запуска одной из функций программы (вычисления значения выражения), а `результат` - сообщение системы о результате выполнения управляющей команды или вывод результирующего значения вычисленного выражения. В командной строке перед знаком > система выводит имя текущего загруженного модуля или слово `Hugs` в случае безымянного модуля. Так, при старте системы содержимое командного окна обычно имеет вид

```
Hugs>
```

В заголовке окна приложения находится также меню для быстрого исполнения наиболее употребительных управляющих команд и инструментальные кнопки, предназначенные для той же цели.

Прежде всего, установим режимы, при которых система ведет подсчет затраченных на выполнение программы ресурсов памяти и времени и выводит тип вычисленного выражения. Это позволит нам иметь максимально полную информацию об исполнении программ и эффективности их работы. Для установки режимов следует исполнить команду меню `File/Options` и на вкладке `Runtime` отметить пункты `Print statistics` и `Print type after evaluation` (см. рисунок П1.2).

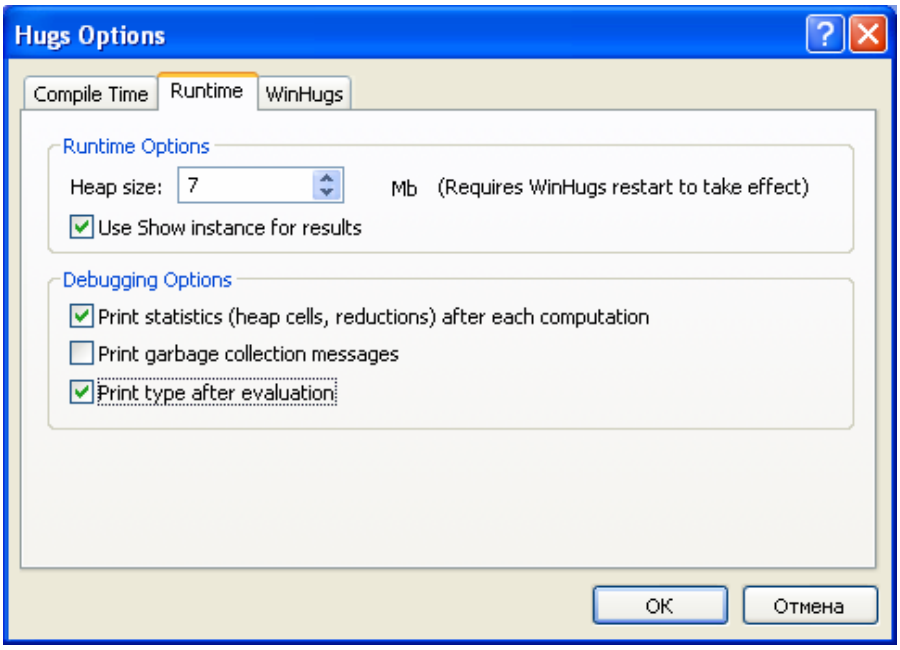

Рис. П1.2. Окно установки режимов

Вывод статистики исполнения происходит в единицах интерпретатора системы *WinHugs*. Время исполнения приводится в количестве выполненных редукций – преобразований исходного

выражения (reductions), а затраченная память оценивается в количестве отведенных ячеек (cells). Следует заметить, что определенное количество ресурсов тратится системой на организацию работы самого интерпретатора, так что оценка времени работы программы и затраченной для ее выполнения памяти на самом деле носит приближенный характер. Кроме того, при всех запусках программы, кроме первого, система может использовать уже вычисленные при первом запуске значения, так что время работы программы при втором и последующих запусках может оказаться меньше, чем при первом запуске программы. Для получения достоверных результатов следует после завершения отладки программы перезапустить систему *WinHugs*, загрузить и один раз выполнить отлаженную программу. Не следует «зацикливать» программу для «усреднения» результата, так как при повторных вычислениях время работы окажется существенно меньше, чем при первом запуске.

Итак, закроем окно установки режимов и введем первое выражение, значение которого требуется подсчитать, скажем

```
> 2 + 2
```

и нажмем клавишу Enter. Нормально работающая система должна выдать что-нибудь похожее на следующий текст (см. также рисунок 1.3):

```
4 :: Integer
(30 reductions, 65 cells)
Hugs>
```

Это означает, что система вычислила результат вычисления введенного выражения (4), указала тип вычисленного выражения (Integer), и затратила при этом 30 единиц времени (сделала 30 редукций) и 65 ячеек памяти. Конечно, для такой простой программы приведенный цифры кажутся чересчур большими, однако следует учесть, что помимо собственно вычислений потребовалось, по крайней мере, преобразовать полученный результат в строку и вывести его в командную строку.

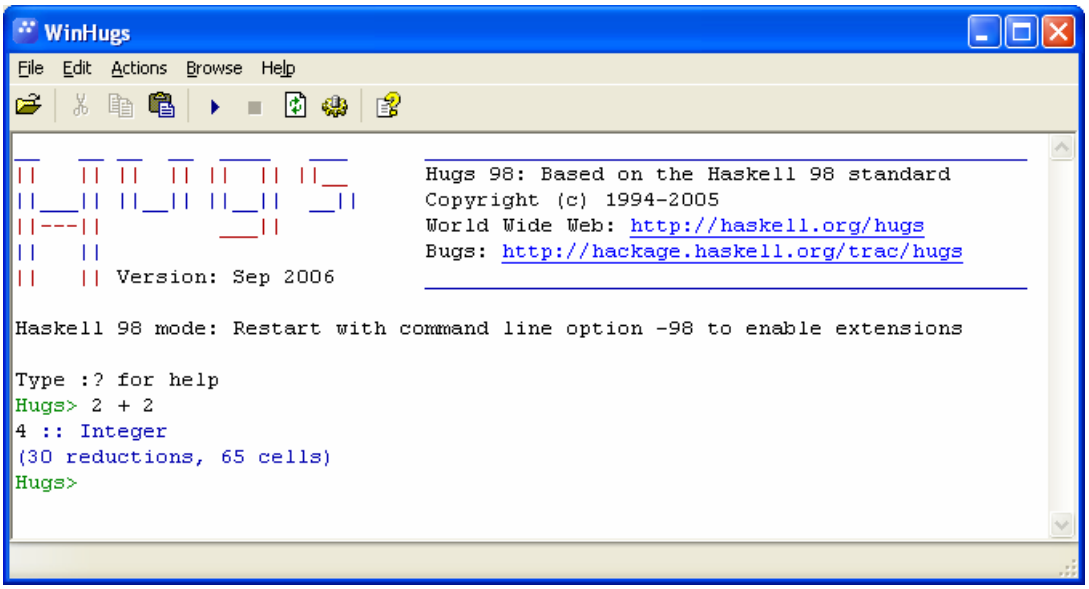

Рис. П1.3. Пример работы системы WinHugs

Теперь подготовим простую программу, запишем ее в текстовый файл и запустим средствами системы.

Прежде всего, надо установить, с помощью какого текстового редактора будет происходить подготовка текста. При установке системы программирования *WinHugs* редактором «по умолчанию» считается стандартный системный текстовый редактор *Notepad* (*Блокнот*). Однако, при желании можно заменить редактор, установив соответствующий режим на вкладке *WinHugs* диалога режимов, который можно вызвать командой `File/Options`. Этот редактор будет вызываться всякий раз, когда вы захотите из интегрированной среды системы программирования *Haskell* изменить текст загруженного модуля или программы. Окно установки режимов показано на рисунке 1.4.

