Последовательное программирование.

В этой главе мы рассмотрим, как на Эрланге пишутся простые последовательные программы.

В первом разделе мы поговорим о модулях и функциях. Мы увидим как принцип сопоставления шаблонов, о котором мы узнали в предыдущей главе, используется при объявлении функций.

Сразу после этого мы вернемся к списку покупок, который мы составили в предыдущей главе, и напишем некоторый код, который будет подсчитывать общую стоимость этого списка.

По мере продвижения вперед, мы будем постепенно улучшать написанные нами программы. И вы, при этом, будете видеть как программы на Эрланге постепенно развиваются, а не просто получите законченные программы, без объяснения того, как мы их получили. Понимание основных шагов этого процесса, может подсказать вам, как вы можете его применять для своих программ.

Затем мы поговорим о функциях высшего порядка (называемых funs), и о том, как они могут быть использованы для создания ваших собственных управляемых абстракций.

В завершении мы поговорим о часовых (guard), записях, и выражениях **case** и **if**.

Итак, давайте приступим…

Модули

В Эрланге, модули – это основная единица кода. Все функции, которые мы пишем, содержатся в модулях. Модули Эрланга сохраняются в файлах с расширением *.erl*. Модули должны быть откомпилированы перед тем как их содержимое будет готово к выполнению. Скомпилированный модуль будет иметь расширение .beam. (beam - это сокращение от "Bogdan’s Erlang Abstract Machine" ("Абстрактная машина Эрланга Богдана"). Богумил (Богдан) Хаусман написал компилятор Эрланга в 1993 году и разработал новый набор инструкций для Эрланга)

Перед тем как написать свой первый модуль, давайте вспомним о сопоставлении шаблонов (или образцов). Для начала создадим пару структур данных, представляющих собой прямоугольник и круг. Затем извлечем из этих структур значения длин сторон для прямоугольника и радиуса для круга. Например, вот так:

```
1> Rectangle = {rectangle, 10, 5}.
```

```
{rectangle, 10, 5}.
2> Circle = {circle, 2.4}.
{circle,2.40000}
3> {rectangle, Width, Ht} = Rectangle.
{rectangle,10,5}
4> Width.
10
5> Ht.
5
6> {circle, R} = Circle.
{circle,2.40000}
7> R.
2.40000
```

В строках 1 и 2 мы создали прямоугольник и круг. В строках 3 и 6 мы извлекли ("*распаковали*") значения полей прямоугольника и круга, используя сопоставление шаблонов. В строках 4, 5 и 7 мы выводим значения переменных, которые были получены путем сопоставления шаблонов. После исполнения строки 7, мы имеем следующие связанные переменные: {Width -> 10, Ht -> 5, R -> 2.4}.

Перейти от сопоставления шаблонов (образцов) в командной строке к сопоставлению шаблонов в функциях - это совсем несложный шаг. Давайте начнем с написания функции area вычисляющей площадь прямоугольника и круга. Мы поместим ее в модуль geometry, а модуль сохраним в файле geometry.erl. Данный модуль выглядит так: /файл geometry.erl /

```
-module(geometry).

-export([area/1]).

area({rectangle, Width, Ht}) -> Width * Ht;

area({circle, R}) -> 3.14159 * R * R.
```

Не обращайте пока внимания на директивы –module и –export (мы поговорим о них позже). сейчас я прошу вас пристально посмотреть на код функции area.

Функция area содержит два варианта сопоставления аргументов, которые мы будем называть клаузами (от англ. слова clause = условие, клауза, клаузула (условия применения в юридическом документе)), либо же "вариантом применения функции". Клаузы между собой разделяются точкой с запятой, а последняя клауза функции завершается точкой.

У каждого варианта применения функции (клаузы) есть свои заголовок и тело;

заголовок содержит имя функции и шаблон аргументов (в круглых скобках), а тело состоит из последовательности выражений (См. раздел 5.4 *Выражения и Последовательности выражений*), которые вычисляются если шаблон аргументов в заголовке совпал с реально выданными этой функции аргументами. Сопоставление шаблонов аргументов с реальными аргументами происходит в том-же порядке, в котором были объявлены клаузы функции. Обратите внимание, что шаблон аргументов {rectangle, Width, Ht} является частью объявления клаузы функции. Каждый шаблон аргументов определяет только один вариант применения функции (клаузу). Давайте посмотрим на первую клаузу:

```
area({rectangle, Width, Ht}) -> Width * Ht;
```

Это функция вычисления площади прямоугольника. Когда мы вызываем geometry:area({rectangle, 10, 5}), сопоставлением шаблонов аргументам присваиваются выданные значения {Width 7-> 10, Ht 7-> 5}. А после стрелки -> следует код, который будет потом выполнен. Это просто Width * Ht, или же 10*5, или же 50.

Сейчас мы это скомпилируем и запустим:

```
1> c(geometry).
{ok,geometry}
2> geometry:area({rectangle, 10, 5}).
50
3> geometry:area({circle, 1.4}).
6.15752
```

Что здесь произошло? В строке 1 мы выполнили команду c(geometry),

которая скомпилировала код в файле (модуле) geometry.erl. Компилятор возвратил

{ok,geometry}, что значит, что компиляция прошла успешно и модуль geometry был скомпилирован и загружен. В строках 2 и 3 мы вызываем функции содержащиеся в модуле geometry. Обратите внимание, мы должны указывать имя функции вместе с именем модуля, для однозначного указания, какую именно функцию мы хотим вызвать (в разных модулях могут быть функции с одинаковыми именами).

Развитие возможностей программы.

Теперь, допустим, мы хотим дополнить нашу программу, добавив квадрат к нашим геометрическим объектам. Мы можем написать это так:

```
area({rectangle, Width, Ht}) -> Width * Ht;
area({circle, R}) -> 3.14159 * R * R;
area({square, X}) -> X * X.
```

или так:

```
area({rectangle, Width, Ht}) -> Width * Ht;
area({square, X}) -> X * X;
area({circle, R}) -> 3.14159 * R * R.
```

В данном случае порядок порядок кауз данной функции не столь важен; для программы не имеет значения в каком порядке они следуют. Потому что шаблоны аргументов в этих клаузах взаимоисключающие. Это делает написание и расширение программ очень простым делом - мы просто добавляем новые варианты применения со своими шаблонами аргументов к нашей функции. Однако, в общем случае, порядок следования клауз с шаблонами аргументов имеет значение. При вызове функции, аргументы, с которыми была вызванна фукнция, сопоставляются с шаблонами ее применения в том порядке, в котором они были объявлены в файле.

Перед как мы пойдем дальше, вы должны обратить внимание на некоторые детали реализации функции area:

- Функция area содержит несколько клауз. Когда мы вызываем функцию, выполнение начинается с первой клаузы, совпавшей с аргументами, с которыми функция была вызвана.
- Наша функция не предусматривает случай, когда не одна из клауз не совпадет с аргументами - тогда наша программа завершится с ошибкой времени исполнения. Это сделано нами преднамеренно.

Куда подевался мой код?

Если вы загрузили примеры кода из этой книги или хотите написать свои собственные примеры, вам нужно убедиться, что, когда вы запускаете компиляцию из оболочки Эрланга, вы находитесь в директории, в которой система сможет найти ваш код.

Если вы запустили оболочку Эрланга из командной строки, то вам будет необходимо изменить текущую директорию на ту, в которой содержится ваш код, перед тем, как пытаться его скомпилировать.

Если вы запускаете оболочку Эрланга в Windows, использую стандартный Erlang дистрибутив, вам также следует изменить директории на те, в которых вы храните ваш код. Две команды в Эрланге помогут вам перейти в нужную директорию. Если вы заблудились, то команда pwd() напечатает вам текущую директорию оболочки Эрланга. А команда cd(Dir) изменит текущую рабочую директорию на директорию Dir. Однако Вы должны использовать прямые слеши в имени директории (даже в Windows); например:

```
1> cd("c:/work" ).
c:/work
```

Подсказка пользователям Windows: Создайте файл `C:/Program Files/erl5.4.12/.erlang` (измените его путь, если в вашей системе он отличается). И добавьте следующие команды в этот файл:

```
io:format("consulting .erlang in \~p\~n",
[element(2,file:get_cwd())]).
%% Edit to the directory where you store your code
c:cd("c:/work" ).
io:format("Now in:\~p\~n" , [element(2,file:get_cwd())]).
```

Теперь, когда вы запустите оболочку Эрланга, текущая директория автоматически изменится на `C:/work`, или ту, которую вы укажите.

