Руководство по программированию на языке Erlang

Язык программирования Erlang относится к функциональным языкам с динамической типизацией. Язык позволяет создавать параллельные процессы, связанные механизмом обмена сообщениями и сигналами выхода. По синтаксису Erlang не похож ни на один существующий язык. В книгах по программированию обычно первым делом излагают так называемую философию языка. По моему, сначала лучше познакомиться с техникой и синтаксисом, а потом уж говорить о философии. Без всяких общих рассуждений предлагаю сразу перейти к знакомству с базовыми конструкциями языка.

1.Основы

Интерактивный режим

Как и другие языки, Erlang позволяет использовать интерактивный режим при разработке и отладке программ (часто этот режим называют словом Repl). Этот режим также оригинален по своим возможностям и по применению, как и сам Erlang. Для вызова repl надо в командной строке выполнить команду ***erl*** (или ***werl***). После команды ***erl*** появится приглашение в виде:

1>

Здесь ***1*** означает номер команды, которую мы будем выполнять. Как мы позже увидим, это номер можно использовать в дальнейшем. После приглашения можно вводить команды, которые будут немедленно выполняться с выдачей результата. Каждая команда должна заканчиваться точкой. После точки в той же строке можно вводить другие команды, все они будут выполнены по очереди.

Если вместо ***erl*** выполнить команду ***werl***, появиться новое окно с некоторыми дополнительными функциями. Можно использовать это окно вместо командной строки, если нравится (я предпочитаю командную строку).

Выход из Repl возможен по команде ***ctrl/c*** или с использованием специальных функций ***q().*** и ***halt()***. Допустима ещё команда ***ctrl/G***, которая возвращает счётчик выполненных команд на ***1*** и отменяет все определения. (Все строчки кода в Repl должны заканчиваться точкой).

Функция ***pwd().*** Позволяет узнать текущую директорию, то-есть, ту, из которой был выполнен вход в Repl. Функция ***ls().*** Выводит список всех файлов текущей директории. Есть много других команд.

В интерактивном режиме можно выполнять выражения и вызывать функции, например:

(2+4)*7. → 42

(Стрелку (→) я буду дальше использовать везде вместо слов «получим», «будет равно» и тому подобное. Сам Repl этой стрелки не выводит.)

math:sin(0.5+0.7). → 0.93203908596722 63

Здесь мы вызвали стандартную функцию ***sin*** из модуля ***math***. Значит, имя функции отделяется от имени модуля двоеточием (в других языках обычно применяется точка). Числа с плавающей точкой в Erlang вычисляются с удвоенной точностью (В других языках это тип ***Double***). Дальше увидим, что создаваемые нами функции вызываются точно также, как стандартные.

Функция ***v(n).*** позволяет вызвать результат ***n***-ой строки интерактивного режима. Этот результат можно использовать в дальнейших вычислениях:

1> ***9/2. → 4.5***

2> 3*(8-4). → 12

3> v(1)*v(2). → 54.0

4> 12*4.5. → 54.0

Аргумент у функции ***v*** может быть отрицательным, тогда номер строки будет отсчитываться назад от номера текущей строки.

Нет необходимости отдельно рассматривать тему арифметических операций, здесь всё, как обычно. Отмечу только, что оператор ***div*** возвращает целую часть частного без округления:

19 div 5. → 3

Это же делает и функция ***trunc***:

trunc(19/5). → 3

Оператор ***rem*** возвращает остаток от деления:

19 rem 5. → 4

Функция ***round*** позволяет выполнять округление числа:
round(19/5). → 4

Repl позволяет объявлять с одновременной инициализацией переменные, идентификаторы которых всегда должны начинаться с буквы в верхнем регистре:
X = 123.
Y = math:cos(0.95).

Чтобы вывести значение переменной достаточно написать её имя и поставить точку:
Y. → 0.5816830894638835

Все переменные в Erlang не изменяемы (immutable), поэтому нельзя им присвоить новое значение:
X = 87. - здесь будет ошибка.

На самом деле в Erlang не выполняется присваивание, вместо этого выполняется одновременно два действия: указанная переменная связывается (bound) со значением и выполняется сопоставление с образцом (pattern matched). Поэтому можно сколько угодно раз повторять
X = 123.
Здесь сопоставление удачно и ошибки не будет. Ошибки также не будет, если написать:
123 = 123.

В документации различают не связанные (свободные) и связанные переменные. Нельзя использовать в выражении не связанную ни с чем (свободную) переменную и нельзя связанную переменную связать снова с каким-то другим значением. Отсюда и проистекает immutable в Erlang. Дальше мы ещё вернёмся к этой теме. Впрочем, очень часто связывание можно рассматривать просто, как инициализацию переменной.

Отметим ещё, что в Repl можно выполнять только выражения, но нельзя создавать функции, кроме как только через анонимные функции. Но об этом позже.

Функции

Erlang язык функциональный, а в таких языках функция — это основной элемент программы. Применяется такая терминология.

Erlang-программы состоят из нескольких параллельных процессов. Процессы выполняют функции, определённые в модуле. Модули это файлы с расширение ***.erl***, которые должны быть скомпилированы раньше их запуска. Скомпилированные модули можно выполнять из Repl, или прямо из командной строки. О параллельных процессах мы ещё будем говорить позже. Впрочем, дело не в терминологии, всё будет ясно из дальнейших примеров.

Для начала по традиции создадим функцию, возвращающую приветствие «Hello World». Итак, создадим файл ***prob.erl*** (далее я всё время буду применять это название файла, сохранять эти ничтожно малые фрагменты кода нет никакого смысла). Поместим этот файл туда, где нравится и запишем в него такой код:

```
-module(prob).
-export([f/0]).
f() -> io:format("Hello, World!~n").
```

Первые две строчки обязательны. В первой после служебного слова ***module*** в скобках надо указать имя файла без расширения. Во второй — после слова **export** в круглых и во вложенных квадратных скобках надо поместить имя создаваемой функции, а через косую черту (прямой слеш) указать количество аргументов (арность функции). Квадратные скобки нужны потому, что, как мы увидим далее, здесь на самом деле список (или массив), в котором может быть объявлено несколько функций. Затем надо указать имя функции, список аргументов (пустые скобки обязательны), стрелку (->) и после неё тело функции, в общем случае состоящее из последовательности выражений (terms), разделённых запятыми. Каждая строка заканчивается точкой. Как видим, техника создания модулей в Erlang предельно простая — каждый файл с программой является модулем и все функции, объявленные в операторе ***-export,*** доступны из других модулей и из Repl.