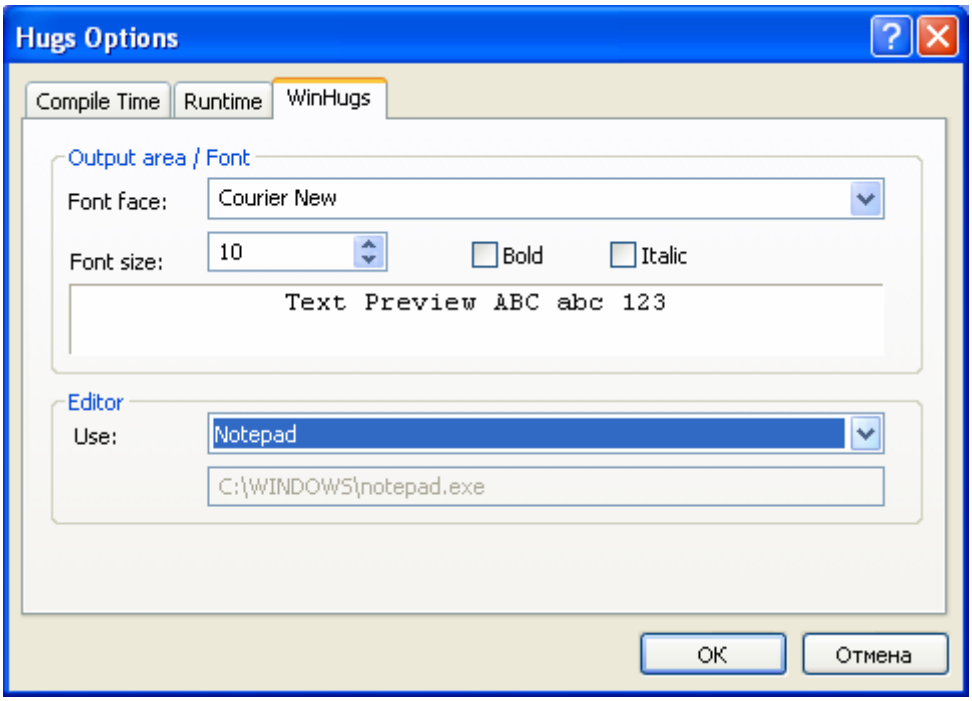

Рис. П1.4. Установка режимов редактирования текста программ

Первоначально текст программы набирается независимо от интегрированный среды *WinHugs* и записывается в некоторый файл. Стандартным типом файла считается *Haskell Script*, и стандартным расширением для этого типа будет расширение *.hs*.

Итак, подготовим определение модуля, включающего в себя определения некоторых простых функций, скажем, функции, которая вычисляет дискриминант квадратного уравнения по его коэффициентам и функции, которая вычисляет значения корней квадратного уравнения по заданным коэффициентам. Будем считать, что в случае отрицательного значения дискриминанта программа должна заканчиваться аварийно при попытке вычислить корни квадратного уравнения. Текст программы может иметь приблизительно такой вид, как показано в листинге П1.1.

Листинг П1.1. Нахождение вещественных корней квадратного уравнения

```
-- Программный модуль для решения квадратного уравнения
module SqRoot where

-- дискриминант квадратного уравнения
discr :: Double -> Double -> Double -> Double
discr a b c = b*b - 4*a*c

-- Сообщение об ошибке.
message :: String
message = "No real roots found"

-- Решение квадратного уравнения.
-- В случае отсутствия вещественных корней выводится сообщение
об ошибке.
```

```
sqroot :: Double -> Double -> Double -> (Double, Double)
sqroot a b c | d >= 0 =
                  ((-b + sqrt(d))/(2*a), (-b - sqrt(d))/(2*a))
             | otherwise = error message
                  where d = discr a b c
```

Откроем редактор, скажем, тот же *Блокнот*, запишем приведенный текст в окно этого редактора и сохраним файл на диск, скажем, под именем *C:\Haskell\Equation.hs*. Заметим, что система программирования *WinHugs* ни в коей мере не локализована (по крайней мере, на момент написания настоящего текста), поэтому следует избегать использования русских букв как в названиях файлов, так и в именах типов, функций и т.п. Можно, впрочем, использовать символы кириллицы для записи комментариев и в качестве содержимого строк, однако при выводе таких строк система может преобразовывать неизвестные ей символы (буквы кириллицы) в их представление в виде числовых кодов.

Теперь, в среде программирования *WinHugs* загрузим программу из подготовленного файла. Для этого можно либо набрать команду загрузки файла непосредственно в командном окне, либо воспользоваться командой меню `File/Open...` В первом случае следует в командном окне набрать системную команду

```
:load "C:\Haskell\Equation.hs"
```

Заметим, что все системные команды начинаются с символа `":"` и могут быть сокращены, так что команда может выглядеть и так:

```
:l "C:\Haskell\Equation.hs"
```

Не надо забывать про двоеточие – это признак системной команды, если его нет, то система воспринимает строку как запись выражения, подлежащего вычислению, так что, например, команда

```
l "C:\Haskell\Equation.hs"
```

Будет воспринята как команда для вычисления выражения, в котором функция `l` применяется (вызывается с аргументом) к строке `"C:\Haskell\Equation.hs"`.

На рисунке П1.5 показано, как можно воспользоваться элементом меню для открытия файла.

