Erlang: И снова

Воды решетом не наносишься, а вот простые числа **Андрею Ушакову** в него попадаются – ну, не без содействия Эратосфена.

Наш эксперт

Андрей Ушаков активно приближает тот день, когда функциональные языки станут мейнстримом.

В этом номере мы продолжим наш практикум по функциональному программированию на языке Erlang и разберемся, как нам работать с разными коллекциями, доступными в языке и в библиотеке Erlang.

Мы помним (**LXF147**), что коллекции предназначены для ситуаций, когда мы хотим обрабатывать хранимые в них данные одним и тем же способом. Этим они концептуально отличаются от другого фундаментального составного типа данных функционального программирования – кортежей. Концептуальное отличие означает, что с точки зрения теории коллекции и кортежи предназначены для хранения данных, обработка которых будет производиться по-разному: данные в коллекциях обрабатываются одним и тем же способом, тогда как данные в кортежах обрабатываются разными способами в зависимости от расположения (позиции) данных. Другими словами, коллекции предназначены для гомогенных данных (возможно, разных типов), а кортежи – для гетерогенных (возможно, имеющих один и тот же тип) В реальной жизни это не обязательно, и грань между коллекциями и кортежами сильно размыта. Мы можем использовать кортежи для хранения гомогенных данных, как сделано, например, в реализации массивов в модуле **array**. Аналогично, мы можем использовать коллекции для хранения гетерогенных данных.

Перед использованием коллекций в своих программах необходимо поставить вопрос о типе (и, возможно, реализации этого типа) используемой коллекции. Этот вопрос относится к правильному проектированию разрабатываемого приложения: если выбрать неподходящий тип коллекции, то такой выбор может привести к дополнительным затратам – как во время реализация, так и во время поддержки приложения.

В языке и библиотеке Erlang доступны следующие типы коллекций: списки, массивы, множества и словари (точнее говоря, списки являются средством самого языка, а все остальные коллекции доступны через библиотечные модули). Для множеств и словарей у нас есть несколько реализаций: стандартная реализация (для которой способ хранения данных не специфицирован, но в данной версии библиотеки она использует хэш-таблицы), реализация на основе упорядоченных списков и реализация на основе сбалансированных двоичных деревьев.

Теперь зададимся вопросом: как правильно выбрать тот или иной тип коллекции (и, возможно, ту или иную реализацию)? Ответ очевиден: все зависит от того, как планируется использовать выбираемую коллекцию и какие к ней есть требования. Если особых требований нет или в первую очередь необходима гибкость работы, которая есть у списков, то выбор очевиден – списки. Если нужно хранить данные в определенном месте (по индексу) и уметь их изменять, не меняя места (индекса), то наш выбор – массивы.

Тут следует заметить следующее: списки легко позволяют получить доступ к элементу по определенному индексу (для этого служит функция **lists:nth/2**), но задать новое значение в списке по определенному индексу так просто нельзя (для этого нет соответствующей функции в модуле **lists**). Чтобы поменять значение по определенному индексу в списке, необходимо действовать, как в следующем примере. Пусть **List = [1,2,3,4,5,6]** – исходный список, а **N = 4** – индекс элемента, который необходимо поменять (индексы списка начинаются с 1); тогда решение этой задачи может быть следующим:

```
{Part1, [Element | Part2]} = lists:split(N – 1, List).
P1 ++ [444] ++ P2.
```

Результатом будет список, получаемый из исходного заменой 4-го элемента на значение 444: **[1,2,3,444,5,6]**. Если требуется функциональность множеств либо словарей, следует выбрать одну из реализаций множеств и словарей (в зависимости от того, важен ли нам порядок элементов). Что интересно, если нам нужна коллекция на основе пар ключ–значение (для хранения таких пар и предназначены словари), то вместо словаря можно выбрать списки (если нам важна их гибкость): модуль **lists** содержит богатый набор функций для работы со списками, хранящими наборы ключ–значение (в виде кортежей, что естественно). Не забудем также, что все типы коллекций содержат функции **from_list/1** и **to_list/1** для конвертации своего содержимого в списки и обратно.

Рассмотрим несколько примеров. Для начала реализуем алгоритм, известный как «Решето Эратосфена». Это алгоритм нахождения всех простых чисел до некоторого заданного числа, авторство которого приписывают древнегреческому математику Эратосфену Киренскому. Основная его идея такова. Мы выписываем все числа от 2 до N (где N определяет интересующий нас диапазон). Затем берем первое число (это будет 2) и вычеркиваем все числа из списка, кратные ему. Далее, берем следующее не вычеркнутое число (это будет 3) и вычеркиваем из списка все числа, кратные этому числу. И так будем поступать до тех пор, пока не упремся в границу диапазона.Тогда все не вычеркнутые числа в списке будут являться простыми числами в заданном диапазоне.