В документации операторы в виде слов с дефисом впереди называются атрибутами модуля (module attribute) и они определяют некоторые свойства модуля. Кроме атрибутов ***-module*** и ***-export*** есть и другие. Атрибуты можно даже создавать самостоятельно.

Тело функции ***f*** состоит из одного оператора вывода текста на экран. Используется функция ***format*** из модуля ***io***. По названию функции можно догадаться, что она позволяет форматированный

вывод, о нём поговорим позже. Сочетание знаков (***~n***) это управляющий символ перевода строки (в других языках вместо (**~**) обычно применяют (/)). Значит, в текстовой константе в двойных кавычках функция ***format*** распознаёт управляющие символы.

Теперь файл ***prob.erl*** можно скомпилировать. Для этого надо перейти в ту директорию, где расположен наш файл, вызвать командой ***erl*** интерактивный режим и выполнить команду:

c(prob).

Если компиляция без ошибок, то будет получен отклик:

{ok,prob}

В результате компиляции в той же директории буде создан дополнительный файл под именем ***prob.beam***. Бесполезно пытаться его распечатывать и анализировать, он содержит некоторый промежуточный код. Слово ***beam*** совпадает с названием виртуальной машины, умеющей выполнять эти файлы. Для вызова функции применяется уже знакомый нам синтаксис:

prob:f(). → Hello, World!

Здесь также при отсутствии аргументов пустые скобки обязательны. Модули Erlang можно компилировать и непосредственно из командной строки (не из Repl). Работа из командной строки несколько сложнее, позже нам это потребуется, там и разберёмся.

Посмотрим теперь на пример функции с двумя аргументами:

```
-module(prob).
-export([f/2]).
f(X,Y) -> X*Y.
```

Вызываем функцию:

prob:f(3,7). → 21

Обычно принято использовать собственные, отражающие какой-то смысл, идентификаторы. Считаю, что это может быть полезно только для долго живущих объектов и в программах, предназначенных для повторного использования. В кратких примерах, когда функция или переменная существуют лишь на нескольких строчках это ничего не даёт, кроме ухудшения восприятия. Чаще всего я буду именовать функции одной буквой ***f***, а переменные буквами ***X, Y*** и так далее. Я вообще сторонник максимально краткого кода и, в частности, в большинстве случаев считаю комментарии внутри программного кода

в простых примерах бесполезными. Кстати, комментарии в Erlang начинаются с комбинации знаков ***%%***.

В функциональных языках циклы обычно создаются с помощью рекурсии, когда функция вызывает сама себя для повторного выполнения. Естественно, что Erlang позволяет рекурсивный вызов. Наверное самый популярный пример использования рекурсии это вычисление факториала числа. Посмотрите, как просто и элегантно программируется эта задача на Erlange:

```
-module(prob).
-export([fac/1]).
fac(1) -> 1;          --здесь должна быть точка с запятой!
fac(N) -> N*fac(N-1).
```

Программу компилируем и вызываем функцию в Repl:

```
prob:fac(5). → 120
```

Кажется, что в этой программе функция ***fac*** объявляется дважды. На самом деле здесь действует механизм сопоставления с образцом (***pattern matching)***, который имеет широчайшее применение в функциональном программировании. Если аргумент функции сопоставляется с указанным образцом, эта строка выполняется, в противном случае происходит переход на следующую строку. Каждая строка в такой конструкции называется функциональным выражением и отделяются они друг от друга точкой с запятой. Все функциональные выражения в совокупности можно считать объявлением функции и его код завершается точкой.

Здесь надо ещё раз обратить внимание на важный фактор, который часто забывают: в конце первой строки, то-есть, после первого функционального выражения ставится не точка, а точка с запятой. В реальной программе функциональных выражений может быть много и механическая привычка ставить везде точку часто служит досадной причиной ошибок.

Из примера видно, что тело функции не заключается в скобки. Тем не менее, Erlang позволяет создавать блоки, для чего используются ограничители ***begin – end***. Например:

```
X = begin Y=2, math:sin(Y) end. → 0.9092974268256817
```

Блоки позволяют группировать разные выражения, иногда это бывает полезно. Выражения в блоке разделяются запятыми, как и в теле функции. Блок всегда возвращает последнее вычисленное

значение. В Erlang всюду перед словом ***end*** никогда не ставится ни точка, ни запятая, ни точка с запятой.

Форматированный вывод

Поскольку дальше в примерах будем использовать вывод на терминал, рассмотрим здесь эту тему подробнее. Оператор вывода ***io:format*** на самом деле является функцией ***format*** из модуля ***io***. Эта функция принимает два аргумента: первый аргумент представлен текстом в двойных кавычках (дальше узнаем, что в Erlang такой текст почти тоже, что и список), второй аргумент представляет список — перечень элементов в квадратных скобках. В предыдущем примере строку вывода можно было бы записать так:
io:format("Hello, World!~n", [])
где ***[]*** - пустой список, который можно и не указывать. Внутри текста в двойных кавычках можно вставлять комбинацию знаков ***~w*** и тогда на место этих знаков при выводе будет вставлен элемент из списка:
io:format("результат: ~w~n ", [math:pow(3,2)]). → результат: 9.0

Значит, элементы списка могут быть представлены выражением. Можно делать несколько вставок:
io:format("Его зовут ~w, ему ~w лет~n", [ivan, 25]). →
Его зовут ivan, ему 25 лет

Имя ***ivan*** на латинице записано с маленькой буквы не случайно. Дело в том, что такой элемент списка рассматривается, как атом (смотрите далее), которые записываются с маленькой буквы. Если написать ***Ivan*** то это будет рассматриваться, как идентификатор переменной и зафиксируется ошибка — использование не инициированной переменной. Применение текста в двойных кавычках приводит к такому результату:
io:format("Его зовут ~w, ему ~w лет~n", ["Ivan", 25]). →
Его зовут [73,118,97,110], ему 25 лет

Можно применить вместо ***w*** букву ***p*** и тогда будет выведен текст:
io:format("Его зовут ~p, ему ~w лет~n", ["Ivan", 25]). →
Его зовут "Ivan", ему 25 лет

Но при этом выводятся кавычки. Применение буквы ***p*** даёт ещё и некоторые другие возможности, увидим их позже. Можно ещё использовать и букву ***s*** и тогда получим:

io:format("Его зовут ~s, ему ~w лет~n", ["Ivan", 25]). →
Его зовут Ivan, ему 25 лет

Можно также воспользоваться тем, что любой текст в одинарных кавычках в Erlang является атомом:

io:format("Его зовут ~w, ему ~w лет~n", ['Ivan', 25]). →
Его зовут 'Ivan', ему 25 лет

Но вывод атома будет тоже в одинарных кавычках. Можно даже написать ***'иван'***, но вообще-то, работа с кириллицей в Erlang представляет некоторую проблему.