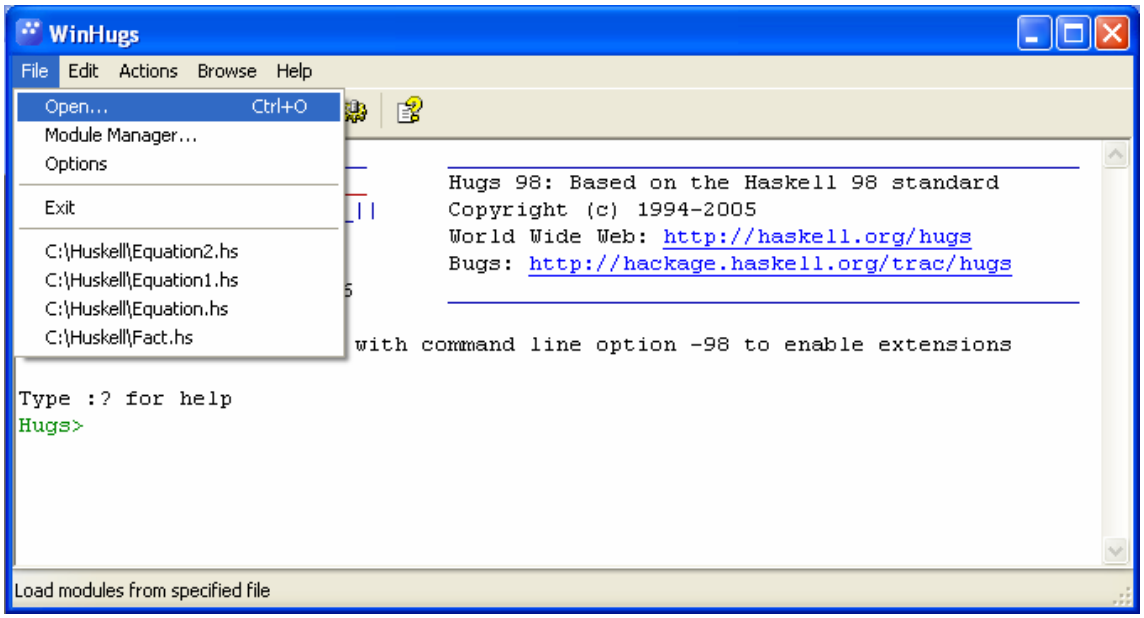

Рис. П1.5. Открытие файла с текстом программы

После загрузки файла программный модуль, содержащийся в нем, сразу же компилируется, и если компилятор обнаружит ошибку, то эта ошибка будет выдана на экран сразу же после загрузки файла. Если ошибок нет, то система просто выдаст приглашение к вводу очередной команды. В нашем случае мы можем вычислить значение корней квадратного уравнения с помощью функции `sqroot`. Например, если мы хотим вычислить корни уравнения $x^2-5x+4=0$, то следует написать выражение

```
sqroot 1 (-5) 4
```

в результате чего система выдаст результат вычисления:

```
(4.0,1.0) :: (Double,Double)
(86 reductions, 188 cells)
```

то есть найдены два вещественных корня – `4.0` и `1.0`, для вычисления выполнено 86 элементарных операций преобразования выражения (редукций), при этом затрачено 188 ячеек памяти для размещения промежуточных данных.

Заметим, что отрицательный аргумент функции заключен в скобки. Это не случайно, при попытке выдать системе команду

```
sqroot 1 -5 4
```

немедленно будет выдано сообщение об ошибке

```
ERROR - Cannot infer instance
*** Instance   : Num (a -> Double -> Double -> (Double,Double))
*** Expression : sqroot 1 - fromInt 5 4
```

Понять, что означает это сообщение довольно трудно. Фактически дело заключается в том, что вместо выражения `sqroot 1 (-5) 4` система пытается вычислить выражение `(sqroot 1) - (5 4)`, а это, конечно, бессмысленно. Вообще, надо быть внимательным в расстановке

скобок: самой приоритетной операцией в выражениях считается применение функции, которое обозначается просто пробелом. Символ минуса в нашем ошибочном выражении – это операция более низкого приоритета, поэтому «по умолчанию» скобки и расставлены таким причудливым на первый взгляд образом.

Если мы попробуем вычислить корни уравнения, которое на самом деле не имеет вещественных корней, скажем, уравнения $x^2+x+1=0$, то на запрос

```
sqroot 1 1 1
```

система выдаст сообщение

```
Program error: No real roots found
```

как результат вызова функции `error`. На рисунке П1.6 показано, как выглядит диалог в том случае, когда вещественных корней нет.

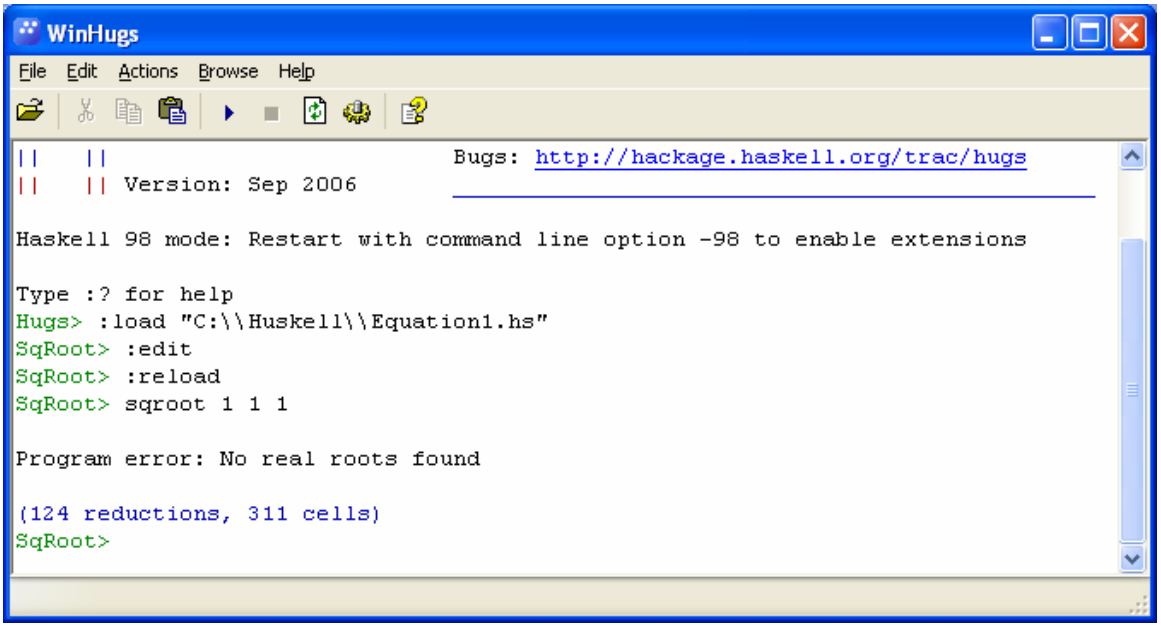

Рис. П1.6. Результат работы системы в случае отсутствия решения

Давайте теперь внесем исправление в нашу программу. Пусть теперь система в случае отсутствия вещественных корней будет выдавать вещественную и мнимую части комплексных корней. Для различения этих двух случаев будем выдавать еще один результат – строку сообщения, в которой указывается тип выданного результата: вещественные корни или вещественная и мнимая части комплексных корней. Соответствующая программа должна выглядеть, скажем, так, как показано в листинге П1.2. По сравнению с предыдущим случаем, в программу внесены еще два изменения.

Во-первых, в заголовке основного модуля указано, что из модуля экспортируется (доступна для использования) только одна функция – `sqroot`. Это означает, что при использовании этого модуля из других

программных модулей только эта функция будет доступна для вызова, остальные функции являются «скрытыми».

Во-вторых, в модуле определена «главная» функция `main`, которая вызывается системной командой запуска программы. В нашей главной программе функция `sqroot` вызывается несколько раз для проверки работы программы с разными тестовыми данными.

Листинг П1.2. Решение квадратного уравнения в общем случае

```
-- Программный модуль для решения квадратного уравнения
module SqRoot(sqroot) where

-- дискриминант квадратного уравнения
discr :: Double -> Double -> Double -> Double
discr a b c = b*b - 4*a*c

-- решение квадратного уравнения
sqroot :: Double -> Double -> Double -> (String, Double,
Double)
sqroot a b c | d >= 0    = ("real", r1+r2, r1-r2)
             | otherwise = ("complex", r1, r2)
                 where d = discr a b c
                       r1 = (-b)/(2*a)
                       r2 = sqrt(abs(d))/(2*a)

-- тестовая функция
main = [sqroot 1 1 1,
        sqroot 1 (-5) 4,
        sqroot 1 (-2) 1,
        sqroot 0 2 5]
```

На рисунке П1.7 показан результат работы программы. В последнем тесте первый коэффициент квадратного уравнения равен нулю, что приводит к ошибке деления на ноль при вычислении по нашим формулам. Обратите внимание на то, что система в этом случае получает «неопределенный» результат, однако, программа успешно дорабатывает до конца. На рисунке П1.7 выделена кнопка запуска основной функции загруженного модуля. Соответствующая системная команда называется

```
:main
```

Можно также выполнить ту же команду нажатием функциональной кнопки `F5` на клавиатуре.