Из основной идеи алгоритма видно, что нам необходимо иметь операцию вычеркивания. Понятно, что ее можно реализовать так, чтобы число действительно «вычеркивалось» (удалялось) из коллекции, но вряд ли такая реализация будет оптимальной (удаление элемента из коллекции – не самая легкая операция). Поэтому мы реализуем операцию вычеркивания следующим образом: будем хранить пару число–флаг, который показывает, вычеркнуто ли данное число из списка или нет.

Теперь выберем тип коллекции для хранения таких пар. На первый взгляд кажется, что идеальный вариант в нашем случае – словарь (ассоциативная коллекция для хранения пар ключ – значение). Но давайте учтем тот факт, что мы будем работать с непрерывным рядом чисел в диапазоне от 2 до N; поэтому можно выбрать в качестве коллекции массив. При таком выборе число становится индексом массива (точнее, число за вычетом 2, т. к. нумерация элементов массива начинается с 0), а флаг – значением элемента массива.

Вполне логичен вопрос: а почему бы не использовать в качестве коллекции списки? Они ведь тоже являются коллекцией с доступом к элементам по индексу! Причина, по которой мы не выбираем списки в качестве коллекции для хранения – в том, что для списков отсутствует операция установки значения элемента по определенному индексу, в отличие от массивов.

После этого небольшого обсуждения алгоритма и некоторых деталей его реализации перейдем к делу. Как и во всех прочих примерах, реализация начинается с объявления модуля:

```
-module(eratos_sieve).
```

практикум

Следующий шаг тоже вполне очевиден: объявление экспортируемых функций. В нашем примере достаточно экспортировать всего лишь одну функцию: **get_primes/1**, возвращающую список всех простых чисел от 2 до N, где N является входным параметром функции.

```
-export([get_primes/1]).
```

Теперь перейдем к реализации алгоритма. Начнем с экспортируемой функции **get_primes/1**. Она выполняет две задачи: формирует ссылку на массив **Sieve** [*англ.* решето] с вычеркнутыми и не вычеркнутыми номерами и превращает решето в список простых чисел. Первую задачу решает функция **process_iteration/3**, вторую задачу решает функция **create_number_list/4**.

Обратите внимание, как мы создаем исходный массив, в котором еще нет вычеркнутых чисел: мы используем функцию **array:new/1**, в которую передаем список пар ключ–значение (список кортежей из двух элементов). В нашем случае мы создаем массив фиксированного размера, все элементы которого имеют значение по умолчанию, равное **true** (**true** означает, что соответствующее число еще не вычеркнуто).

```
get_primes(MaxNumber) ->
  Sieve = process_iteration(array:new([{size, MaxNumber-1},
{fixed, true}, {default, true}]), 2, MaxNumber), create_number_
list(Sieve, 2, MaxNumber, []).
```

Основная задача алгоритма – вычеркнуть все не простые числа из списка чисел (сформировать решето). Ее решает функция **process_iteration/3**. Эта функция является рекурсивной функцией (функцией с хвостовой рекурсией), каждая итерация которой обрабатывает очередное простое число из списка (вычеркивает все числа, кратные обрабатываемому числу). Эта функция содержит два варианта. Первый вариант обрабатывает ситуацию, когда мы не можем найти очередное простое число (а это означает, что мы дошли до конца списка); этот вариант является не рекурсивным и позволяет нам закончить рекурсивное вычеркивание чисел. Второй вариант является рекурсивным, и на каждой итерации для заданного простого числа (через один из параметров), вычеркивает все числа, кратные заданному простому числу.