Перед знаком ***w*** (а также и перед **p** и **s**) может стоять целое число, определяющие размер области в знаках для вывода данного элемента:

io:format("Его зовут ~9w, ему ~5w лет~n", ['Ivan', 25]). →
Его зовут 'Ivan', ему 25 лет

Выводимый текст прижимается к правому краю выделенной области. Чтобы прижать к левому краю, надо указать отрицательное число:

io:format("Его зовут ~-9w, ему ~-5w лет~n", ['Ivan', 25]). →
Его зовут 'Ivan' , ему 25 лет

При выводе чисел типа float приходится заботится о том, чтобы число поместилось в выделенную область и было округлено до приемлемой длины:

X=1/3. → 0.3333333333333333
io:format("X = ~10w~n", [round(X*1000)/1000]). → X = 0.333

Кроме функции ***format*** в том же модуле ***io*** имеется функция ***fwrite*** предоставляющая некоторые дополнительные возможности.

Атомы (atoms)

Атомы - особый вид объектов в Erlang, не имеющий аналогов в других языках. Это не переменные и не функции, а просто имена, используемые в специальных случаях. Атом всегда записывается с первой буквой в нижнем регистре. Атомы также можно записывать в одинарных кавычках. Вместо словесной характеристики лучше всего показать атомы на примерах. Напишем такую программу:

```
-module(prob).
-export([f/2]).
f(M, dm) -> M * 2.54;
```

f(N, sm) -> N / 2.54.

Здесь выполняется перевод значения длины, заданной в дюймах, в сантиметры, и наоборот. Функция ***f*** принимает два аргумента. Первый, ***M*** или ***N***, это заданная длина, а второй и есть этот самый атом. В примере атом указывает, в каких единицах будет задано исходное значение длины: ***dm*** означает дюймы, а ***sm*** – сантиметры. Вызов функции в интерактивном режиме будет выглядеть так:

c(prob). → {ok,prob}
prob:f(5, dm). → 12.7
prob:f(4.7, sm). → 1.8503937007874016

В примере снова действует сопоставление с образцом, а выбор нужного варианта здесь происходит по заданному атому. Разумеется, в обоих случаях для численной переменной можно было употребить один и тот же идентификатор, например ***M***, ничего бы не изменилось.

Динамическая типизация хороша тем, что избавляет нас от заботы о типах; о них вообще можно не думать. Все типы определяются (выводятся) на этапе runtime, кроме, конечно, тех случаев, когда это невозможно по смыслу. Посмотрим, например, что получится, если для функции ***f*** вместо числа ввести строку:

prob:f("abc", dm). →
an error occurred when evaluating an arithmetic expression in function prob:f/2 (prob.erl, line 3)

Компьютер нам сообщает, что ошибка произошла при попытке вычислить арифметическое выражение. Действительно, нельзя выполнить операцию умножения для текста. Кроме того, можно выполнить команду ***v(n)***. (здесь ***n*** - номер команды в Repl), и тогда получим более подробное сообщение об ошибке. Впрочем, дополнительная информация относится к обслуживающей программе и для нас пока мало интересна.

Кортежи (tuples)

Кортеж это упорядоченный набор элементов, заключённых в фигурные скобки. Количество элементов может быть любым, но чаще всего используются кортежи с двумя элементами. Обычно их называют парами, например: ***{22, 77}***. Предыдущий пример можно сделать более удобным, применив кортеж. Сделаем так, чтобы

единица измерения длины (***dm, sm***) указывалась не только для исходного значения, но также и для результата. Перепишем программу в таком виде:

```
-module(prob).
-export([f/1]).
f({X, sm}) -> {X / 2.54, dm};
f({X, dm}) -> {X * 2.54, sm}.
```

Теперь аргумент функции ***f*** представлен кортежем (поэтому количество аргументов указано равным единице: ***-export([f/1]).***) и возвращаемый функцией результат также кортеж:

```
prob:f({7, dm}). → {17.78,sm}
```

Работать с этой программой приятнее, не правда ли?

Заметьте, что первый элемент в паре представляет число, а второй — атом. Значит, элементы в паре могут быть разными (иметь разные типы). Кортеж может иметь вложенные пары, например температуру воздуха в разных городах можно было бы представить в виде следующих кортежей:

```
{moscow, {c, -10}}
{paris, {f, 28}}
```

Вторые элементы этих пар сами являются парами. Атомы ***c*** и **f** могут обозначать температуру по Цельсию и по Фаренгейту соответственно.

Встроенная функция ***tuple_to_list*** преобразует кортеж в список:

```
X = {a,d,c,d}.
Y = tuple_to_list(X). → [a,d,c,d]
```

А функция ***list_to_tuple*** выполняет обратное преобразование.

Кортежи позволяют групповое присваивание:

```
{X,Y,Z}={1,2,3}.
Y. → 2
Z. → 3
```

При этом отдельные элементы могут совпадать, компилятор соглашается с этим:

```
{A,0.25,B}={true,0.25,"Anna"}.
A. → true
B. → "Anna"
```

Этот приём позволяет извлекать любой элемент из кортежа, например:

{_,X,_}={true,0.25,"Anna"}.
X. → 0.25
Значит, ненужные элементы надо заметить знаком (_). Позже мы увидим, как это можно применить с пользой.

Встроенная функция ***element*** позволяет извлечь элемент кортежа по его номеру:
T = {2.7, august, "Bob"}.
element(2,T). → august
Значит, нумерация элементов начинается с единицы. Функция ***setelement*** позволяет заменить элемент по его номеру :
T1 = setelement(2,T,{hello, world}).
T1. → {2.7,{hello,world},"Bob"}

Второй элемент заменили на вложенный кортеж. Функция ***tuple_size*** позволяет узнать размер кортежа:
tuple_size(T1). → 3

Списки (Lists)

Списки это упорядоченные коллекции с произвольным числом элементов, заключённых в квадратные скобки. В интерактивном режиме объявление и инициализация списков может выглядеть так:
S1 = [1,2,3,4,5].
S2 = ["Anna","Marta","Peter"].