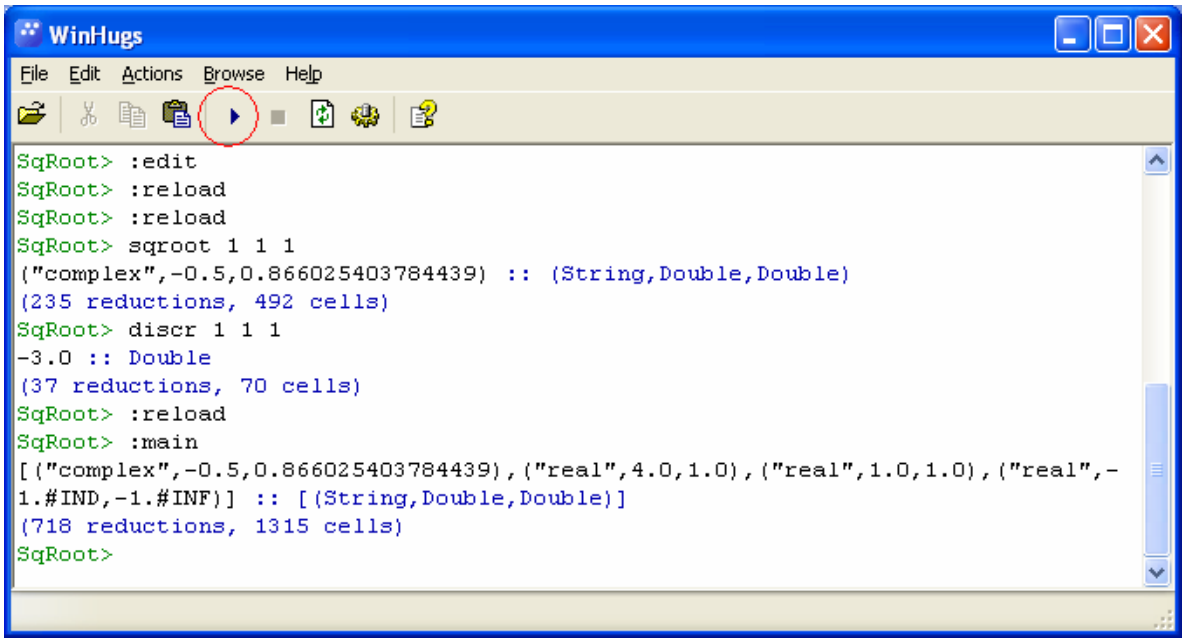

Рис. П1.7. Запуск главной программы с тестовыми данными

Редактор, установленный в системе в качестве основного текстового редактора, может быть вызван непосредственно из командного окна с помощью команды меню `Actions / Open Text Editor` или системной команды `:edit`, сокращенно, `:e`. После того, как вы закончите редактирование и сохраните исправленный текст программы, модуль автоматически перезагружается. Впрочем, всегда можно перезагрузить модули и с помощью явно заданной системной команды `:reload` (`:r`) или нажатием кнопки  .

Итак, основные действия по созданию, редактированию и запуску программ наиболее экономным способом выполняются из главного окна системы следующим образом:

Вызывается текстовый редактор системной командой `:e`. Текст программы вводится средствами этого редактора. Система в течение сессии «помнит» имя последнего редактируемого файла, так что, если Вы работаете с одной и той программой, можно не держать постоянно открытым окно редактирования программы.

Программа сохраняется на диске в файле с именем, не содержащим символов кириллицы.

Программный модуль загружается в систему командой `:l` с указанием имени файла (удобнее использовать кнопку открытия файла) или, в случае, если файл с этим именем уже был загружен ранее, командой повторной загрузки `:r`.

Основная функция main запускается командой `:m` или кнопкой запуска программы. Можно также вычислить выражение, введя его в командной строке.

При необходимости отредактировать текст программы редактор текущего программного модуля запускается командой `:e`. После сохранения текста модуль необходимо перезагрузить командой `:r`.

Еще одно небольшое замечание. При отладке программ часто приходится исполнять одни и те же команды и вычислять одни и те же выражения. Система хранит информацию о последних введенных командах, и их можно вызвать на экран, нажимая клавиши «стрелка вверх» и «стрелка вниз». Вызванную команду можно отредактировать и запустить обычным образом, нажав клавишу `Enter`.

Приведенных сведений вполне достаточно для подготовки, редактирования и запуска простейших программ. Рассмотрим теперь, как создаются и редактируются многомодульные проекты и как происходит управление модулями.

Каждый модуль в программе имеет имя и содержит набор функций. Некоторые функции несут смысловую нагрузку только в пределах данного модуля и не экспортируются в другие модули. Другие функции могут использоваться всеми модулями. Набор таких функций называется списком экспорта данного модуля.

В заголовке модуля указывается его имя (имя модуля должно начинаться с большой буквы) и список экпорта, таким образом, заголовок модуля имеет следующий вид:

```
module MyModuleName (export1, export2, ...) where
```

и далее следуют определения функций констант и т.п. Если список экспорта отсутствует, то считается, что экспортируются все идентификаторы, определенные в модуле.

В задаче о вычислении корней квадратного уравнения мы определяли имя модуля `SqRoot`, и в качестве списка экспорта указывали единственную функцию `sqroot`. Тем не менее, поскольку загружаемый модуль был «главным», то мы могли вызывать любые функции из этого модуля, в частности, функцию `main`, а также и любые другие функции, например, `discr`. В многомодульных программах, однако, всем другим модулям будет доступна единственная функция модуля `SqRoot` – функция `sqroot`.

Определим программу из двух модулей. В первом из них определим две функции, вычисляющие разложение натурального числа на простые множители. Одна функция – `factorize` – будет вычислять список всех простых сомножителей числа, при этом список будет содержать каждый сомножитель столько раз, сколько раз он входит в разложение числа на простые множители. Вторая функция – `dividers` – будет также определять список простых сомножителей числа, но каждое число будет входить в него по одному разу.

Второй модуль будет содержать единственную функцию, которая вызывает определенные в первом модуле функции, передавая им тестовые аргументы.

Программы для функций `factorize` и `dividers` основаны на том, что наименьший делитель натурального числа всегда является простым и, следовательно, входит в разложение числа на простые множители. После того, как простой сомножитель найден, исходное число можно разделить на него и продолжить поиск сомножителей. В случае функции `factorize` поиск продолжается с того же самого числа, а в случае функции `dividers` число делится на найденный сомножитель, пока это возможно, а поиск следующего сомножителя начинается со следующего числа.

Полный текст модуля приведен в листинге П1.3.