Операцию вычеркивания можно слегка оптимизировать: очевидно, что для текущего простого числа **p** вычеркивать надо, начиная с числа p^2, так как все составные числа до p^2 уже вычеркнуты. После вычеркивания всех кратных чисел, функция **process_iteration/3** вызывается для следующего простого (не вычеркнутого) числа.

```
process_iteration(Sieve, not_found, _MaxNumber) -> Sieve;
process_iteration(Sieve, Current, MaxNumber) ->
  NewSieve = erase_multiple(Sieve, Current*Current, MaxNumber,
Current),  process_iteration(NewSieve, find_next_prime(Sieve,
Current+1, MaxNumber), MaxNumber).
```

В конце каждой итерации функции **process_iteration/3** нам необходимо найти следующее простое (не вычеркнутое) число относительно заданного простого числа. За эту задачу отвечает функция **find_next_prime/3**. В ней последовательно просматривается список чисел (при помощи хвостовой рекурсии), начиная с некоторого заданного числа, и выбирается первое не вычеркнутое число. Если в какой-то момент мы перебрали весь список и не нашли ни одного не вычеркнутого числа, то мы возвращаем атом **not_found** (за это отвечает первый вариант этой функции). Это значение определяет выбор первого варианта функции **process_iteration/3**, что означает окончание операции вычеркивания (формирования решета).

```
find_next_prime(_Sieve, Current, MaxNumber) when Current >
MaxNumber -> not_found;
find_next_prime(Sieve, Current, MaxNumber) ->
  case array:get(Current-2, Sieve) of
  true -> Current;
  false -> find_next_prime(Sieve, Current+1, MaxNumber)
  end.
```

Следующая функция, которую мы рассмотрим – это функция для вычеркивания чисел **erase_multiple/4**. В этой функции мы проходим по массиву (при помощи хвостовой рекурсии) и вычеркиваем числа, кратные заданному числу (операция вычеркивания означает, что мы устанавливаем значение **false** по индексу, соответствующему числу).

```
erase_multiple(Sieve, Current, MaxNumber, _Delta) when Current
> MaxNumber -> Sieve;
erase_multiple(Sieve, Current, MaxNumber, Delta) ->
  erase_multiple(array:set(Current-2, false, Sieve), Current+Delta,
MaxNumber, Delta).
```

И, наконец, остался последний не рассмотренный нами метод: **create_number_list/4**. Этот метод служит для преобразования массива флагов, вычеркнуто или не вычеркнуто число, соответствующее индексу в списке простых чисел. В этом методе мы последовательно идем по массиву флагов (при помощи хвостовой рекурсии), и если флаг равен **true** (т. е. соответствующее число не вычеркнуто), то мы добавляем это число в начало списка простых чисел. Пройдя весь массив флагов, мы переворачиваем список (при помощи **lists:reverse/1**), чтобы простые числа шли в нем в порядке возрастания, и возвращаем его. Напомним также следующее соотношение между числом и соответствующим ему индексом флага в массиве: т. к. мы рассматриваем диапазон чисел от 2 до N, а нумерация элементов массива начинается с 0, то значение выражения «число минус его индекс» равно 2.

```
create_number_list(_Sieve, Current, MaxNumber, Dest) when
Current > MaxNumber -> lists:reverse(Dest);
create_number_list(Sieve, Current, MaxNumber, Dest) ->
  case array:get(Current-2, Sieve) of
    true -> create_number_list(Sieve, Current+1, MaxNumber,
[Current]++Dest);
    false -> create_number_list(Sieve, Current+1, MaxNumber,
Dest)
    end.
```

Теперь пора все откомпилировать и удостовериться в корректности нашей реализации алгоритма. Для этого запустим среду выполнения Erlang и скомпилируем модуль **eratos_sieve** (командой **c(eratos_sieve).**). Проверим, что все работает правильно: так, например, вызов **eratos_sieve:get_primes(20).** возвращает все простые числа в диапазоне от 2 до 20: **[2, 3, 5, 7, 11, 13, 17, 19]**.

На основе приведенного выше примера давайте решим следующий пример, в качестве которого возьмем задачу номер 35 »

» **Пропустили номер?** Узнайте на с. 104, как получить его прямо сейчас.

с сайта Project Euler (http://projecteuler.net/problem=35). Условие этой задачи выглядит следующим образом.

Число 197 называется циклическим простым числом, потому что все числа, получающиеся из этого числа при помощи циклических перестановок цифр (это будут числа 197, 971 и 719), являются простыми. Существует только 13 таких простых чисел меньше 100: 2, 3, 5, 7, 11, 13, 17, 31, 37, 71, 73, 79 и 97. Сколько существует циклических простых чисел меньше 1 000 000?

Из условия задачи понятно, что первым шагом будет поиск всех простых чисел меньше 1 000 000 – именно то, что делает предыдущий пример. После этого для каждого простого числа мы будем строить все его циклические перестановки и проверять, являются ли они простыми. Понятно, что предварительно мы выясним, не обрабатывалось ли данное простое число.