Прочитаем значение:
S2. → ["Anna","Marta","Peter"]

Допустимо присутствие в списке элементов разных типов:
S3 = ["Hello", 23.89, true].

Встроенная функция ***length*** позволяет узнать размер списка:
length(S3). → 3

Как уже упоминалось, любые переменные, включая списки, в Erlang неизменяемы (immutable) и, значит, нельзя список инициировать снова, например:
S3 = [1,2,3]. - здесь будет ошибка.
Неизменяемость — необходимое качество для любого полноценного функционального языка.

Erlang позволяет выделять первый элемент списка, для чего применяется такой синтаксис:

[A | B] = S1.
A. → ***1*** - это head
B. → ***[2,3,4,5]*** - это tail

Обычно первый элемент называют «голова» (head), а список, полученный после удаления первого элемента - «хвост» (tail).

Можно выделить сразу два (или более) элемента:

[A1, B1 | C] = S1.
A1. → ***1***
B1. → ***2***
C. → ***[3,4,5]***

Если в списке всего два элемента, tail будет представлять пустой список:

S3 = [5, 9].
[A2, B2 | C1] = S3.
A2. → ***5***
B2. → ***9***
C1. → ***[]***

Неизменяемость вынуждает вводить всё новые идентификаторы.

Есть также две встроенных функции ***hd*** и ***tl***, выделяющие голову и хвост списка соответственно:

hd([7, 3, 5, 9]). → ***7***
tl([7, 3, 5, 9]). → ***[3,5,9]***

Легко добавить новый элемент в начало списка:

S4 = [25|S1].
S4. → ***[25,1,2,3,4,5]***

Возможность выделения head и tail позволяет решать разнообразные задачи по обработке списков. Для тех, кто незнаком с функциональным программированием приведённая далее техника кажется необычной. На самом деле в таких программах используется сопоставление с образцом (pattern matching), если сопоставление удачно, строка выполняется, если неудачно — пропускается, а циклы в функциональном программировании создаются с помощью рекурсии. Например, составим программу для вычисления суммы элементов списка (в модуле ***lists*** имеется для этого готовая функция ***sum***):

-module(prob).
-export([f/1]).

f([]) -> 0;
f([A | B]) -> A + f(B).

Когда функции передаётся список, переменная ***A*** равна head, а переменная ***B*** – tail этого списка. Применяя рекурсию, мы выстраиваем цепочку суммирования: a1+a2+a3+...+an+0, где ai – элементы списка. Цепочка обрывается, когда tail становится пустым списком.

p***rob:f([1,2,3,4,5]). → 15***

Такой алгоритм не эффективен по затратам памяти. Этот недостаток можно устранить, применив так называемый аккумулятор — переменную, в которой накапливается сумма по ходу дела;

-module(prob).
-export([f/2]).
f([], S) -> S;
f([A | B], S) -> f(B, S + A).

Здесь аккумулятором служит переменная ***S***, которой надо передать начальное значение, равное нулю:

prob:f([1,2,3,4,5], 0). → 15

Если знак суммы (+) заменить на знак умножения (*****) и задать начальное значение аккумулятора равное ***1***, то получим произведение элементов массива:

-module(prob).
-export([f/2]).
f([], S) -> S;
f([A | B], S) -> f(B, S * A).
prob:f([1,2,3,4,5], 1). → 120

Если в цикле добавлять к сумме не head (у нас это переменная ***A***), а единицу, то получим длину списка (количество элементов):

-module(prob).
-export([f/2]).
f([], S) -> S;
f([A | B], S) -> f(B, S + 1).
prob:f([1,2,3,4,5], 0). → 5

Программа работает, но при этом выводится предупреждение о том, что объявленная переменная (речь идёт об ***A***) не используется. Чтобы избавиться от предупреждения, можно вместо **A** поставить знакозаменитель (placeholder), обозначаемый знаком (_). Такой

знакозаменитель в подобных ситуациях применяется практически во всех современных языках.

f([_ | B], S) -> f(B, S + 1).

Erlang не имеет данных типа string, строка рассматривается и обрабатывается, как список. Предыдущая программа, например, может вычислять длину строки:

prob:f("Hello World", 0). → 11

Списки, у которых элементы представлены целыми числами, соответствующими символам Unicode, рассматриваются, как строки:

X=[97,98,99].

X. → "abc"

Знак конкатенации в Erlang два плюса (++):

X="Hello ".

Y = "World!".

Z=X++Y.

Z. → "Hello World!"

Поскольку строки это списки, то этот же знак (++) выполняет и объединение двух списков в один:

A=[1,2,3].

B=["aa","bb"].

A++B. → [1,2,3,"aa","bb"]

Для строк знак конкатенации допускается опускать, при этом он подразумевается:

"Hello" "World". → "HelloWorld"

В противоположность знаку конкатенации (++) можно использовать знак (--) для удаления элементов из списка. Работает эта операция довольно своеобразно:

[***1,2,3,4,5] -- [2,4]. → [1,3,5]***

Удаляются заданные элементы независимо от места их расположения. Если есть повторяющиеся элементы, то удаляются те, которые встречаются первыми от начала списка:

[1,2,3,2,1,2] -- [2,1,2]. → [3,1,2]

Оператор (--) также работает и с текстами в двойных кавычках:

"Мама мыла раму" — "Мыа". → "ма мла раму"

В модуле ***lists*** есть функция ***reverse*** для реверса списков и строк. Для тренировки создадим такую функцию самостоятельно. Назовём её хотя бы ***rev***:

```
-module(prob).
-export([rev/1]).
rev([]) -> [];
rev([X | Y]) -> rev(Y) ++ [X].
```

Вызовем функцию для строки:

```
prob:rev("Hello"). → "olleH"
```

Или с использованием аккумулятора ***S***:

```
-module(prob).
-export([rev/2]).
rev([],S) -> S;
rev([A|B], S) -> rev(B, [A]++S).
```

Можно применять для строк и для списков:

```
prob:rev("Hello",""). → "olleH"
prob:rev([1,2,3,4,5], []). → [5,4,3,2,1]
```

Можно не применять операцию конкатенации, а поступить так:

```
-module(prob).
-export([f/2]).
f([],S) -> S;
f([A|B], S) -> f(B, [A|S]).
```

Такой вариант считается предпочтительнее, поскольку не создаётся каждый раз новый список, а используется одна и та же копия.