Листинг П1.3. Факторизация натурального числа

```
{-------------------------------------------------------------
 Модуль для решения задач факторизации числа
 -------------------------------------------------------------}
module Factorize(factorize, dividers) where

-------------------------------------------------------------
-- Функция dividers находит все простые сомножители заданного
-- числа и формирует список, в котором эти сомножители входят
-- по одному разу.
-------------------------------------------------------------
dividers :: Integer -> [Integer]

-------------------------------------------------------------
-- Функция factorize находит все простые сомножители
-- заданного числа и формирует список, в котором эти
-- сомножители входят столько раз, сколько их имеется в
-- разложении числа на простые множители.
-------------------------------------------------------------
factorize :: Integer -> [Integer]

-------------------------------------------------------------
-- Вспомогательные функции имеют дополнительные аргументы
-------------------------------------------------------------
dividers' :: Integer -> Integer -> [Integer] -> [Integer]
factorize' :: Integer -> Integer -> [Integer] -> [Integer]
-------------------------------------------------------------
-- Вспомогательная функция multidiv осуществляет
-- многократное деление первого аргумента на второй
-- до тех пор, пока деление возможно.
-------------------------------------------------------------
multidiv :: Integer -> Integer -> Integer
-------------------------------------------------------------
n `multidiv` i | n `mod` i == 0  = (n `div` i) `multidiv` i
```

```
              | otherwise       = n

dividers n
   | n `mod` 2 == 0  = dividers' 3 (n `multidiv` 2) [2]
   | otherwise       = dividers' 3 n []

dividers' i n lst
   | n == 1  = lst
   | i*i > n = n:lst
   | n `mod` i == 0 = dividers' (i+2) (n `multidiv` i) (i:lst)
   | otherwise      = dividers' (i+2) n lst

factorize n | n `mod` 2 == 0  = 2:factorize (n `div` 2)
            | otherwise       = factorize' 3 n []

factorize' i n lst
   | n == 1 = lst
   | i*i > n = n:lst
   | n `mod` i == 0 = factorize' i (n `div` i) (i:lst)
   | otherwise = factorize' (i+2) n lst
```

Теперь определим второй модуль, содержащий только вызов тестовых примеров. В этом втором модуле необходимо указать «список импорта» – имена тех модулей, функции из которых используются в данном модуле. В нашем случае список импорта будет выглядеть следующим образом:

```
import Factorize
```

Итак, текст второго модуля приводится в листинге П1.4.

Листинг П1.4. Модуль запуска тестовых примеров

```
{--
 Основная функция данного модуля осуществляет
 тестирование функций разложения числа
 на простые сомножители из модуля Factorize
--}
module Main where

import Factorize  -- импортируются factorize и dividers

-- тестовые примеры
main = [(factorize 24, dividers 24),
        (factorize 255, dividers 255),
        (factorize 2, dividers 2),
        (factorize 37, dividers 37),
        (factorize 1, dividers 1)
       ]
```

Теперь, когда оба модуля написаны и сформированы два программных файла, сформируем многомодульную программу. Для этого, прежде всего, запустим менеджер модулей с помощью команды

`File / Module Manager...`. Окно управления модулями показано на рисунке П1.8.

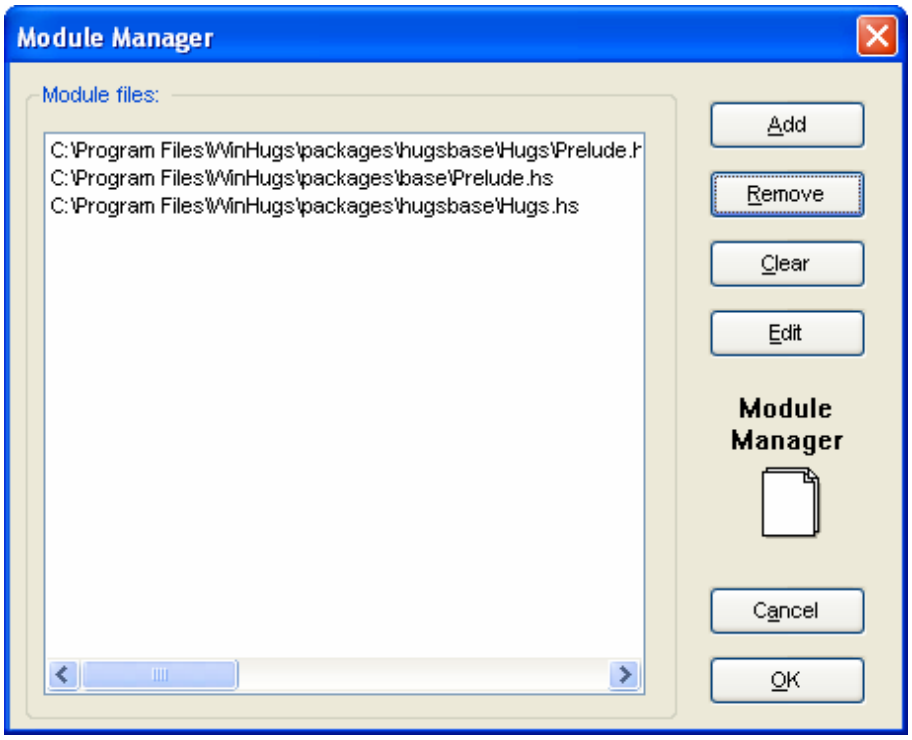

Рис. П1.8. Окно управления модулями

В начале работы программы загружены три основных модуля, первые два из которых представляют собой стандартную библиотеку типов и функций системы *WinHugs*. Не удаляйте эти модули! Третий загруженный модуль – пустой, это «заглушка» для написания своих текстов. Этот модуль имеет имя `Hugs`. Нажатием кнопки `Add` мы можем добавить два своих модуля в загрузку. Если два наших модуля имеют имена файлов `C:\Haskell\Factorize.hs` и `C:\Haskell\main.hs` соответственно, то после загрузки окно управления модулями приобретет такой вид, как показано на рисунке П1.9.

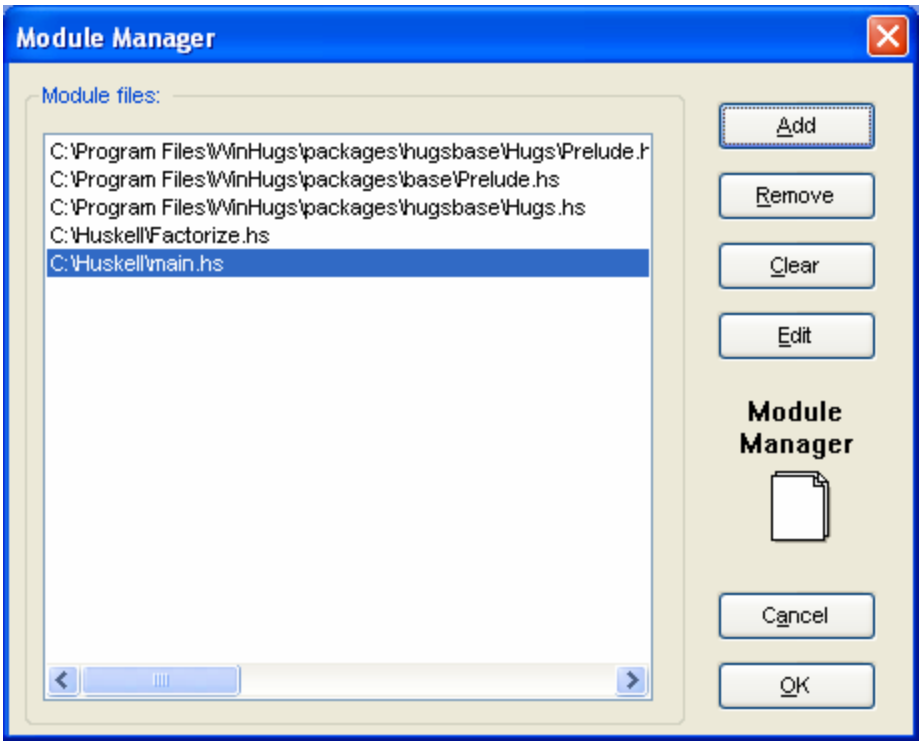

Рис. П1.9. Загрузка нескольких модулей

Теперь можно закрыть окно и запустить готовую программу. Результат запуска программы показан на рисунке П1.10.