(Прим. переводчика: с русскими буквами в Эрланге могут быть проблемы. Везде.)

Многие языки программирования, такие как Си, содержат только одну точку входа в свои функции. Поэтому, если мы захотим реализовать нашу функцию на Си, то аналогичный код будет выглядеть примерно следующим образом:

```
enum ShapeType { Rectangle, Circle, Square };

struct Shape {
    enum ShapeType kind;
    union {
        struct { int width, height; } rectangleData;
        struct { int radius; } circleData;
        struct { int side;} squareData;
    } shapeData;
};

double area(struct Shape* s) {
    if( s->kind == Rectangle ) {
        int width, ht;
        width = s->shapeData.rectangleData.width;
        ht = s->shapeData.rectangleData.ht;
        return width * ht;
    } else if ( s->kind == Circle ) {
...
```

Данный Си-код, по-сути, также выполняет операцию сопоставления шаблонов реальным аргументам функции. Но только в данном случае, программист должен сам

написать код реализующий все это, да еще и убедиться, что он работает корректно.

В Эрланге же мы просто пишем шаблоны аргументов в клаузах функций, а компилятор генерирует оптимальный код для сопоставления шаблонов, который всегда выбирает корректный вариант.

Мы можем также рассмотреть, как будет выглядеть аналогичный код написанный на Java (в духе объектно-ориентированного программирования):

```
abstract class Shape {
    abstract double area();
}

class Circle extends Shape {
    final double radius;
    Circle(double radius) { this.radius = radius; }
    double area() { return Math.PI * radius*radius; }
}

class Rectangle extends Shape {
    final double ht;
    final double width;

    Rectangle(double width, double height) {
        this.ht = height;
        this.width = width;
    }

    double area() { return width * ht; }
}

class Square extends Shape {
    final double side;

    Square(double side) {
        this.side = side;
    }

    double area() { return side * side; }
}
```

Если сравнить этот код с кодом написанный на Эрланге, то можно легко заметить, что в Java-программе код для вычисления площади находится в трех различных местах, а в Erlang-программе этот код находится компактно, в одном месте.

Вернемся к шоппингу

Напомню, что у нас имеется список покупок, который выглядит вот так ({*предмет,количество*}):

```
[{oranges,4},{newspaper,1},{apples,10},{pears,6},{milk,3}]
```

Теперь предположим, что мы хотим узнать сколько будут стоить все наши продукты. Для этого мы должны знать какова стоимость каждого элемента из списка покупок. Давайте соберем эту необходимую информацию в модуле с именем shop. Запустите свой любимый текстовый редактор и введите код, приведенный ниже, в файл с именем shop.erl. /файл shop.erl /

```
-module(shop).

-export([cost/1]).

cost(oranges) -> 5;

cost(newspaper) -> 8;

cost(apples) -> 2;

cost(pears) -> 9;

cost(milk) -> 7.
```

Функция cost/1 (что означает: функция cost с одним аргументом) состоит из 5 клауз. Заголовок каждой клаузы содержит шаблон аргументов (в данном случае это простейший шаблон, состоящий из одного атома). Когда мы вызовем shop:cost(X) система попытается сопоставить X с каждым шаблоном в этих клаузах функции cost. Если совпадение будет найдено, выполнится код следующий в этой клаузе вслед за символом ->.

Кроме того, функция cost/1 должна быть *экспортирована* из своего модуля; это необходимо если мы хотим вызывать ее вне модуля. (Также можно написать -compile(export_all). , что экспортирует все функции, содержащиеся в модуле с такой директивой).

Давайте протестируем все это. Мы скомпилируем и запустим нашу программу из оболочки Эрланга:

```
1> c(shop).
```

```
{ok,shop}
2> shop:cost(apples).
2
3> shop:cost(oranges).
5
4> shop:cost(socks).
=ERROR REPORT==== 30-Oct-2006::20:45:10 ===
Error in process <0.34.0> with exit value:
    {function_clause,[{shop,cost,[socks]},
    {erl_eval,do_apply,5},
    {shell,exprs,6},
    {shell,eval_loop,3}]}
```

В строке 1 мы компилируем модуль из файла с именем shop.erl. В строках 2 и 3 мы проверяем сколько стоят яблоки и апельсины (результат: 2 и 5 денежных единиц (не важно каких)). В строке 4 мы запрашиваем стоимость носок, но подобного варианта применения (клаузы) в cost нет, и поэтому, мы получаем ошибку сопоставления шаблонов, о чем система выводит нам сообщение. (Пометка “function_clause” в сообщении об ошибке, как раз и говорит нам о том, что вызов функции оказался невозможным, поскольку ни одна из ее клауз не подошла переданным аргументам.)

Вернемся к нашему списку. Предположим мы имеем следующий список покупок:

```
1> Buy = [{oranges,4}, {newspaper,1}, {apples,10}, {pears,6}, {milk,3}].
[{oranges,4},{newspaper,1},{apples,10},{pears,6},{milk,3}]
```

И, допустим, мы мы хотим подсчитать общую стоимость всех элементов этого списка. Мы можем сделать это, например, вот так: /файл shop1.erl /

```
-module(shop1).

-export([total/1]).

total([{What, N}|T]) -> shop:cost(What) * N + total(T);

total([]) -> 0.
```

Давайте поэкспериментируем с этим:

```
2> c(shop1).
{ok,shop1}
3> shop1:total([]).
0
```

Почему здесь 0? А потому, что вторым вариантом применения в функции total/1 является total([]) -> 0;

```
4> shop1:total([{milk,3}]).
21
```

Вызов функции total([{milk,3}]) подошел клаузе total([{What,N}|T]} в которой

T = [] (поскольку [X]=[X|[]]). После успешного сопоставления, переменные функции принимают следующие значения {What -> milk, N -> 3, T -> [ ]}. Затем выполняется тело клаузы функции (shop:cost(What) * N + total(T)). Все переменные заменяются на присвоенные им значения. Таким образом, тело функции выглядит как shop:cost(milk) * 3 + total([ ]).

shop:cost(milk) равно 7, а total([]) равно 0; следовательно, значение этого выражения равно 7*3+0 = 21.

Когда мне ставить точку с запятой?

В Эрланге мы используем три типа пунктуации:

- Запятые (,) разделяют аргументы в вызовах функции, конструкторах данных и шаблонах.
- Точки (.) (с последующим пробелом) разделяют функции и выражения в оболочке Эрланга.
- Точка с запятой (;) разделяет клаузы, которые мы используем в различных контекстах: в объявлении функций, а так же в блоках **case**, **if**, **try..catch** и в выражениях **receive**.

Всякий раз когда мы имеем набор шаблонов, с последующими за ними выражениями, мы используем точку с запятой как их разделитель:

```
Pattern1 ->
    Expressions1;
Pattern2 ->
    Expressions2;
...
```

Как насчет более сложных аргументов?

```
5> shop1:total([{pears,6},{milk,3}]).
75
```

На этот раз в первой клаузе переменные связываются со следующими значениями: {What -> pears, N -> 6, T -> [{milk,3}]}. В результате: shop:cost(pears) * 6 + total([{milk,3}]), или же 9 * 6 + total([{milk,3}]).

Но мы уже знаем, что результатом работы total([{milk,3}]) будет 21, поэтому результат равен 9*6 + 21 = 75.

И, наконец:

```
6> shop1:total(Buy).
123
```

Перед тем как закончить этот раздел, давайте более детально рассмотрим функцию total. Функция total(L) работает в зависимости от результата анализа аргумента L. Тут возможно два варианта: или L - это не пустой список, или же наоборот. Мы реализовали клаузы для каждого возможного варианта, вот так:

```
total([Head|Tail]) ->
    some_function_of(Head) + total(Tail);
total([]) ->
    0.
```

В нашем случае, Head была шаблоном {What,N}. Первая клауза срабатывает в случае, когда в нее передается непустой список. Она отделяет от него первый элемент ("голову"), что-то с ним делает, и потом функция вызывает саму себя для оставшейся части списка ("хвоста"). Вторая клауза срабатывает, когда полученный список пуст ([]).