В следующем примере посмотрим, как можно использовать списки, кортежи и рекурсию совместно. Функцию я обозначил буквой ***g*** поскольку буква ***f*** использована, как атом для обозначения температуры по Фаренгейту.

```
-module(prob).
-export([g/1]).
g([{X, {f, F}} | Y]) -> T = {X, {c, (F -32)* 5 / 9}}, [T | g(Y)];
g([Z | Y]) -> [Z | g(Y)];
g([]) -> [].
```

Здесь функция ***g*** принимает список, который сразу разделяется на голову и хвост. По структуре головы видим, что элементы списка представлены кортежами. Конкретно первый член кортежа будет содержать название города, а второй сам представляет кортеж, состоящий из атома, задающего название шкалы температуры, и значения самой температуры по этой шкале. Использовано два атома:

c и ***f***, для температуры по Цельсию и по Фаренгейту. Если имеем атом ***f***, то значение температуры пересчитывается для шкалы ***c***. При этом использована локальная переменная ***T***. Если задан атом ***c***, то пересчёт не происходит. В программе есть один нюанс, на который надо обратить внимание. Функция ***g*** возвращает список и при рекурсивном вызове она накапливает элементы списка, а не заменяет их последовательно. Последняя строка служит для выхода из цикла по условию пустого списка. Вот как выглядит вызов функции ***g*** и возвращаемый ею результат:

prob:g([{moscow, {c, -10}}, {stockholm, {c, -4}}, {paris, {f, 28}}, {london, {f, 36}}]). →

[{moscow,{c,-10}}, {stockholm,{c,-4}}, {paris,{c,-2.222}}, {london,{c,2.222}}]

Значит, если исходная температура задана в градусах Цельсия, то это же значение и получаем. И только для температуры по Фаренгейту происходит перевод в шкалу по Цельсию.

Как и кортежи, списки позволяют выполнять групповое присваивание:

[X,Y]=[1,2].
X. → 1
Y. → 2

И также допустимы одинаковые элементы:

[X, 77] = [0.5, 77].
X. → 0.5

Функция ***seq*** из модуля ***lists*** позволяет создавать ранги в заданном диапазоне:

lists:seq(3,7). → [3,4,5,6,7]

С помощью placeholder можно применять, например, такой трюк. Пусть у нас есть список:

S = [1,2,3].

Тогда выражение:

S1 = [_,_,_|_] = S.

Вернёт нам исходный список:

S1. → [1,2,3]

А если мы напишем:

[_,_,_,_|_] = S. → no match of right hand side value [1,2,3]

То получим исключение, так как количество требуемых элементов больше длины списка ***S***. Это позволяет создать функцию, которая принимает список с числом элементов не меньше ***3***:
f([_,_,_|_] = S) -> ...
Если функции ***f*** передать список с двумя элементами, получим исключение. Таким образом, здесь для аргумента функции задано ограничение по длине списка без использования встроенной функции ***length.***

Отображения (Map, Hash)

Не установилось общепринятое наименование этих коллекций. Иногда их называют ещё ассоциированными списками и даже словарями. Для краткости дальше будем называть их словом «хеш». В Erlang для хешей принят следующий синтаксис:
M = #{"a" => 1, "b" => 2, "c" => 3}.

Значит, элементы хеша заключены в фигурные скобки, впереди которых ставится знак ***(#).*** Элементами являются пары, члены которых соединены знаком (=>). Первые члены пар называют ключами (***key***), а вторые — значениями (***value***). Ключи и значения могут быть любых типов:
M1 = #{12.03 => "alpha", "Marta" => 25, 123 => true}.

Среди ключей не может быть одинаковых по смыслу, значения же могут быть любыми. Если попытаться создать, например, такой хеш:
#{a=>1,b=>2,a=>3}. → #{a => 3,b => 2}
то последняя пара с ключом ***a*** аннулирует предыдущую.

В Erlang хеш всегда представлен отсортированным по ключам, независимо от того, в каком порядке располагались пары при его создании:
H1 = #{ b => 2, c=>3, a => 1 }.
H1. → #{a => 1,b => 2,c => 3}

Для хеша неприменим способ выделения head и tail, как для списков, а имеются свои приёмы для работы с элементами. В частности, можно изменить значение (value) для заданного элемента хеша. Применяется такой замысловатый синтаксис:
H = #{a=>1, b=>2, c=>3}.
H1 = H#{b := 77}.

H1. → #{a => 1,b => 77,c => 3}

При этом исходный хеш ***H*** конечно не изменяется, можно только создать новый хеш ***H1***. Новое значение должно иметь тот же тип. Как видим, здесь применяется знак ***(:=)*** вместо ***(=>)***.

Для добавления в хеш новых пар применяется такой синтаксис:

H1 = #{ c => 2, d=>3, a => 1 }.
H2=H1#{b=>77}. → #{a => 1,b => 77,c => 2,d => 3}

То-есть, для добавления надо вместо ***(=:)*** применить ***(=>)***. При этом опять получаем отсортированный хеш.

С помощью знака ***(:=)*** можно извлечь значение по ключу и одновременно инициировать этим значением какую-нибудь переменную:

H=#{a=>11, b=>22, c=>33}.
#{b:=X} = H.
X. → 22

Встроенная функция ***map_size*** позволяет узнать число пар в хеше (длину хеша):

map_size(H). → 3

Кроме того, в модуле ***maps*** имеется несколько функций для работы с хешами. Так функция ***get*** позволяет извлечь значение по ключу:

maps:get(b,H). → 22

Функция ***update*** позволяет изменить значение по ключу (вместо указанного выше способа):

maps:update(c,88,H). → #{a => 11,b => 22,c => 88}

Кроме этого, в модуле maps имеются ещё такие функции для работы с хеш: ***new, is_map, to_list, from_list, is_key, find, keys, remove, without, difference***. Смысл большинства из них понятен из названия.