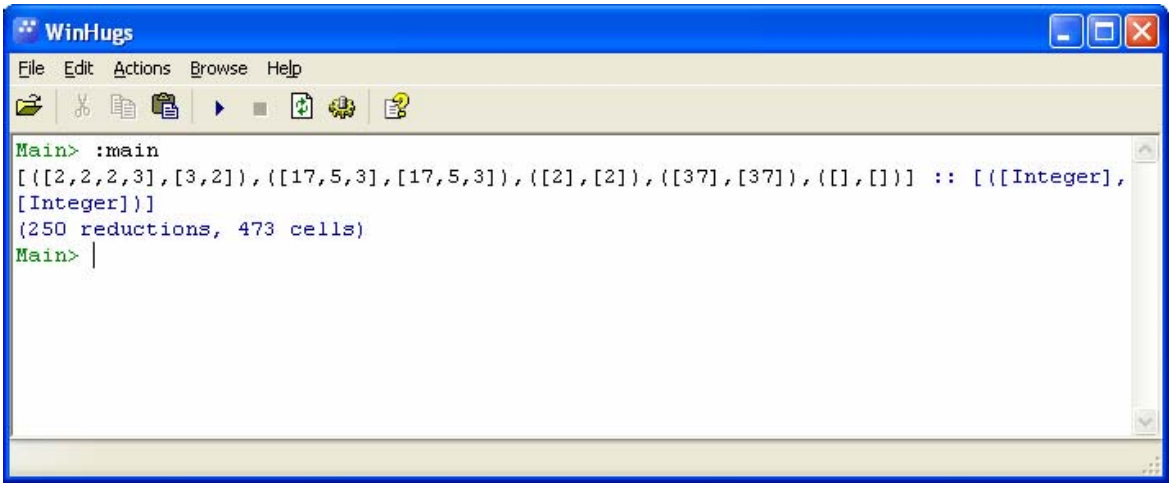

Рис. П1.10. Результат запуска многомодульной программы

Порядок загрузки модулей не важен. Система сама построит зависимости между модулями и найдет «главный» модуль.

Давайте изменим наш основной программный модуль `Main` и переопределим в нем функцию `dividers` таким образом, чтобы воспользоваться результатами функции `factorize`. Действительно, для того, чтобы получить список всех делителей числа по одному разу, можно просто удалить дублирующие вхождения элементов в списке, который является результатом работы функции `factorize`. Это можно сделать с помощью вспомогательной функции `remDuplicates`, которая

пользуется тем, что одинаковые значения в списке всегда расположены рядом. Определение этой функции и переопределение функции `dividers`, сделанное в модуле `Main`, показано в листинге П1.5.

Листинг П1.5. Переопределение функций в одном из модулей

```
{--
 Основная функция данного модуля осуществляет
 тестирование функций разложения числа
 на простые сомножители из модуля Factorize
--}
module Main where

import Factorize  -- импортируются factorize и dividers

-- функция remDuplicates удаляет дублирующие
-- рядом расположенные элементы в списке
remDuplicates :: Eq a => [a] -> [a]
remDuplicates [] = []
remDuplicates lst@[x] = lst
remDuplicates (x:lst2@(y:lst))
                        | x == y    = remDuplicates lst2
                        | otherwise = x:remDuplicates lst2

dividers = remDuplicates . factorize

-- тестовые примеры
main = [(factorize 24, dividers 24),
        (factorize 255, dividers 255),
        (factorize 2, dividers 2),
        (factorize 37, dividers 37),
        (factorize 1, dividers 1)
       ]
```

Однако при попытке запустить исправленную программу система зафиксирует ошибку, так как система не может определить, какой из двух вариантов функции `dividers` вызывается в функции `main`. Ошибка будет обнаружена на этапе компиляции модулей и будет выглядеть так, как показано на рисунке П1.11.

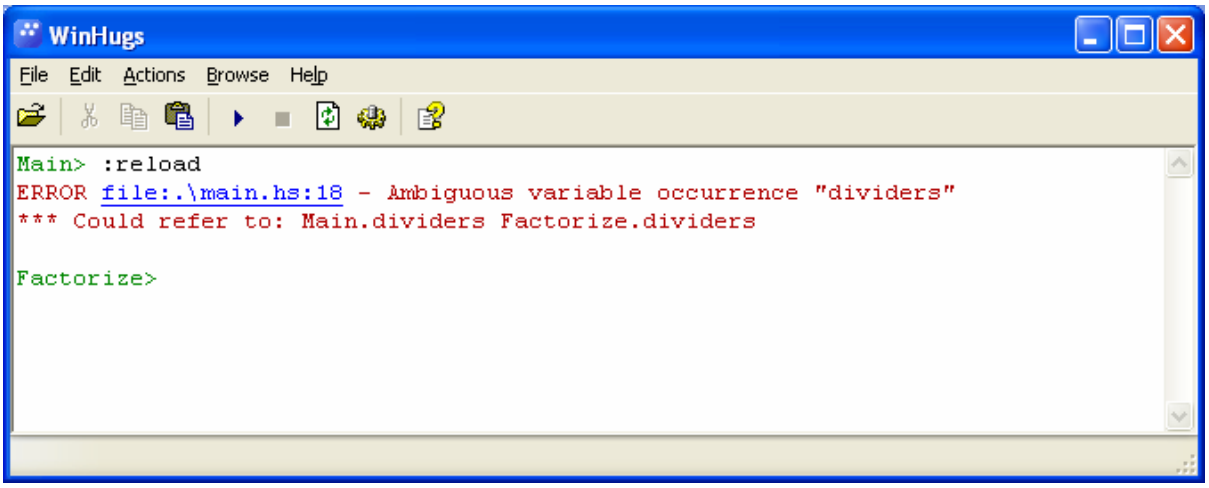

Рис. П1.11. Конфликт имен функций в двух модулях

В данном случае конфликт имен можно разрешить двумя различными способами. Во-первых, можно явно указать, каким из двух вариантов функции надо пользоваться. Вот как, например, может выглядеть определение функции main, в котором вызываются обе версии функции dividers.

```
main =
  [(factorize 24, Main.dividers 24, Factorize.dividers 24),
   (factorize 255, Main.dividers 255, Factorize.dividers 255),
   (factorize 2, Main.dividers 2, Factorize.dividers 2),
   (factorize 37, Main.dividers 37, Factorize.dividers 37),
   (factorize 1, Main.dividers 1, Factorize.dividers 1)
  ]
```