Начнем реализацию как обычно: с объявления модуля и списка экспортируемых функций.

```
-module(problem_035).
-export([solve/1]).
```

Список экспортируемых функций содержит у нас только одну функцию – **solve/1**. Эта функция возвращает количество циклических простых чисел, меньших, чем значение параметра функции **MaxNumber**. Работа функции **solve/1** состоит из следующих двух шагов: создание списка всех простых чисел, меньших значения **MaxNumber**, и создание множества циклических простых чисел из списка простых чисел (количество циклических простых чисел равно размеру этого множества). Первый шаг решается при помощи функции **eratos_sieve:get_primes/1** из предыдущего примера. Для второго шага мы вводим функцию **find_circular_primes/1**.

```
solve(MaxNumber) ->
  PrimeNumbers = eratos_sieve:get_primes(MaxNumber),
  sets:size(find_circular_primes(PrimeNumbers)).
```

Функция **find_circular_primes/1** является интерфейсной функцией к функции **find_circular_primes/3**. В ней из списка простых чисел, полученного на предыдущем шаге, мы строим множество простых чисел. Помимо этого, мы создаем пустое множество для хранения циклических простых чисел.

```
find_circular_primes(PrimeNumbers) ->
  find_circular_primes(PrimeNumbers, sets:from_
list(PrimeNumbers), sets:new()).
```

Функция **find_circular_primes/3** является центральной для построения множества циклических простых чисел. Функция принимает три параметра: список необработанных простых чисел, множество всех простых чисел, меньших заданного, и множество циклических простых чисел. Ситуация, когда список необработанных простых чисел пуст, является условием окончания обработки (а также шаблоном для выбора должного варианта функции).

Основной вариант этой функции рекурсивно обрабатывает (при помощи хвостовой рекурсии) список необработанных простых чисел поэлементно. На каждой итерации мы проверяем, не находится ли текущее простое число во множестве циклических простых чисел (что означает, что мы это число уже обработали). Если проверка дает положительный результат, то мы просто переходим к следующей итерации. В противном случае, мы строим список всех чисел, получаемых из обрабатываемого простого числа циклическим сдвигом цифр (этот список включает и обрабатываемое простое число), проверяем, все ли числа из построенного списка являются простыми числами, и если все числа из построенного списка являются простыми, добавляем их во множество циклических простых чисел. После чего переходим к следующей итерации.

```
find_circular_primes([], _PrimeSet, CircularPrimeSet) ->
CircularPrimeSet;
find_circular_primes([PrimeNumber | Rest], PrimeSet,
CircularPrimeSet) ->
  case sets:is_element(PrimeNumber, CircularPrimeSet) of
    false ->
      CircularNumbers = create_circular_numbers(PrimeNumber),
                  NewCircularPrimeSet = check_and_add_
numbers(CircularNumbers, PrimeSet, CircularPrimeSet),
      find_circular_primes(Rest, PrimeSet, NewCircularPrimeSet);
    true -> find_circular_primes(Rest, PrimeSet, CircularPrimeSet)
  end.
```

Функция **check_and_add_numbers/3** проверяет, являются ли числа из списка, построенного при помощи циклического сдвига цифр исходного простого числа, простыми числами. Если все числа являются простыми, это означает, что все они являются и циклическими простыми числами. В таком случае, мы все эти числа добавляем во множество циклических простых чисел. Для проверки списка целых чисел мы используем функцию **lists:all/2**, для добавления этих чисел во множество циклических простых чисел – функцию **lists:foldl/3**.

```
check_and_add_numbers(Numbers, PrimeSet, CircularPrimeSet)
->
  Check = lists:all(fun(Number) -> sets:is_element(Number,
PrimeSet) end, Numbers),
  if
    Check == true -> lists:foldl(fun(Number, Dest) -> sets:add_
element(Number, Dest) end, CircularPrimeSet, Numbers);
    Check == false -> CircularPrimeSet
  end.
```

Функция **create_circular_numbers/1** создает список всех возможных чисел, полученных из исходного числа циклическим сдвигом цифр. Для этого мы преобразуем число в список всех цифр, составляющих это число (при помощи **get_digits/1**), после чего создаем список всех возможных циклических перестановок полученного списка цифр (при помощи **get_circular_shifts/1**), и, наконец, все элементы (списки цифр) списка всех циклических перестановок цифр преобразуем обратно в числа (при помощи **lists:map/2** и **get_number/1**).

```
create_circular_numbers(Number) ->
  lists:map(fun(Digits) -> get_number(Digits) end,
get_circular_shifts(get_digits(Number))).
```