На самом деле, функция total/1 делает две разные вещи. Во-первых, она находит цену каждого элемента в списке, и, потом, складывает все их значения. Мы можем переписать функцию total таким образом, что бы разделить получение значения каждого элемента и сложение этих значений. Получившийся код будет более ясным и простым для понимания. Чтобы сделать это, мы напишем две небольшие функции для работы с списками и назовем их sum и map. Но до того, как мы поговорим об этом, давайте немного познакомимся с анонимными функциями (funs). После этого мы напишем наши функции sum и map, а затем улучшим нашу функцию total.

Функции с одинаковыми именами и различной арности (arity)

Арность функции - это количество аргументов, принимаемых этой функцией. В Эрланге, две функции,объявленные в одном модуле, с одним именем, но разным количеством аргументов, представляют собой две различные функции. Они не имеют

ничего общего, кроме как одинакового имени.

По общепринятому соглашению, Erlang программисты часто используют функции с одинаковыми именами, но разным количеством аргументов, как вспомогательные функции. Например:

```
sum(L) -> sum(L, 0).

sum([], N) -> N;

sum([H|T], N) -> sum(T, H+N).
```

Функция sum(L) складывает элементы списка L. Она делает это используя вспомогательную функцию sum/2, хотя, она могла бы быть названа как угодно иначе. Вы можете назвать вспомогательную функцию hedgehog/2, и принцип работы программы не изменится. Но sum/2 это более лучший вариант для имени такой функции, так как он дает читателю вашей программы подсказку о ее содержании, а так же избавляет от необходимости придумывать новое имя для функции (что, как известно, не просто).

Анонимные функции(Funs)

funs ("фаны") это “анонимные” функции. Они называются так, потому что у них нет имени. Давайте немного поэкспериментируем. Для начала объявим функцию и сопоставим ее с переменной Z:

```
1> Z = fun(X) -> 2*X end.
#Fun<erl_eval.6.56006484>
```

Когда мы объявляем анонимную функцию, оболочка Erlang выводит

Fun<...>, где ... это какое-то странное число. Но здесь это не важно.

Сейчас мы можем сделать с анонимными функциями только одну вещь, а именно, применить ее к списку аргументов, например:

2> Z(2).

Z не самое лучшее имя для функции; более лучшим вариантом будет имя Double, так как оно описывает то, что функция делает:

3> Double = Z.

#Fun<erl_eval.6.10732646>

4> Double(4).

8

Анонимная функция может принимать любое количество аргументов. Мы можем написать функцию вычисления гипотенузы прямоугольного треугольника следующим образом:

5> Hypot = fun(X, Y) -> math:sqrt(X*X + Y*Y) end.

#Fun<erl_eval.12.115169474>

6> Hypot(3,4).

5.00000

Если количество аргументов будет неправильным, вы получите сообщение об ошибке:

7> Hypot(3).

** exited: {{badarity,{#Fun<erl_eval.12.115169474>,[3]}},

[{erl_eval,expr,3}]} **

Почему эта ошибка называется badarity? Вспомните, что арностью называется количество аргументов принимаемых функцией. badarity означает, что Erlang не может найти функцию с передаваемым именем, которая принимает переданное количество параметров - наша функция принимает 2 аргумента, а мы передаем только один.

Анонимная функция может также иметь различные варианты применения (клаузы). Приведем функцию которая может конвертировать значения температуры из шкалы Фаренгейта в шкалу Цельсия и наоборот:

8> TempConvert = fun({c,C}) -> {f, 32 + C*9/5};

8> ({f,F}) -> {c, (F-32)*5/9}

8> end.

#Fun<erl_eval.6.56006484>

9> TempConvert({c,100}).

{f,212.000}

10> TempConvert({f,212}).

{c,100.000}

11> TempConvert({c,0}).

{f,32.0000}

Обратите внимание: Выражение в строке 8 занимается несколько строк. После того, как мы начали вводить выражение, оболочка повторяет приглашение "8>" каждый раз как мы вводим новую строку. Это значит, что выражение не закончено, и оболочка ожидает продолжение ввода.

Эрланг - это функциональный язык программирования. Кроме всего прочего это означает, что анонимные функции могут быть переданы как аргументы для функции, а также, что функции (или анонимные функции) могут возвращать анонимные функции в качестве результата.

Функция, которая возвращает другие функции, или же может принимать другие функции в качестве своих аргументов, называется *функцией высшего порядка*. Мы увидим несколько примеров таких функций в следующих разделах.

Все это пока может звучать не так восхитительно, поскольку мы еще не видели, что можно делать с анонимными функциями. Пока код анонимных функций выглядел точно также, как и код обычных функций в модуле, но, как правило, это не так. Функции высшего порядка это очень существенная часть функциональных языков программирования - они просто зажигают огонь в теле кода. И когда вы научитесь ими пользоваться - вы полюбите их. В будущем мы встретимся с ними в огромном количестве.

Функции, принимающие другие функции в качестве своих аргументов

Модуль lists, входящий в стандартные библиотеки Эрланга, экспортирует несколько функций, которые принимают другие функции в качестве аргументов. Наиболее полезная из них это функция lists:map(F, L). Она возвращает список, созданный применением функции F к каждому элементу из списка L:

12> L = [1,2,3,4].

[1,2,3,4]

13> lists:map(Double, L).

[2,4,6,8].

Другая полезная функция - lists:filter(P, L), она возвращает новый список из таких элементов E списка L, для которых функция P(E) равна true.

Давайте создадим функцию Even(X), которая возвращает true, если X - это четное число:

14> Even = fun(X) -> (X rem 2) =:= 0 end.

#Fun<erl_eval.6.56006484>

Здесь X rem 2 вычисляет остаток от деления X на 2, а оператор =:= сравнивает его с нулем. Теперь мы можем использовать функцию Even как аргумент функция map и filter:

15> Even(8).

true

16> Even(7).

false

17> lists:map(Even, [1,2,3,4,5,6,8]).

[false,true,false,true,false,true,true]

18> lists:filter(Even, [1,2,3,4,5,6,8]).

[2,4,6,8]

Мы называем функции, такие как map и filter, которые делают что-либо со списком за один вызов функции, *список-за-раз* (list-at-a-time) операциями. Использование таких операций делает наши программы более короткими и простыми для понимания. И более простыми они становятся потому, что мы можем обработать каждый элемент списка за один логический шаг нашей программы. Иначе нам пришлось бы считать элементарным шагом нашей программы каждую индивидуальную операцию, над каждым элементом списка.

Функции, возвращающие функции

Функции могут использоваться не только в качестве аргументов других функций (таких как map и filter). Функции могут также *возвращаться* другими функциями.

Приведем пример - допустим, что у нас есть список чего-либо, предположим фруктов:

```
1> Fruit = [apple,pear,orange].

[apple,pear,orange]
```

Теперь объявим функцию MakeTest(L), которая преобразует любой список (L) в функцию, которая проверяет, находится ли ее аргумент в этом списке L:

```
2> MakeTest = fun(L) -> (fun(X) -> lists:member(X, L) end) end.

#Fun<erl_eval.6.56006484>

3> IsFruit = MakeTest(Fruit).

#Fun<erl_eval.6.56006484>
```

lists:member(X, L) возвращает true если X находится в списке L, в противном случае она возвращает false. Давайте протестируем нашу функцию:

```
4> IsFruit(pear).

true

5> IsFruit(apple).

true

6> IsFruit(dog).

false
```

Также мы можем использовать ее как аргумент функции lists:filter/2:

```
7> lists:filter(IsFruit, [dog,orange,cat,apple,bear]).

[orange,apple]
```

Описания анонимных функции, которые сами возвращают анонимные функции, требуют некоторого времени, чтобы освоиться с ними. Так что давайте немного отделим, для ясности, нотацию описаний от того, что происходит на самом деле.