Ветвления

Оператор if в Erlang похож на switch-case в других языках программирования. В общем виде его можно описать так:

```
if
   условие -> выражение;
   условие -> выражение;
   условие -> выражение
end
```

Каждая пара условие-выражение (будем дальше их называть сопоставлениями) заканчивается точкой с запятой, кроме последней перед служебным словом ***end***. Например:

X=2.
Y=if X<0 -> -1; X==0 -> 0; x>0 -> 1 end.
Y. → 1

Я записал все сопоставления в одной строке, но можно, конечно, и столбиком, как нравится. Вообще Erlang равнодушен к расположению выражений и операторов, главное, чтобы не было двусмысленностей.

Из примера видим, что оператор ***if*** возвращает значение. Условия проверяются последовательно до тех пор, пока одно из них не вернёт ***true***. Тогда вычисляется соответствующее выражение и возвращается результат, а последующие условия уже не проверяются. Если ни одно из условий не выполняется, будет зафиксирована ошибка. Чтобы исключить такой вариант, надо ввести такое условие, которое даст ***true*** в любом случае. Проще всего на место условия просто поставить это значение ***true***:

if X==7 -> "X=7"; true -> "X<>7" end.

В одном сопоставлении можно проверять несколько условий, которые перечисляются через запятую:

X=7. Y=9.
if X==7, Y==9 -> "Yes"; true -> "No" end. → "Yes"

При этом запятая означает логическое «***И***» - сопоставление удачно, если выполняются все (у нас два) условия. Но запятую можно заменить на точку с запятой, которая будет означать логическое «***ИЛИ***» - сопоставление удачно, если выполняется хотя бы одно условие:

if X==5; Y==6; Z==7 -> "Yes"; true -> "No" end. → "Yes"

Erlang имеет два сорта операторов сравнения, продемонстрируем их на примерах:

5 =:= 5 . → true
1 =:= 0 . → f a l s e
1 =/= 0 . → true
5 =:= 5 . 0 . → f a l s e
5 == 5 . 0 . → true
5 /= 5 . 0 . → f a l s e

Хотя в арифметических выражениях Erlang не делает различия

между целыми числами и числами с плавающей точкой, при сравнениях такое различие можно учесть, а можно и не учитывать. Отметим ещё, что оператор меньше-равно записывается так (=<), в остальном всё, как обычно. Отметим ещё, что Erlang позволяет сравнивать всё, что угодно:

5 =:= true . → f a l s e

Кроме оператора ***if*** Erlang имеет структуру ***case of***, позволяющую выполнять классическое pattern matching, но со значительно большими возможностями. Посмотрим на более сложном примере:

```
-module(prob).
-export([f/1]).
f(A) -> case A of
            {rectangle, {X, Y}} -> X * Y;
            {circle, R} -> math:pi() * R * R
        end.
```

Программа вычисляет площадь прямоугольника или круга. Нужный вариант выбирается по атому ***rectangle*** или ***circle***. Аргумент функции ***f*** представлен кортежем, причём в первом случае второй элемент кортежа тоже кортеж ***{X, Y}***, а во втором — переменная ***R***. Так будет выглядеть вызов функции:

```
prob:f({rectangle, {2, 7}}). → 14
prob:f({circle, 2}). → 12.566370614359172
```

При желании мы можем и результат получить в форме кортежа:

```
-module(prob).
-export([f/1]).
f(A) -> case A of
            {rectangle, {X, Y}} -> {rectangle, X * Y};
            {circle, R} -> {circle, math:pi() * R * R}
        end.
prob:f({rectangle, {2, 7}}). → {rectangle,14}
prob:f({circle, 2}). → {circle,12.566370614359172}
```

Макро-функции (macro)

Тему macro впоследствии рассмотрим отдельно. Пока ограничимся примером, который показывает, как macro позволяет создавать некие условные функции, проверяющие свои аргументы на соответствие

произвольным сложным условиям. Макро-функции создаются с помощью встроенной функции ***-define*** (записывается с дефисом), принимающей два аргумента: первый — идентификатор макро (некоторой условной функции) со своим списком аргументов, второй — проверяемые этой функцией условия. Например:

```
-module(prob).
-export([f/1]).
-define(g(V), (is_float(V) andalso V >= 0.0 andalso V =< 1.0)).
f(X) when ?g(X) -> math:asin(X).
```

Здесь создаётся макро ***g*** – функция с одним аргументом, который должен иметь тип ***float*** и принадлежать диапазону ***0 — 1***, включая крайние значения. Как видим, в общем случае условий может быть сколько угодно, синтаксически они разделяются служебным словом ***andalso***. Далее функция ***g*** используется для контроля переменной ***X*** - единственного аргумента функции ***f***. Для этого применяется служебное слово ***when***, после которого указывается имя макро-функции со знаком ***?*** впереди, которой передаётся этот аргумент ***X***. Если ***X*** не будет соответствовать заданным ограничениям, генерируется сообщение об ошибке.

```
prob:f(0.9). → 1.1197695149986342
prob:f(-0.5). → ошибка, число меньше нуля
prob:f(1). → ошибка, число не принадлежит типу float.
```

(В Erlang «меньше равно» обозначается =<, а «не равно» /=).

Проверять можно не одну переменную, а сколько угодно:

```
-module(prob).
-export([f/4]).
-define(g(V), (is_float(V) andalso V >= 0.0 andalso V =< 1.0)).
f(X1,X2,X3,X4) when ?g(X1), ?g(X2), ?g(X3), ?g(X4) -> [X1,X2,X3,X4].
C = prob1:f(0.3, 0.4, 0.7, 1.0).
C. → [0.3,0.4,0.7,1.0]
```

Здесь функция ***f*** возвращает свои четыре аргумента в форме списка, если все аргументы удовлетворяют заданным условиям.

После служебного слова ***when*** не обязательно должна быть макро-функция, вместо неё можно использовать условное выражение:

```
-module(prob).
-export([f/1]).
f(X) when X > 0 -> math:log(X).
```

prob:f(2.3). → ***0.832909122935103***
prob:f(-0.5). → no function clause matching prob:f(-0.5) (prob.erl, line 3)

Здесь я показал пример реального вывода сообщения об ошибке в подобных случаях.