Функция **get_digits/1** преобразует число в список всех его цифр. Для этого число преобразуется в строку (в список символов, составляющих строковое представление числа) при помощи BIF **integer_to_list/1**, после чего строка преобразуется в список цифр при помощи техники конструирования списков [List Comprehensions].

```
get_digits(Number) -> [Char-$0 || Char
<- integer_to_list(Number)].
```

Следующая функция является обратной функцией к предыдущей функции, т. е. преобразует список цифр в число, состоящее из этих цифр. Для этого мы преобразуем список цифр в список символов строкового представления соответствующего числа (т. е. в строку), после чего преобразуем полученную строку в целое число при помощи BIF **list_to_integer/1**.

```
get_number(Digits) -> list_to_integer([Digit+$0 || Digit <- Digits]).
```

Функция **get_circular_shifts/1** является интерфейсной функцией к функции **get_circular_shifts/3**. Она возвращает список, элементами которого являются списки, полученные из исходного списка при помощи всех возможных циклических перестановок элементов.

```
get_circular_shifts(Source) -> get_circular_shifts(Source, Source,
[]).
```

И, наконец, последняя функция (**get_circular_shifts/3**) делает всю работу по формированию списка, элементами которого являются списки, полученные из исходного списка при помощи всех возможных циклических перестановок элементов.

Циклическая перестановка элементов означает, что один или несколько элементов из начала списка идут в конец списка в том же порядке, в котором они шли в начале списка. Так, например, для списка [1, 2, 3, 4] списки [2, 3, 4, 1] и [3, 4, 1, 2] получаются из исходного списка циклической перестановкой одного и двух элементов соответственно. Список [3, 4, 1, 2] получается из списка [2, 3, 4, 1] циклической перестановкой одного элемента. Этот пример показывает нам, что мы можем рекурсивно (при помощи хвостовой рекурсии) реализовать данный алгоритм: мы начинаем с исходного списка, на каждом шаге циклической перестановкой одного элемента получаем новый список, и обработку заканчиваем, когда мы вернемся к исходному списку. Понятно, что на каждом шаге мы запоминаем полученный список.

Отметим еще следующее: условием окончания работы функции является ситуация, когда новый список, полученный из текущего циклической перестановкой одного элемента, совпадает с исходным списком (эту ситуацию обрабатывает соответствующий вариант функции **get_circular_shifts/3**). Однако, когда мы начинаем работу, у нас текущий список тоже совпадает с исходным списком. Чтобы разрешить эту ситуацию, мы вводим еще один вариант функции (что важно перед вариантом, заканчивающим работу). Этот вариант обрабатывает ситуацию, когда текущий список совпадает с исходным списком и список, который хранит все списки, полученные циклической перестановкой, пуст (что логично, т. к. мы только начали).

```
get_circular_shifts(Source, Source, []) ->
  [Head | Tail] = Source,
  get_circular_shifts(Source, Tail ++ [Head], [Source]);
get_circular_shifts(Source, Source, Dest) -> Dest;
get_circular_shifts(Source, Current, Dest) ->
  [Head | Tail] = Current,
  get_circular_shifts(Source, Tail ++ [Head], [Current] ++ Dest).
```

Теперь проверим, что все у нас работает правильно. Для этого запускаем среду выполнения Erlang и компилируем модуль **eratos_sieve** (полученный в предыдущем примере) и модуль **problem_035** (для компиляции выполняем команду **c(module_name).** в консоли среды выполнения Erlang, где **module_name** – имя соответствующего модуля). Для работы необходимо, чтобы оба откомпилированных модуля располагались в одной директории (по сути, это ограничение не является обязательным, и его можно обойти; как это сделать, мы поговорим на одном из будущих уроков). Зная, что количество циклических простых чисел, меньших 100, равно 13, вводим в консоли среды выполнения Erlang команду **problem_035:solve(99).** и получаем желаемый результат – 13. Теперь решим нашу исходную задачу: определить количество циклических простых чисел, меньших 1 000 000. Для этого введем в консоли среды выполнения Erlang команду **problem_035:solve(999999).** – результат будет 55.

Каждый может зайти на сайт Project Euler в задачу номер 35 (http://projecteuler.net/problem=35), ввести данный ответ и убедиться, что он правильный (правда, для этого ему придется зарегистрироваться).