Функции, возвращающие "нормальные" значения, выглядят следующим образом:

```
1> Double = fun(X) -> ( 2 * X ) end.

#Fun<erl_eval.6.56006484>

2> Double(5).
```

10

Код в круглых скобках (2 * X) это фактически "возвращаемое значение" функции. Теперь давайте попробуем поместить анонимную функцию внутрь круглых скобок. Помним, что выражение внутри скобок - это возвращаемое значение:

3> Mult = fun(Times) -> (fun(X) -> X * Times end) end.

#Fun<erl_eval.6.56006484>

Функция fun(X) -> X * Times end внутри скобок - это просто функция от X, но как в ней появилась переменная Times? Ответ: это просто аргумент "внешней" анонимной функции.

Вызов Mult(3) *вернет* fun(X) -> X * 3 end, что является телом внутренней (обычной) функции, в которой переменная Times заменена на число 3. Давайте это протестируем:

4> Triple = Mult(3).

#Fun<erl_eval.6.56006484>

5> Triple(5).

15

Таким образом, функция Mult это *обобщение (generalization)* функции Double. Она, вместо вычисления значения, *возвращает функцию, которая вычисляет требуемое значение*.

Объявление собственных управляющих абстракций

Подождите секундочку - вы заметили это? До сих пор нам так и не попалось никаких выражений с **if**, **switch**, **for** или **while** и это казалось нам совершенно нормальным. Все было написано с использованием сопоставления шаблонам и с помощью функций высшего порядка. До сих пор нам так и не потребовались другие управляющие программные структуры.

Ну, а если нам потребуются дополнительные управляющие структуры, то у нас есть супер-мощное средство для их создания. Давайте рассмотрим пример, как это делается: в Эрланге не существует (совсем) цикла **for** , так давайте сделаем его:

/ файл lib_misc.erl /

for(Max, Max, F) -> [F(Max)];

for(I, Max, F) -> [F(I)|for(I+1, Max, F)].

Здесь, например, вычисление for(1,10,F) создаст список [F(1), F(2), ..., F(10)].

Как же работает сопоставление по шаблону в этом цикле for? Первый вариант применения for срабатывает только тогда, когда первый и второй аргументы for одинаковы. Поэтому, если мы вызовем for(10,10,F) то переменной Max сопоставится 10, а результатом этого вызова будет [F(10)]. Если мы вызовем for(1,10,F), первый вариант применения for не подойдет, так как Max не может быть одновременно и 1 и 10, а второй вариант применения for сопоставит I ->1 и Max -> 10. А значением функции тогда будет [F(I)|for(I+1,10,F)], где I заменяется на 1, а Max заменяется на 10, что составит [F(1)|for(2,10,F)].

Таким образом, у нас появился простой цикл for . (Это не совсем тоже самое, что for в императивных языках программирования, но для наших целей его достаточно.) Мы можем использовать его для создания списка целых чисел от 1 до 10:

1> **lib_misc:for(1,10,fun(I) -> I end).**

[1,2,3,4,5,6,7,8,9,10]

Или мы можем его использовать для вычисления квадратов целых от 1 до 10:

2> **lib_misc:for(1,10,fun(I) -> I*I end).**

[1,4,9,16,25,36,49,64,81,100]

Когда используются функции высшего порядка?

Как мы видели, используя функции высшего порядка, мы можем создавать наши собственные, новые управляющие абстракции, можем передавать функции в качестве аргументов и мы можем создавать функции, возвращающие абстрактные функции. Но не все эти возможности используются на практике достаточно часто:

Практически все модули, которые я написал, использовали функции типа list:map/2 настолько часто, что я уже практически считаю функцию map просто частью языка Эрланг. Обращение к функциям map, filter и partition из модуля lists встречается очень часто.

Иногда я создавал свои собственные управляющие абстракции. Хотя это встречается гораздо реже, чем вызов высших функций из стандартных библиотечных модулей. Такое бывает до нескольких раз, для большого модуля.

Написание функций, которые возвращают абстрактные функции я использовал крайне редко. Если бы я писал сотню модулей, то эта техника встречалась бы скорее всего,

только в одном или двух из них. Программы с функциями, возвращающими абстрактные функции бывает весьма трудно отлаживать. Но, с другой стороны, мы можем использовать такие функции для реализации таких вещей, как ленивые вычисления, и мы можем легко создавать возвратные парсеры и комбинаторы парсеров, являющиеся функциями, которые возвращают парсеры.

(примечание переводчика: Ж8-О)

По мере накопления вашего опыта, вы можете обнаружить, что возможность создавать свои собственные управляющие структуры может феноменально сократить размер ваших программ и, иногда, сделать их гораздо понятнее. Поскольку теперь вы можете создавать управляющие структуры конкретно под решение ваших проблем и поскольку теперь вы не ограничены небольшим и фиксированным набором управляющих структур вашего языка программирования.

Стандартные ошибки

Некоторые читатели, часто пытаются, по ошибке, набрать примеры, приводимые в листингах модулей в этой книге, в оболочке Эрланга. Зря, это вовсе не команды для оболочки. Вы получите за это очень странные сообщения об ошибках. Поэтому мы вас предупреждаем: не делайте этого.

Если вы случайно выберете имя для своего модуля, которое совпадает с одним из системных модулей, тогда, после компиляции вашего модуля, вы получите странное сообщение, утверждающее, что вы не можете загрузить модуль из такой-то директории. Просто переименуйте ваш модуль (не забудьте про директиву -module(...) внутри файла модуля) и удалите все beam файлы, которые вы могли создать при компиляции вашего первоначального модуля.

3.5 Простая обработка списков

Теперь, когда мы познакомились с анонимными функциями, мы можем вернуться к написанию sum и map, которые нам потребуются для улучшения total (о которой, я уверен, вы еще не забыли!).

Мы начнем с функции sum , которая вычисляет сумму элементов в списке:

/файл mylists.erl/

```
sum([H|T]) -> H + sum(T);

sum([]) -> 0.
```

Отметим, что, в данном случае, порядок написания двух клауз данной функции не важен, поскольку, первая клауза срабатывает только на непустые списки, а вторая

только на пустой список. То есть они - взаимно исключающие.

Теперь мы можем протестировать функцию sum :

1> **c(mylists).** %% <-- Last time I do this

{ok, mylists}

2> **L = [1,3,10].**

[1,3,10]

3> **mylists:sum(L).**

14

В строке 1 я скомпилировал модуль lists. Но с этого момента, я буду, обычно, не показывать команду компиляции модуля, и вам нужно будет не забывать делать это самостоятельно.

Далее - все просто. Давайте рассмотрим выполнение sum по шагам:

\1. sum([1,3,10])

\2. sum([1,3,10]) = 1 + sum([3,10]) (by 1)

\3. = 1 + 3 + sum([10]) (by 1)

\4. = 1 + 3 + 10 + sum([ ]) (by 1)

\5. = 1 + 3 + 10 + 0 (by 2)

\6. = 14

И, наконец, давайте посмотрим map/2 , с которой мы уже встречались ранее. Вот как она определяется:

/файл mylists.erl/

map(_, []) -> [];

map(F, [H|T]) -> [F(H)|map(F, T)].

1 строка - Первая клауза этой функции говорит нам о том, что она делает с пустым списком. Применение любой функции к пустому списку (в котором ничего нет!) дает нам снова пустой список.

2-я строка - Вторая клауза описывает, что происходит с непустым списком, состоящим

из головы H и хвоста T. Это тоже просто. Получается новый список с головой F(H) и хвостом map(F,H) .

Примечание: Данное определение функции map/2 скопировано нами из модуля lists стандартной библиотеки в модуль mylists. С модулем mylists вы можете делать все, что угодно. Но ни при каких обстоятельствах не пытайтесь создать свой собственный модуль с именем lists, если только вы абсолютно уверены в том, что вы делаете.

Мы можем поработать с map , используя парочку функций, которые удваивают или возводят в квадрат элементы из списка, следующим образом:

1> **L = [1,2,3,4,5].**

[1,2,3,4,5].