Локальные функции

Функции, указанные в операторе ***-export*** глобальны — они доступны из других модулей и из Repl. Erlang позволяет применять и локальные функции, которые вне модуля невидимы. Объявляются локальная функция предельно просто, достаточно просто присвоить ей имя. Рассмотрим для примера программу по определению максимального элемента в списке. Отмечу сразу, что в модуле ***lists*** есть функция ***max*** решающая эту задачу:
lists:max([3,1,5,2]). → ***5***
Значит, пример будем рассматривать только для тренировки. Приведу вариант программы из документации Erlang:

```
-module(prob).
-export([f/1]).
f([H|T]) -> g(T, H).                (1)
g([], R) -> R;                      (2)
g([H|T], X) when H > X -> g(T, H);  (3)
g([_|T], X) -> g(T, X).             (4)
```

Чтобы прокомментировать то, что тут происходит, я пронумеровал строки. В строке (1) функция ***f*** только объявляет новую локальную функцию ***g***, передавая ей аргументы из полученного списка. Обычно число аргументов функции называют её арностью. Значит, функция ***f*** имеет арность, равную ***1***, а локальная функция ***g*** имеет арность ***2***. Этот приём потребовался для того, чтобы выделить первый элемент исходного списка (голову), чтобы с него начать сравнения элементов по величине. (Обращаю внимание на то, что эта строка заканчивается точкой).

Далее выполняются pattern matching с локальной функцией ***g***. Строка (2) выполняется, если первый аргумент — пустой список и тогда возвращается второй аргумент, как результат. Строка (3) выполняется, если голова (***H***) полученного списка больше второго аргумента (***X***). Здесь ***H*** будет представлять уже второй элемент

исходного списка, а ***X*** – первый элемент. Если это так, то рекурсивно вызывается функция ***g***, и первый её аргумент будет хвост исходного списка, а второй аргумент — второй (больший, чем первый) элемент списка. В противном случае выполняется строка (4), где рекурсивно вызывается функция ***g*** и первый аргумент - хвост исходного списка, а второй аргумент — первый элемент списка. Поскольку голова списка тут не используется, следует заменить её на placeholder. Организуется цикл, в котором вторым аргументом функции ***g*** всегда будет больший из двух первых элементов списка, который в свою очередь на каждом цикле будет укорачиваться на один элемент. В итоге список исчерпается и строка (2) вернёт нам результат.

Вообще-то локальной функции можно было бы присвоить тот же идентификатор ***f***, то-есть написать:

```
-module(prob).
-export([f/1]).
f([H|T]) -> f(T, H).
f([], R) -> R;
f([H|T], X) when H > X -> f(T, H);
f([_|T], X) -> f(T, X).
```

Хотя имя одно и то же, но это разные функции и компилятор распознаёт их по арности. Существует два мнения: одни считают, что не стоит вводить новые имена, если можно использовать одни и те же, другие утверждают, что применение новых имён улучшает читабельность программ. Мне кажется, каждый программист может писать программы по своему вкусу. Все переменные в Erlang неизменяемы. Может показаться, что в программе переменные H и T изменяют свои значения. На самом деле здесь одни и те же обозначения использованы в разных функциях, имеющих свои области видимости, а это значит, что это разные переменные с одинаковыми идентификаторами. Можно было бы, конечно, применить разные обозначения, например так:

```
-module(prob).
-export([f/1]).
f([H|T]) -> g(T, H).
g([], R) -> R;
g([H1|T1], X) when H1 > X -> g(T1, H1);
g([_|T2], X1) -> g(T2, X1).
```

Вообще-то никакой необходимости в локальной функции в этом примере нет, и мне больше нравится такой вариант:

```
-module(prob).
-export([f/1]).
f([H|[]]) -> H;
f([H1,H2|T]) when H1>H2 -> f([H1|T]);
f([_|T]) -> f(T).
```

Эта программа более лаконичная и намного проще предыдущей. По-моему, тут даже не нужны никакие комментарии. Отмечу только, что вторую строку можно было бы заменить на такую:

```
f([H1,H2|T]) when H1>H2 -> f([H1]++T);
```

И такой вариант возможно легче понять.

Анонимные функции

Как и все современные языки программирования, Erlang позволяет использовать анонимные функции (closure). Такие функции — полезное и необходимое средство для любого функционального языка. Реально эти функции мало отличаются от обычных, за исключением того, что они не имеют имени. Для создания анонимной функции применяется пара служебных слов ***fun – end***, например:

```
fun(X) -> X*2 end.
```

Эта функция умножает свой аргумент на два. Такую функцию можно вызвать, передав ей аргумент:

```
(fun(X) -> X*2 end)(7). → 14
```

Как увидим далее, во многих случаях не требуется вводить имя функции, но при желании это имя можно и ввести. Для этого достаточно просто инициировать какую-нибудь переменную анонимной функцией:

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

Теперь ***F*** – функция, которая от обычной отличается только тем, что её идентификатор начинается с буквы в верхнем регистре. Функцию ***F*** можно вызвать, как обычную:

```
F(7). → 14
```

Анонимные функции применяют в итераторах — операторах, позволяющих работать с элементами списков. Рассмотрим применение итератора ***foreach*** из модуля ***lists***. Итератор ***foreach***

принимает два аргумента: анонимную функцию и список. Итератор организует цикл, в котором применяет заданную функцию к каждому элементу списка. При этом итератор ничего не возвращает, кроме слова ***ok***, поэтому о выводе результата мы должны позаботиться сами:

S = [5,2,8,3].

lists:foreach(fun(X) -> io:format("~w*2 = ~w, " , [X,X*2]) end, S). →

5*2 = 10, 2*2 = 4, 8*2 = 16, 3*2 = 6, ok

Здесь первый аргумент итератора — анонимная функция, умножающая свой аргумент на два и выводящая результат на терминал, а второй — обрабатываемый список. Конечно, мы можем предварительно присвоить имя анонимной функции и тогда код станет более читабельным:

F = fun(X) -> io:format("~w*2 = ~w, " , [X,X*2]) end.

lists:foreach(F, S). → 5*2 = 10, 2*2 = 4, 8*2 = 16, 3*2 = 6, ok

Итератор ***map*** из того же модуля ***lists*** делает то же самое, что и итератор ***foreach***, и, кроме того, возвращает результат в виде нового списка:

lists:map(fun(X) -> 2*X end, S). → [10,4,16,6]

Посмотрим, как может выглядеть применение анонимной функции в модуле:

```
-module(prob).
-export([f/2]).
f(F,S) -> lists:map(F, S).
```

Вызываем функцию ***f*** и передаём ей анонимную функцию и список:

prob:f(fun(X) -> 3*X end, S). → [15,6,24,9]

Кроме итераторов ***foreach*** и ***map*** в модуле lists есть и другие. Например, итератор ***any*** принимает два аргумента: функцию, проверяющую некоторое заданное условие и список. Итератор ***any*** проверяет выполнение условия для каждого элемента списка и возвращает ***true***, если в списке есть хотя бы один элемент, удовлетворяющий условию. В противном случае получаем ***false***. Например:

F = fun(X) -> if X > 10 -> true; true -> false end end.