Сегодня мы на практических задачах поработали с разными типами коллекций. Не следует забывать, что к выбору типа коллекций при реализации того или иного алгоритма следует подходить вдумчиво: этот выбор влияет на реализацию алгоритма. Представьте, например, что если в алгоритме «Решета Эратосфена» мы вместо массивов решили бы использовать словари; как изменилась бы реализация этого алгоритма? (Читатель может для интереса попробовать переписать реализацию алгоритма с использованием любой другой коллекции и сравнить полученный результат с исходной реализацией.) Наш практикум на этом не заканчивается: в следующем номере мы займемся «черной магией» битовых строк. LXF

Полезные заметки

Списки

Списки – фундаментальный тип данных, встроенный в язык. Список – это упорядоченная коллекция, хранящая набор элементов в порядке их расположения в списке (в порядке их позиции в списке); это наиболее богатый по возможностям тип коллекций в языке Erlang. Для обработки списков доступны следующие возможности:

- Операция соответствия шаблону [Pattern Matching]. С ее помощью можно выделить первые N элементов с начала списка и его остаток. Имеет следующий синтаксис: **[$Head_1$, $Head_2$, …, $Head_N$ | Rest]**, где **$Head_1$**, **$Head_2$**, …, **$Head_N$** – элементы с начала списка, **Rest** – список, содержащий остаток (исходный список минус N элементов с начала исходного списка). Позволяет обрабатывать списки рекурсивно.
- Операция конструирования списков [List **Comprehensions**]. Данная операция позволяет генерировать список из декартова произведения нескольких множеств источников, с применением к этому произведению фильтров. Имеет следующий синтаксис: **[Expr || Qualifier1, …, QualifierN]**, где **Expr** – произвольное выражение для формирования элементов конечного списка, **Qualifier1**, …, **QualifierN** – либо генератор (как списков, так и битовых строк), либо фильтр.
- Набор функций для работы со списками, определенный в модуле **lists**. Он позволяет решать практически любые задачи, которые могут встретиться при работе со списками. Следует заметить, что данный набор функций не позволяет задать новый объект для определенной позиции (по определенному индексу).
- Операторы "++" и "--" для создания суммы и разности двух списков.

Массивы

Массив – это коллекция уровня библиотеки языка Erlang, связывающая хранимые элементы с их позицией (индексом) в массиве. Массивы, в отличие от списков, позволяют изменять значение элемента по определенному индексу. Функции для работы с массивами определены в модуле **array** (и это единственный способ работать с ними). При работе с функциями из этого модуля следует помнить, что нумерация элементов массивов начинается с 0 (в отличие от элементов списков).

Множества

Множество – это коллекция уровня библиотеки языка Erlang, обеспечивающая уникальность хранимых элементов (это означает, что во множестве не могут храниться дубликаты любого объекта).

В библиотеке языка Erlang существует три реализации множеств: стандартная реализация (для которой представление данных не определено, но в данной версии используется хэш-таблица), реализация на основе упорядоченного списка и реализация на основе сбалансированного дерева. Разница между стандартной реализацией и двумя другими в том, что в стандартной реализации порядок хранения элементов не задан, тогда как в двух других реализациях используется естественный порядок хранения элементов.

Функции для работы со стандартной реализацией множеств определены в модуле **sets**, функции для работы с реализацией на основе упорядоченного списка – в модуле **ordsets**, функции для работы с реализацией на основе сбалансированного двоичного дерева – в модуле **gb_sets**.

Словари

Словарь – это ассоциативная коллекция, хранящая пары ключ–значение. В языке Erlang словари определяются на уровне библиотеки. Существуют две реализации словарей: стандартная реализация (для которой представление данных не определено, но в данной версии используется хэш-таблица) и реализация на основе упорядоченного списка.

Разница между стандартной реализацией и реализацией на основе упорядоченного списка в том, что в первой порядок хранения элементов не задан, тогда как во второй элементы расположены в соответствии с естественным порядком хранения ключей.

Функции для работы со стандартной реализацией словарей определены в модуле **dict**, функции для работы с реализацией словарей на основе упорядоченного списка – в модуле **orddict**. Реализации словаря на основе сбалансированного двоичного дерева в библиотеке языка Erlang нет (в отличие от соответствующей реализации множества), но зато присутствует просто модуль для работы со сбалансированным двоичным деревом **gb_trees**. Эту функциональность в некоторой степени можно считать реализацией словаря на основе сбалансированного двоичного дерева. И, что вполне очевидно, данные в таком дереве хранятся в соответствии с естественным порядком ключей.