2> **mylists:map(fun(X) -> 2*X end, L).**

[2,4,6,8,10]

3> **mylists:map(fun(X) -> X*X end, L).**

[1,4,9,16,25]

И на этом все, о функции map? Ну, на самом деле, не совсем. Позднее, мы покажем ее еще более короткую версию, написанную с использованием обработчиков списков. А в разделе 20.2 *Распараллеливание последовательного кода* , мы покажем, как можно вычислять все элементы получаемого после map списка *параллельно* (что ускорит выполнение нашей программы на многоядерном компьютере), но это мы сейчас забежали очень и очень далеко вперед. Сейчас, мы лишь можем только переписать функцию total с помощью этих двух функций:

/файл shop2.erl /

```
-module(shop2).

-export([total/1]).

-import(lists, [map/2, sum/1]).

total(L) ->

sum(map(fun({What, N}) -> shop:cost(What) * N end, L)).
```

Мы можем проверить, как она работает на следующем примере:

1> **Buy = [{oranges,4},{newspaper,1},{apples,10},{pears,6},{milk,3}].**

[{oranges,4},{newspaper,1},{apples,10},{pears,6},{milk,3}]

2> **L1=lists:map(fun({What,N}) -> shop:cost(What) * N end, Buy).**

[20,8,20,54,21]

3> **lists:sum(L1).**

123

Как я пишу свои программы

Когда я пишу программу, то я пользуюсь подходом: "напиши чуть-чуть - потом протестируй это". Я начинаю с маленького модуля, состоящего всего из нескольких функций, потом я компилирую его и тестирую с помощью нескольких команд в оболочке Эрланга. Когда он меня вполне удовлетворяет, я дописываю в нем еще несколько функций, компилирую его и тестирую. И так далее.

Часто я не представляю себе заранее, какие именно структуры данных мне потребуются в моей программе, но по мере того, как я попробую несколько простых примеров, я уже могу оценить, какие подходящие варианты из них мне следует выбрать.

Я предпочитаю, скорее "наращивать" свои программы, а не продумывать их заранее полностью, до их написания. При таком способе, я, скорее всего, не сделаю заранее крупной ошибки, до того, как обнаружу, что все пошло не так. И, кроме того, так гораздо веселее. Я немедленно получаю обратную связь и вижу, что мои идеи начинают работать, как только я реализовал их в программе.

А когда я понимаю, как что-то можно сделать в командной оболочке, то, часто я иду и пишу для данного случая make-файл и, возможно, немного кода, которые воспроизводят то, что я научился делать в оболочке Эрланга.

Обратите, также, внимание на использование объявлений **-import** и **-export** в этом модуле:

Объявление -import(lists, [map/2, sum/1]). означает , что функции map/2 и sum/1 *импортируются* из модуля lists. Это означает, что мы можем писать map(Fun, ...) вместо lists:map(Fun, ...) . А поскольку функция cost/1 не была указана в объявлении -import, то нам придется использовать ее "полное" имя shop:cost .

Объявление -export([total/1]) означает, что функция total/1 может вызываться снаружи данного модуля shop2. Только проэкспортированные так функции могут вызываться снаружи модулей.

Возможно, сейчас вам кажется, что нашу функцию total больше уже улучшить нельзя, но вы не правы. Это вполне возможно. Для этого мы используем обработчики списков (list comprehension).

3.6 Обработчики списков

Обработчики списков - это выражения, которые создают списки без использования анонимных функций, отображений (maps) или фильтров. Это делает нашу программу еще проще и доступнее для понимания.

Мы начнем с небольшого примера. Предположим, что у нас имеется список L:

1> **L = [1,2,3,4,5].**

[1,2,3,4,5]

И предположим, что мы хотим удвоить каждый элемент в данном вписке. Мы это уже делали раньше, но я вам напомню:

2> **lists:map(fun(X) -> 2*X end, L).**

[2,4,6,8,10]

Но существует гораздо более легкий способ сделать это, используя обработчики списков:

4> **[2*X || X <- L].**

[2,4,6,8,10]

Такая запись вида [F(X) || X <- L] означает "список из элементов F(X), где X берутся из списка L ". Следовательно, запись [2*X || X <- L] означает "список значений 2*X, где X берутся из списка L ".

Чтобы увидеть, как пользоваться обработчиками списков, мы можем ввести несколько команд в оболочку Эрланга и посмотреть, что из этого получится: Мы начнем с определения списка Buy (покупки):

1> **Buy=[{oranges,4},{newspaper,1},{apples,10},{pears,6},{milk,3}].**

[{oranges,4},{newspaper,1},{apples,10},{pears,6},{milk,3}].

А теперь, давайте удвоим число покупок по каждой из позиций из исходного списка:

2> **[{Name, 2*Number} || {Name, Number} <- Buy].**

[{oranges,8},{newspaper,2},{apples,20},{pears,12},{milk,6}]

Отметим дополнительно, что набор (тьюпл) {Name, Number} с права от знака "||" - это *образец* (паттерн), которому сопоставляется каждый элемент из списка Buy. А набор (тьюпл) слева от знака "||" - {Name, 2*Number} - называется* конструктор.*

**

Предположим теперь, что мы хотим посчитать общую стоимость всех элементов в исходном нашем списке. Это можно сделать следующим образом: Вначале, заменим имя каждой из покупок на ее цену:

3> **[{shop:cost(A), B} || {A, B} <- Buy].**

[{5,4},{8,1},{2,10},{9,6},{7,3}]

Теперь перемножим эти значения:

4> **[shop:cost(A) * B || {A, B} <- Buy].**

[20,8,20,54,21]

А теперь, просуммируем их:

5> **lists:sum([shop:cost(A) * B || {A, B} <- Buy]).**

123

И, наконец, если мы хотим превратить все это в одну функцию, мы можем написать следующее:

total(L) ->

lists:sum([shop:cost(A) * B || {A, B} <- L]).

Таким образом, вы видите, что обработчики списков действительно могут сделать ваш код очень коротким и понятным. Мы даже можем с их помощью, чисто ради развлечения, дать более короткое определение функции map:

map(F, L) -> [F(X) || X <- L].

В самом общем виде выражение для обработчика списков выглядит следующим образом:

[X || Qualifier1, Qualifier2, ...]

где X - это произвольное выражение, а каждый из определителей (Qualifier) это либо генератор, либо фильтр. При этом:

Генераторы записываются в виде Образец <- Списочное Выражение, где СписочноеВыражение - это любое выражение, дающее на выходе список термов Эрланга.

Фильтры - это либо предикаты (функции, которые возвращают либо true (истина) либо false (ложь)), либо же просто логические выражения.

Заметьте, что генератор в обработчике списков тоже работает, как своеобразный фильтр. Например:

1> **[X || {a, X} <- [{a,1},{b,2},{c,3},{a,4},hello,"wow"]].**

[1,4]

Мы закончим этот раздел, посвященный обработчикам списков, несколькими небольшими примерами:

Быстрая сортировка

Вот как можно написать алгоритм сортировки, используя всего лишь два обработчика списков:

```
/ файл lib_misc.erl /

qsort([]) -> [];

qsort([Pivot|T]) ->

qsort([X || X <- T, X < Pivot])

++ [Pivot] ++

qsort([X || X <- T, X >= Pivot]).
```

Здесь ++ - это инфиксный оператор добавления. А Pivot переводится как "центр вращения".

(Данный код приведен здесь, скорее, из-за своей элегантности, чем своей эффективности. Использование оператора ++ таким образом, вообще-то, не считается хорошей практикой программирования.)

Пример:

1> **L=[23,6,2,9,27,400,78,45,61,82,14].**

[23,6,2,9,27,400,78,45,61,82,14]

2> **lib_misc:qsort(L).**

[2,6,9,14,23,27,45,61,78,82,400]

Чтобы понять, как работает эта функция мы рассмотрим это все по шагам: Для начала у нас есть список L и мы вызываем qsort(L). Срабатывает вторая клауза функции qsort:

3> [Pivot|T] = L.

[23,6,2,9,27,400,78,45,61,82,14]

которая связывает переменные Pivot -> 23 и T -> [6,2,9,27,400,78,45,61,82,14].