Во-вторых, можно импортировать из модуля `Factorize` только функцию `factorize`, указав явно в списке импорта, какие функции импортируются. Тогда функция `dividers`, определенная в модуле `Factorize` будет недоступна, и конфликт также будет разрешен. Список импорта в этом случае будет выглядеть так:

```
import Factorize(factorize)  -- импортируется только factorize
```

Отметим, что при использовании имен, квалифицированных именем модуля, недопустимо окружать символ точки пробелами, то есть составное имя в нашем примере может выглядеть как `Main.dividers`, но не может выглядеть как `Main. dividers`, `Main .dividers` или `Main . dividers`. Это связано с тем, что символ точки имеет в языке и другое назначение – это операция образования суперпозиции функций. Эта операция, кстати, была нами использована при определении функции `dividers` в модуле `Main`. В этом случае, напротив, рекомендуется окружать знак операции (впрочем, как и знаки других операций) пробелами.

Рассмотрим теперь программу, состоящую из трех модулей. В первом модуле определяются некоторые функции обработки строк, в

частности, функция, которая определяет, состоят ли две заданные строки из одних и тех же букв. Во втором модуле эта функция используется для того, чтобы определить все анаграммы в заданном наборе слов, то есть все подмножества слов, состоящих из одних и тех же букв. Третий модуль задает главную тестирующую функцию.

Функция проверки того, состоят ли два слова из одних и тех же букв, может просто отсортировать буквы в каждом из двух заданных слов и после этого проверить, равны ли получившиеся отсортированные списки букв. Сортировку будем производить простейшим методом простых вставок, так что текст модуля будет иметь вид, показанный в листинге П1.6.

Листинг П1.6. Модуль некоторых функций над строками

```
{-------------------------------------------------------------
 Модуль, определяющий некоторые функции над строками
 -------------------------------------------------------------}
module Strings (anagrams, split) where

-------------------------------------------------------------
-- сортировка списка простыми вставками
-------------------------------------------------------------
-- основная функция insSort осуществляет сортировку списка.
-- Аргумент: исходный список;
-- Результат: отсортированный список
-------------------------------------------------------------
insSort :: Ord a => [a] -> [a]
insSort = insSort' []

-------------------------------------------------------------
-- Вспомогательная функция insSort' имеет дополнительный
-- накапливающий аргумент - отсортированная часть списка
-- Аргументы: 1) уже отсортированная часть списка;
--            2) еще не отсортированные элементы
-- Результат: полностью отсортированный список
-------------------------------------------------------------
insSort' :: Ord a => [a] -> [a] -> [a]
insSort' lst [] = lst   -- все элементы уже отсортированы
insSort' lst (x:tail) = insSort' (insert x lst) tail

-------------------------------------------------------------
-- Вспомогательная функция insert вставляет заданное значение
-- в отсортированный список
-- Аргументы: 1) элемент для вставки;
--            2) упорядоченный список
-- Результат: список (2) с вставленным в него элементом (1)
-------------------------------------------------------------
insert :: Ord a => a -> [a] -> [a]
insert x [] = [x]
```

```
insert x lst@(y:tail) | x <= y = x:lst
                      | otherwise = y:insert x tail

-----------------------------------------------------------
-- Функция для проверки того, что два слова состоят
-- из одних и тех же символов. Функция обобщена на случай
-- списков из любых упорядочиваемых элементов,
-- не обязательно букв.
-- Аргументы: 1) первый список;
--            2) второй список
-- Результат:
--      True, если списки состоят из одних и тех же элементов,
--      False в противном случае
-----------------------------------------------------------
anagrams :: Ord a => [a] -> [a] -> Bool
anagrams w1 w2 =
    (length w1) == (length w2) && (insSort w1) == (insSort w2)

-----------------------------------------------------------
-- Функция для разделения строки на составляющие ее слова.
-- Слова в строке разделяются пробелами.
-- Аргумент: исходная строка
-- Результат: список слов исходной строки
-----------------------------------------------------------
split :: String -> [String]
split = (split' [] []) . trimLeft . reverse

-----------------------------------------------------------
-- Вспомогательная функция для удаления
-- ведущих пробелов из строки.
-- Аргумент: исходная строка
-- Результат: та же строка, но без пробелов в ее начале.
-----------------------------------------------------------
trimLeft (' ':s) = trimLeft s
trimLeft s = s

-----------------------------------------------------------
-- Вспомогательная функция для выполнения основной работы по
-- разделению строки на составляющие ее слова.
-- Аргументы: 1) список уже обработанных слов;
--            2) очередное формирующееся слово;
--            3) еще не обработанный текст
-- Результат: полный список слов исходной строки
-----------------------------------------------------------
-- если текст уже кончился, то выдаем полученный список слов
split' words [] [] = words
split' words w [] = w:words
-- ищем начало очередного слова
split' words [] (' ':tail) = split' words [] tail
-- найден конец слова, присоединяем новое слово к списку
```

```
split' words word (' ':tail) = split' (word:words) [] tail
-- очередная буква слова присоединяется к формируемому слову
split' words word (c:tail) = split' words (c:word) tail
```

Мы можем загрузить этот модуль и проверить, правильно ли работают экспортируемые функции `anagrams` и `split`. Результаты такой проверки показаны на рисунке П1.12.

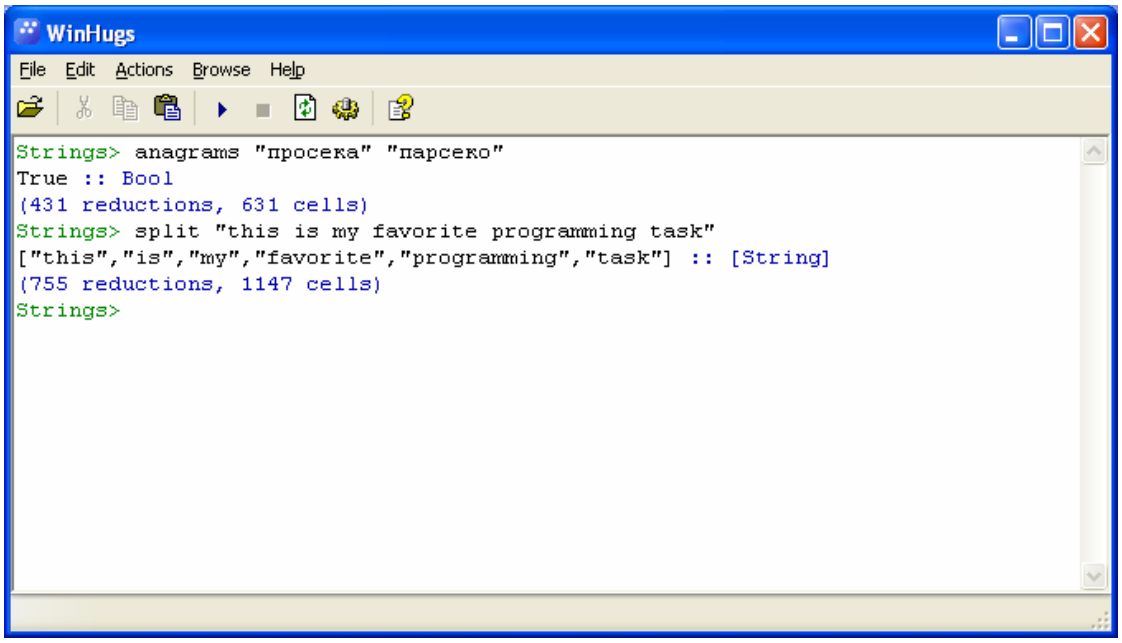

Рис. П1.12. Результаты проверки функций модуля Strings

Теперь напишем модуль, основная функция которого получает словарь из некоторых слов и выделяет в нем группы слов, состоящих из одних и тех же букв. Текст данного модуля приведен в листинге П1.7.