Теперь мы разделяем список T на два списка : один из элементов, которые меньше чем Pivot, а второй - из элементов, которые больше или равны Pivot:

4> **Smaller = [X || X <- T, X < Pivot].**

[6,2,9,14]

5> **Bigger = [X || X <- T, X >= Pivot].**

[27,400,78,45,61,82]

Теперь мы можем отсортировать списки Smaller и Bigger и соединить их обратно с Pivot:

qsort([6,2,9,14]) ++ [23] ++ qsort([27,400,78,45,61,82])

= [2,6,9,14] ++ [23] ++ [27,45,61,78,82,400]

= [2,6,9,14,23,27,45,61,78,82,400]

Тройки Пифагора

Тройки Пифагора - это такие наборы из трех натуральных чисел {A,B,C} для которых верно равенство: .

Нижеследующая функция pythag(N) генерирует список всех натуральных чисел {A,B,C}, для которых выполняется указанное равенство и сумма которых меньше или равна N:

```
/ файл lib_misc.erl /

pythag(N) ->

[ {A,B,C} ||

A <- lists:seq(1,N),

B <- lists:seq(1,N),
```

C <- lists:seq(1,N),

A+B+C =< N,

A*A+B*B =:= C*C

].

Небольшие пояснения: функция lists:seq(1, N) возвращает нам список из всех целых чисел от 1 до N. Следовательно, запись A <- lists:seq(1, N) означает, что A принимает все значения от 1 до N. И, значит, нашу программу можно прочитать следующим образом: "Возьми все значения A от 1 до N, все значения B от 1 до N и все значения C от 1 до N? такие что A+B+C менее или равно N и A*A + B*B = C*C."

1> **lib_misc:pythag(16).**

[{3,4,5},{4,3,5}]

2> **lib_misc:pythag(30).**

[{3,4,5},{4,3,5},{5,12,13},{6,8,10},{8,6,10},{12,5,13}]

Анаграммы

Если вы интересуетесь традиционными Английскими словарными играми, то вы несомненно знакомы с разгадыванием анаграмм. Давайте используем Эрланг для нахождения всех перестановок строки символов, с помощью симпатичной, маленькой функции perms следующего вида:

/ файл lib_misc.erl /

perms([]) -> [[]];

perms(L) -> [[H|T] || H <- L, T <- perms(L--[H])].

Здесь "--" это оператор вычета одного списка из другого. Он вычитает все элементы второго списка из первого. Его более точное определение дано в разделе 5.4 *Операции со списками* ++ и -- .

1> **lib_misc:perms("123").**

["123","132","213","231","312","321"]

2> **lib_misc:perms("cats").**

["cats", "cast", "ctas", "ctsa", "csat", "csta", "acts", "acst",

"atcs", "atsc", "asct", "astc", "tcas", "tcsa", "tacs", "tasc",

"tsca", "tsac", "scat", "scta", "sact", "satc", "stca", "stac"]

Я не буду объяснять как работает функция perm , поскольку это объяснение получится гораздо длиннее, чес сама запись этой функции. Так что, вы можете разобраться с этим сами! (Но я дам Вам намек: чтобы вычислить все перестановки X123, вам надо вычислить все перестановки 123 (что составит [123, 132, 213, 231, 312, 321], а теперь вам надо вставить X во все возможные места каждой из полученных перестановок. Например, добавление X к 123 даст нам X123 1X23 12X3 123X; добавление X к 132 даст нам X132 1X32 13X2 132X и так далее. Примените это правило рекурсивно.)

3.7 Арифметические выражения

Все возможные арифметические выражения в Эрланге приведены ниже в таблице 3.1. В ней каждый арифметический оператор имеет один - два аргумента, которые могут быть либо целым, либо числом (что означает либо целое, либо вещественное число).

Также, с каждым оператором ассоциирован его *приоритет*, от которого зависит порядок исполнения сложных арифметических выражений: сначала выполняются все операторы с приоритетом 1 (слева направо), потом - с приоритетом 2 и так далее.

Оператор

Описание

Типы

аргументов

Приоритет

+X

+X

Число

1

-X

-X

Число

1

X*Y

X * Y

Число

2

X/Y

X/Y (деление с дробной частью)

Число

2

bnot X

побитовое отрицание X

Целое

2

X div Y

целочисленное деление X на Y

Целое

2

X rem Y

целый остаток от деления X на Y

Целое

2

X band Y

побитовое И между X и Y

Целое

2

X + Y

X + Y

Число

3

X - Y

X - Y

Число

3

X bor Y

побитовое ИЛИ между X и Y

Целое

3

X bxor Y

побитовое исключающее ИЛИ

Целое

3

X bsl N

Арифметический сдвиг битов X влево на N бит

Целое

3

X bsr N

Арифметический сдвиг битов X вправо на N бит

Целое

3

Таблица 3.1 : Арифметические выражения

Для изменения порядка исполнения арифметических операций можно использовать круглые скобки. Операторы с одинаковым приоритетом считаются лево-ассоциативными и исполняются слева направо.

3.8 Контролеры (Guards)

Контролеры позволяют нам еще более усилить мощь механизма сопоставления по образцу. Используя их, мы можем дополнительно проверять и сравнивать переменные из образца. Предположим, что мы хотим написать функцию max(X,Y), которая должна вычислять максимум из X и Y. С помощью контролеров, мы можем написать ее следующим образом:

```
max(X, Y) when X > Y -> X;

max(X, Y) -> Y.
```

Первая клауза этой функции срабатывает когда X больше Y и возвращает нам X.

А, если первая клауза не срабатывает, то в дело вступает вторая клауза, которая нам просто всегда возвращает Y. При этом Y должен быть больше или равно X, поскольку, иначе, сработала бы первая клауза этой функции.

Вы можете использовать контролеры и при определении функции, в их заголовках, где они следуют за ключевым словом **when**. Либо же вы можете их использовать в любом месте программы, где можно использовать выражение. И если контролеры используются как выражение, то они вычисляются до одного из атомов: true или false. Считается, что если вычисление контролера привело к true, то его вычисление прошло *успешно*. В противном случае - *неудачным*.

Последовательности контролеров

Последовательностью контролеров называют либо одиночного контролера, либо последовательность контролеров, разделяемых точкой с запятой (;). Значение последовательности контролеров G1; G2; ...; Gn считается равным true (истина), если хотя бы один из контролеров G1, G2, ... принимает значение true.

А теперь: *контролером* в Эрланге называют последовательность *контрольных выражений,* разделенных запятыми. Последовательность контрольных выражений считается истинной тогда и только тогда, когда все истинны все контрольные выражения входящие в нее.

Множество выражений, которые могут входить в контрольные выражения, несколько меньше, чем все доступные выражения в Эрланге. И причиной такого ограничения для контрольных выражений является необходимость гарантировать, что их вычисление будет абсолютно не иметь побочных эффектов. Контролеры являются частью

выражений сопоставления по образцу и, поскольку, оно не имеет побочных эффектов, они нам также не нужны и при вычислении контрольных выражений.

Кроме того, контролеры не могут быть логическими выражениями, определяемыми пользователями, поскольку нам опять таки нужны гарантии, что в них не будет побочных эффектов и, что они завершат свое выполнение.

Следующие синтаксические формы языка Эрланг допустимы в контрольных выражениях:

атом true (истина)

Другие константы (термы и связанные переменные) все они заменяются на false (ложь) в контрольных выражениях.

Вызовы контрольных предикатов из нижеследующей Таблицы 3.2 и встроенные функции (BIF) из Таблицы 3.3.

Сравнения термов (Таблица 5.3)

Арифметические выражения (Таблица 3.1, ранее)

Логические выражения (Раздел 5.4 *Логические выражения)*

Упрощенные (или укороченные) логические выражения (Раздел 5.4. подраздел *Упрощенные логические выражения*)

При вычислении контрольных выражений выполняются правила старшинства операторов, описанные в разделе 5.4, подразделе *Старшинство операторов*.

Примеры контролеров

f(X,Y) when is_integer(X), X > Y, Y < 6 -> ...

Эта запись означает, что "когда X это целое число и X больше чем У и У меньше чем 6, то ...". Запятая, разделяющая отдельные проверки в контролере означает логическое "И".

is_tuple(T), size(T) =:= 6, abs(element(3, T)) > 5

element(4, X) =:= hd(L)

...

Первый контролер проверяет, что X это либо cat (кошка), либо dog (собака). Второй контролер проверяет, что, либо X - это целое число, которое больше чем У; либо, что абсолютное значение У меньше чем 23.

А вот несколько примеров контролеров использующих упрощенные (или укороченные) логические выражения:

A >= -1.0 andalso A+1 > B

is_atom(L) orelse (is_list(L) andalso length(L) > 2)

Для опытных: Общей причиной разрешения использовать логические выражения в контролерах, было стремление сделать их синтаксически похожими на все прочие выражения. А причина для введения операторов orelse и andalso состоит в том, что операторы and/or изначально были определены через обязательное вычисление обоих своих аргументов. Однако, в контролерах может быть разница при использовании and и andalso, а также or и orelse. Например, рассмотрим следующие контролеры:

f(X) when (X == 0) or (1/X > 2) ->

...

g(X) when (X == 0) orelse (1/X > 2) ->

...

Контролер для f(x) выдаст ошибку, когда X будет нулем, но сработает для g(x).

Но на практике, весьма малое количество программ использует сложные контролеры, а для решения большинства проблем, вполне хватает простых (,) контролеров.

Предикат

Его значение

is_atom(X)

X - это атом

is_binary(X)

X - бинарная последовательность

is_constant(X)

X - константа

is_float(X)

X - это действительное число

is_function(X)

X - это функция

is_function(X, N)

X - это функция с числом аргументов (арностью) равным N

is_integer(X)

X - это целое число

is_list(X)

X - это список

is_number(X)

X -это целое или действительное число

is_pid(X)

X - это идентификатор процесса (PID)

is_port(X)

X - это порт

is_reference(X)

X - это ссылка

is_tuple(X)

X - это кортеж

is_record(X,Tag)

X - это запись с типом Tag

is_record(X,Tag,N)

X - это запись с типом Tag и размера N

Таблица 3.2. Предикаты используемые в контролерах

Использование контролера true (истина)

Во-первых, вы можете спросить: а зачем вообще нужен такой контролер true? Причина здесь в том, что атом true удобно использовать для контролера "все прочие" в конце выражения if . Приблизительно вот так:

if

Guard -> Expressions;

Guard -> Expressions;

...

true -> Expressions

end

Само выражение if будет обсуждаться нами в разделе 3.10 *Выражение if* .

Устаревшие контрольные выражения

Если вы столкнетесь со старым кодом на Эрланге, написанным несколько лет назад, то вы можете обнаружить, что тогда проверки в контролерах были несколько иными. Тогда в контролерах использовались проверки atom(X), constant(X), float(X), integer(X), list(X), number(X), pid(X), port(X), reference(X), tuple(X), и binary(X). Эти проверки означают тоже самое, что и современные их варианты типа is_atom(X) и так далее. Использование старых названий в современном коде не рекомендуется.

Функция

Значение функции

abs(X)

Абсолютное значение X.

element(N, X)

Элемент номер N из кортежа X.

float(X)

Конвертация числа X в действительное число.

hd(X)

Голова списка X.

length(X)

Длинна списка X.

node()

Данный узел Эрланга.

node(X)

Узел на котором X был создан, где X это может быть процесс, идентификатор, ссылка или порт.

round(X)

Конвертирует число X в целое число.

self()

Идентификатор этого (данного) процесса.

size(X)

Размер X, где X - это кортеж или бинарная последовательность.

trunc(X)

Обрезает число X до целого (отбрасывает дробную часть).

tl(X)

Хвост списка X.

Таблица 3.3. Встроенные функции для контролеров

3.9 Записи

Когда мы используем в нашей программе кортежи, мы можем столкнуться с трудностями, если число элементов в этих кортежах станет, вдруг, очень большим. Тогда становится достаточно трудно помнить, какой элемент в кортеже что означает. А записи позволяют привязать к каждому элементу кортежа его собственное имя, что решает данную проблему.

В маленьких кортежах таких сложностей обычно не возникает и, поэтому, мы часто видим программы, которые манипулируют небольшими кортежами, не вызывающих сомнений в предназначении своих элементов.

Записи объявляются с помощью следующего синтаксиса:

```
-record(Name, {

%% the next two keys have default values
```

key1 = Default1,

key2 = Default2,

...

%% The next line is equivalent to

%% key3 = undefined

key3,

...

}).

Предупреждение: record это вовсе не команда для оболочки Эрланга (используйте в ней команду rr, описанную чуть ниже). Определение записи может быть использовано только в исходном коде Эрланга, но не в его командной оболочке.

В вышеуказанном примере, Name - это имя всей этой записи. key1, key2 и так далее - это имена полей этой записи. Все эти имена должны быть атомами Эрланга. При этом, key1 и key2 имеют значения по-умолчанию (Default1 и Default2, соответственно), которые присваиваются этим полям при создании новой записи Name, если для них не указано другого значения. Поле key3 является изначально неопределенным полем записи.

Предположим, например, что мы хотим создать что-то вроде списка дел на будущее. Мы начнем с определения записи todo и сохраним ее в файле. Определения записей могут быть либо сразу включены в файлы с исходным кодом Эрланга, либо помещены в файлы с расширением .hrl и потом включены в файлы с исходным кодом (что является единственным способом, чтобы в разных Эрланг-файлах было одно и тоже определение этих записей).

/ файл records.hrl /

-record(todo, {status=reminder,who=joe,text}).

Как только запись была определена, мы можем создавать ее представителей в программе.

Чтобы сделать это в командной оболочке Эрланга, нам надо, сначала, прочитать определение записи в оболочку с помощью команды rr (сокращение от read record (прочитать запись)):

1> rr("records.hrl").

[todo]

Создание и изменение записей

Теперь мы готовы создавать записи и манипулировать ими:

2> X=#todo{}.

#todo{status = reminder,who = joe,text = undefined}

3> X1 = #todo{status=urgent, text="Fix errata in book"}.

#todo{status = urgent,who = joe,text = "Fix errata in book"}

4> X2 = X1#todo{status=done}.

#todo{status = done,who = joe,text = "Fix errata in book"}

В строках 2 и 3 мы *создали* новые записи. С помощью выражения вида #todo{key1=Val1,..., keyN=ValN} можно создавать новые записи todo , но, при этом все атомы key1,...,keyN должны быть такими же, как и в определении записи. Если какое-то из полей записи пропущено при ее создании, то этому полю присваивается значение по-умолчанию из определения записи (или undefined, если его там не было).

В строке 4 мы *скопировали* существующую запись. Синтаксис X1#todo{status=done} означает: создать копию записи X1 (которая должна быть типа todo) и изменить в ней значение поля status на done (сделано). Обратите внимание, что это будет только *копия*. Исходная запись при этом не изменится.

Извлечение значений полей из записей

Как и для всего прочего, для этого используется сопоставление по образцу:

5> **#todo{who=W, text=Txt} = X2.**

#todo{status = done,who = joe,text = "Fix errata in book"}

6> **W.**

joe

7> **Txt.**

"Fix errata in book"

Как вы видите, с левой стороны оператора сопоставления по образцу (=) мы написали подобие нашей записи с несвязанными переменными W и Txt. Если этот оператор

завершится успешно (т.е произойдет сопоставление) данные переменные окажутся связанными с соответствующими значениями полей в нашей записи. При этом, если нам надо только одно значение поля записи, то мы можем использовать более простой синтаксис "с точкой":

8> **X2#todo.text.**

"Fix errata in book"

Сопоставление по образцу записей в функциях

Мы можем писать функции, которые сопоставляют по образцу поля записей, либо которые создают новые записи. Для этого, обычно, используется следующий код:

clear_status(#todo{status=S, who=W} = R) ->

%% Inside this function S and W are bound to the field

%% values in the record

%%

%% R is the *entire* record

R#todo{status=finished}

%% ...

(В комментариях тут написано: Внутри этой функции переменные S и W будут связаны со значениями указанных полей переданной в функцию записи.)