Листинг П1.7. Выделение групп слов, состоящих из одних и тех же букв.

```
{--
 Модуль экпортирует единственную функцию - функцию разбиения
 слов на группы слов, состоящих из одних и тех же букв
--}
module Groups(groups) where

import Strings

-----------------------------------------------------------------
-- Основная функция формирования групп слов, составленных
-- из одних и тех же букв. Одно слово группой не считается,
-- все группы состоят как минимум из двух слов.
-- Аргумент: список слов
-- Результат: список групп слов, каждая из которых состоит из
--            слов, составленных из одних и тех же букв.
-----------------------------------------------------------------
groups :: [String] -> [[String]]
groups [] = []
groups words | length grp == 1 = groups rest
```

```
            | otherwise       = grp : groups rest
    where (grp, rest) = collect words

-------------------------------------------------------------
-- Вспомогательная функция collect собирает в одну группу все
-- слова из списка слов, состоящие из одних и тех же букв.
-- Аргумент: непустой список слов
-- Результат: (grp, rest) где grp - группа слов исходного
--           списка, состоящих из букв первого слова в списке;
--           rest - список всех остальных слов
-------------------------------------------------------------
collect :: [String] -> ([String], [String])
collect (x:words) = collect' [x] [] words

-------------------------------------------------------------
-- Функция collect' имеет дополнительные аргументы.
-- Аргументы:
--    1) группа слов, состоящих из одних и тех же букв;
--    2) список слов, состоящих из других букв;
--    3) еще не обработанный список слов.
-- Результат: см. основную функцию collect
-------------------------------------------------------------
collect' grp rest [] = (grp, rest)
collect' grp@(w1:_) rest (w:words)
                | anagrams w w1 = collect' (w:grp) rest words
                | otherwise     = collect' grp (w:rest) words
```

Наконец, третий модуль будет содержать несколько тестов, проверяющих правильность работы определенных в двух других модулях функций. Текст тестирующего модуля показан в листинге П1.8.

Листинг П1.8. Тестирующий модуль для функций группировки слов

```
{--
 Основная функция данного модуля осуществляет тестирование
 функции разбиения слов на группы слов,
 состоящих из одних и тех же букв
--}
module Main where

import Groups
import Strings

-- тестовые примеры
source =
   "-astronomers- algorithm blood bloude cloed dolce " ++
   " doodle double logarithm morons-stare- no-more-stars"
main = groups (split source)
```

В данном примере существенно, что импортируются сразу два модуля – как модуль `Groups`, так и модуль `Strings`. Это необходимо делать потому, что в тестирующей функции используется как функция

groups из модуля Groups, так и функция split из модуля Strings. Даже несмотря на то, что функция split импортируется в модуль Groups в составе модуля Strings, все-таки функция split не будет «видна» из тестирующего модуля, если в нем имеется импорт только лишь модуля Groups.

На рисунке П1.13 приведен результат работы программы.

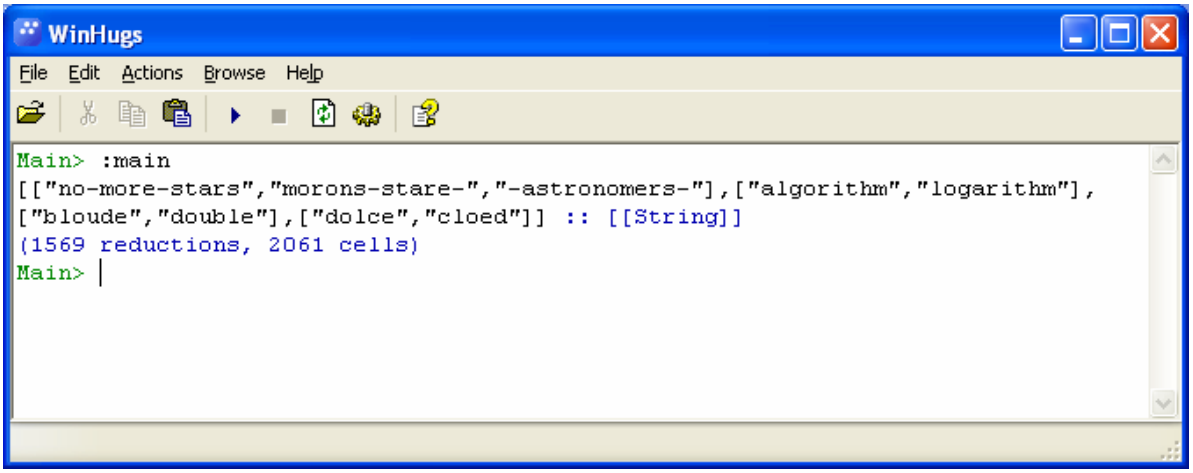

Рис. П1.13. Результат работы программы поиска анаграмм

На этом рисунке видно, что из исходного списка слов удалены слова, которые не имеют анаграмм, такие как "blood" и "doodle".

Оглавление

Функциональное программирование 4

Введение 4
Глава 1. Элементы функционального программирования на языке *Haskell* 8
1.1. Функциональный стиль программирования 8
Примеры решения задач 13
Задания для самостоятельной работы 19
1.2. Элементы языка *Haskell* 20
Примеры решения задач 39
Задания для самостоятельной работы 44
1.3. Определение новых типов данных 45
Глава 2. Что еще есть в функциональном программировании 61
2.1. Концевая рекурсия 61
2.2. Функции высших порядков. Карринг 65
Примеры решения задач 86
Задания для самостоятельной работы 106
2.3. Ленивые вычисления 107
Примеры решения задач 115
Задания для самостоятельной работы 120
Глава 3. Лямбда-исчисление 122
3.1. Формальные теории в программировании 122
3.2. Система вывода результатов 125
Примеры решения задач 136
Задания для самостоятельной работы 138
3.3. Чистое лямбда-исчисление 138
Примеры решения задач 149
Задания для самостоятельной работы 151
Глава 4. Системы исполнения функциональных программ 152
4.1. Промежуточный язык программирования 152
4.2. eval/apply интерпретатор 162
Примеры решения задач 175
Задания для самостоятельной работы 180
4.3. Компиляция в SECD-машину 180
4.4. Функциональные эквиваленты императивных программ 218
Примеры решения задач 225
Задания для самостоятельной работы 228
Приложение 1. Система функционального программирования WinHugs 229
П1.1. Работа в системе WinHugs 229
Оглавление 251