Чтобы сопоставляться с записями определенного, нужного нам типа, мы можем написать определение функции подобно следующему:

do_something(X) when is_record(X, todo) ->

%% ...

Данная клауза функции сработает только когда X - это запись типа todo.

Записи - это замаскированные кортежи

На самом деле записи - это просто кортежи. Давайте заставим оболочку Эрланга забыть про определение записи todo:

11> X2.

#todo{status = done,who = joe,text = "Fix errata in book" }

12> rf(todo).

ok

13> X2.

{todo,done,joe,"Fix errata in book" }

В строке 12 мы скомандовали оболочке забыть определение записи todo. Поэтому теперь, когда мы пытаемся напечатать X2, оболочка показывает ее как кортеж. Внутри программы все представлено только в виде кортежей. Записи - это всего лишь удобная их форма, в которой вы можете дать имена различным элементам кортежа.

3.10 Выражения case и if

До сих пор, мы использовали для решения *всех* своих задач только механизм сопоставления по образцу. Это делает Эрланг компактным и последовательным. Но, иногда, определять различные клаузы в функциях для каждого случая бывает довольно неудобно. В этом случае мы можем воспользоваться выражениями case и if.

Выражение case

Выражение case имеет следующий синтаксис:

case Expression of

Pattern1 [when Guard1] -> Expr_seq1;

Pattern2 [when Guard2] -> Expr_seq2;

...

end

Оно вычисляется следующим образом. Во-первых вычисляется Expression, предположим, что при этом оно принимает значение Value. Далее Value сопоставляется с Pattern1 (вместе с опциональным контролером Guard1), Pattern2 и так далее, до первого успешного сопоставления. Как только это случится, вычисляется соответствующая последовательность выражений (Expr_seqN) и результат этого вычисления становится результатом всего данного case-выражения. Если не один из паттернов не подойдет, то происходит исключительная ситуация.

Ранее мы уже рассматривали функцию filter(P,L). Она возвращает список элементов X из списка L для которых P(X) истинно. Если использовать только сопоставление по образцу, то мы можем определить filter следующим образом:

```
filter(P, [H|T]) -> filter1(P(H), H, P, T);

filter(P, []) -> [].

filter1(true, H, P, T) -> [H|filter(P, T)];

filter1(false, H, P, T) -> filter(P, T).
```

Но такое определение довольно некрасиво, так как нам пришлось изобретать еще одну функцию filtr1 и передавать ей все аргументы filter/2.

Но мы можем сделать это гораздо более простым образом, используя конструкцию case следующим образом:

```
filter(P, [H|T]) ->

case P(H) of

true -> [H|filter(P, T)];

false -> filter(P, T)

end;

filter(P, []) ->

[].
```

Выражение if

Также в Эрланге имеется и вторая условная конструкция if. Вот ее синтаксис:

```
if

Guard1 ->

Expr_seq1;

Guard2 ->

Expr_seq2;

...

end
```

Она вычисляется следующим образом: Сначала вычисляется контролер Guard1. Если его значение равно true, то тогда значение всего выражения if получается вычислением последовательности выражений Expr_seq1. Если же контролер Guard1 не

сработал, то вычисляется Guard2 и так далее, пока кто-то из них на примет значение Истина (true). Если же такового не найдется, то будет поднято исключение.

Часто последним контролером выражения if бывает атом true, который "пропускает всех", тем самым гарантируя, что, по крайней мере одна из последовательностей выражений будет вычислена, даже если все остальные охранники не сработают.

3.11 Построение списков в естественном порядке

Самым эффективным способом построения списков является добавление новых его элементов в его голову, и, поэтому, мы часто можем увидеть код приблизительно такого вида:

some_function([H|T], ..., Result, ...) ->

H1 = ... H ...,

some_function(T, ..., [H1|Result], ...);

some_function([], ..., Result, ...) ->

{..., Result, ...}.

Этот код проходит по списку, берет его голову H и вычисляет от нее какое-то значение определяемое в данной функции (мы назвали его H1). Потом H1 добавляется к итоговому списку Result.

Когда исходный список закончится, сработает финальная клауза данной функции и итоговая переменная Result будет возвращена из данной функции.

Но элементы в списке Result будут находится в обратном порядке к породившем их элементам в исходном списке. Для некоторых ситуаций это вовсе не проблема, но если, все таки, это так, то они легко могут быть переставлены на заключительном шаге.

Основная идея очень проста:

Всегда добавляйте новые элементы в голову списку.

Когда вы берете исходные элементы из головы исходного списка и и добавляете их (или результаты их обработки) в голову результирующего списка, вы получаете список в обратном порядке к исходному.

Но, если вам важен правильный порядок следования элементов, используйте функцию list:reverse/1 которая реализована крайне оптимально (в смысле скорости работы).

Избегайте использования других рекомендаций, кроме этих.

Примечание: Когда бы вы не захотели обратить список вспять, вы должны пользоваться функцией list:reverse и ничем иным. Если вы захотите посмотреть на ее исходный код в модуле lists, то там тоже будет и ее определение. Но учтите, оно приведено там только для иллюстрации. Компилятор, когда он встречается с вызовом lists:reverse, обращается к гораздо более эффективной внутренней версии этой функции.

Как только вы увидите код, подобный нижеследующему:

```
Lists ++ [H]
```

у вас в голове должен сработать сигнал тревоги, поскольку это очень не эффективный способ, который приемлем, только если список List очень короткий.

3.12 Аккумуляторы

Как нам получить из одной функции два списка? Как нам написать функцию, которая разделяет список целых чисел на два, в которых будут только четные и нечетные числа из исходного списка? Вот один из способов, как это можно сделать:

```
/ файл lib.misc.erl /

odds_and_evens(L) ->

Odds = [X ll X <- L, (X rem 2) =:= 1],

Evens = [X ll X <- L, (X rem 2) =:= 0],

{Odds, Evens}.

5> lib_misc:odds_and_evens([1,2,3,4,5,6]).

{[1,3,5],[2,4,6]}
```

Но проблема с этим кодом состоит в том, что мы проходим по исходному списку дважды, что не очень страшно, когда он короткий, но если он очень длинный - это может стать проблемой.

Чтобы избежать этого двойного прохождения по списку, мы можем переписать наш код следующим образом:

```
/ файл lib.misc.erl /

odds_and_evens_acc(L) ->
```

```
odds_and_evens_acc(L, [], []).

odds_and_evens_acc([H|T], Odds, Evens) ->

case (H rem 2) of

1 -> odds_and_evens_acc(T, [H|Odds], Evens);

0 -> odds_and_evens_acc(T, Odds, [H|Evens])

end;

odds_and_evens_acc([], Odds, Evens) ->

{Odds, Evens}.
```

Теперь мы проходим по списку только один раз и добавляем его четные и нечетные элементы в их собственные выходные списки (которые мы называем аккумуляторами). У этого кода, кроме того, есть еще одно преимущество, которое гораздо менее очевидно: версия с аккумуляторами гораздо более эффективна в смысле использования памяти, чем версия с конструкциями типа [H || filter(H)].

Если мы запустим это, то мы получим почти такой же результат, как и ранее:

```
1> lib_misc:odds_and_evens_acc([1,2,3,4,5,6]).

{[5,3,1],[6,4,2]}
```

Разница тут в том, что элементы в четном и нечетном списках здесь в обратном порядке. Это является следствием того, как эти списки были получены. Если нам нужны эти списке в порядке следования элементов в исходном списке, нам всего лишь надо сделать реверсирование списка в финальной клаузе нашей функции следующим образом:

```
odds_and_evens_acc([], Odds, Evens) ->

{lists:reverse(Odds), lists:reverse(Evens)}.
```

Чему нам удалось научиться

Теперь мы умеем создавать модули Эрланга и писать на нем простые последовательные программы. А также, мы почти уже готовы к написанию на нем более сложных последовательных программ.

Однако, следующая глава данной книги посвящена краткому рассмотрению темы работы с ошибками в Эрланге. После нее мы снова вернемся к последовательному программированию и рассмотрим в нем все, пока оставленные нами, детали.