Эта функция возвращает ***true***, если её аргумент больше ***10***.

lists:any(F, [1,2,3,12,5]). → true

Аналогичный итератор ***all*** возвращает ***true***, если все элементы списка удовлетворяют заданному условию:

lists:all(F, [11,12,13,14,25]). → true

lists:all(F, [1,2,13,14,5,25]). → false

Итератор ***filter*** принимает такие же аргументы и возвращает список из всех элементов, удовлетворяющих условию:

lists:filter(F, [1,2,13,14,5,25]). → [13,14,25]

Для работы со списками в модуле ***lists*** есть ещё итераторы: ***foldl, mapfoldl, takewhile, dropwhile, splitwith***.

Одной и той же функции ***fun*** можно передать сразу несколько различных списков аргументов и, соответственно, различных тел анонимной функции, разделяя их точкой с запятой. Например, перевод температуры из одной шкалы в другую можно запрограммировать так:

F=fun({c,C}) -> {f, 32 + C*9/5}; ({f,F}) -> {c, (F-32)*5/9} end.

F({c,100}). → {f,212.0}

F({f,212}). → {c,100.0}

Нужный вариант выбирается по переданному атому: ***c*** или ***f.*** Списки аргументов и тела функции могут быть совершенно разными:

F=fun({c,C}) -> {f, 32 + C*9/5}; (X) -> X*X end.

F(7). → 49

Поскольку анонимными функциями можно инициировать переменные, то их можно и применять, как переменные: передавать другим функциям в качестве аргументов, возвращать в виде результатов, использовать в качестве локальных переменных. Дальше мы увидим применение анонимных функций в разных ситуациях, а здесь рассмотрим ещё пример, когда возвращаемый результат является функцией. Пусть у нас имеется такой список фруктов:

Fruit = [apple,pear,orange].

Создадим функцию ***IsFruit***, определяющую принадлежность заданного атома этому списку:

F = fun(L) -> (fun(X) -> lists:member(X, L) end) end.

IsFruit = F(Fruit).

Использованная здесь функция ***member(X, L***) из модуля lists определяет принадлежность переменной ***X*** списку ***L***. Значит, нужная нам функция ***IsFruit*** является результатом, возвращаемым функцией ***F.***

IsFruit(apple). → true
IsFruit(dog). → false

Конечно, от замены функции ***member*** на функцию ***IsFruit*** мы выгадали не слишком много.

Оператор цикла for

Erlang, как и другие функциональные языки, не имеет оператора for для организации циклов. Используя рекурсию, мы легко можем создать функцию ***for***, вполне соответствующую оператору цикла for. Кроме того, используя возможность передачи функций в качестве аргументов для других функций, можно функцию ***for*** снабдить ещё и дополнительными свойствами, например так:

```
-module(prob).
-export([for/3]).
for(Max, Max, F) -> [F(Max)];
for(I, Max, F)   -> [F(I)|for(I+1, Max, F)].
```

Здесь переменная ***I*** выполняет роль параметра цикла. При вызове функции ***for*** надо передать начальное значение этого параметра. Аргумент ***Max*** задаёт верхнюю границу для параметра ***I***. Аргумент ***F*** представляет функцию, определяющую те действия, которые надо выполнить с параметром ***I***, например:

prob:for(1, 10, fun(I) -> I*I end). → [1,4,9,16,25,36,49,64,81,100]

Здесь аргументу ***F*** передана анонимная функция, возводящая свой аргумент в квадрат. Выход из цикла происходит по условию достижения параметром ***I*** верхней границы ***Max***. Можно, конечно, поступать и так:

```
G=fun(X) -> 2*X end.
prob:for(1,10,G). → [2,4,6,8,10,12,14,16,18,20]
```

2.Параллельное программирование

Erlang, конечно, язык универсальный. Но одной из главных целей авторов языка было создание эффективного средства для параллельного программирования. Язык позволяет создавать одновременно работающие процессы (то же, что обычно называют

потоками) и взаимодействие между ними. К знакомству с этой техникой теперь и приступим.

Процессы

Для создания процесса в Erlang применяется встроенная функция ***spawn***, которая в свою очередь использует функцию, определяющую его функциональность, то-есть то, что процесс способен делать. Эта функция передаётся функции ***spawn***, как параметр, при этом название модуля, имя функции и список аргументов в квадратных скобках передаются, как три отдельных аргумента. Посмотрим на примере:

```
-module(prob).
-export([f/2,g/1]).
f(_, 0) -> io:format("~w~n", [done]);
f(X, T) -> io:format("~w~n", [X]),
f(X, T - 1).
g(N) ->
    spawn(prob, f, [hello, N]),
    spawn(prob, f, [goodbye, N]).
```

Здесь функция ***g*** создаёт два процесса на базе одной и той же функции ***f***, которая выводит на терминал значение первого аргумента (***X***), а второй её аргумент (***T***) задает число повторений цикла. Процессы отличаются друг от друга только списками аргументов функции ***f***.

```
prob:g(3). →
        hello
        goodbye
        hello
        goodbye
        <0.533.0>
        hello
        goodbye
        done
        done
```

Вызвав функцию ***g*** убеждаемся, что процессы работают параллельно (одновременно), поскольку слова ***hello*** и ***goodbye*** чередуются. Каждый процесс заканчивается выводом слова ***done,***

когда переменная ***T*** становится равна нулю. Число ***<0.533.0>*** является идентификатором (***pid***) процесса, который возвращает функция ***spawn***. Поскольку функция ***g*** возвращает последнее вычисленное значение, то это значит, что мы получили pid второго процесса. Вывод pid произошёл где-то на середине рекурсивного цикла функции ***f***. Это связано со временем — процессы уже сформировались, а функция ***f*** всё ещё выполняет свой цикл. Дальше мы увидим, как этот ***pid*** можно использовать.

Процесс может посылать сообщение другому процессу. Эта операция обозначается знаком (***!***) и имеет такой синтаксис:

A ! B

Здесь ***A*** – переменная, содержащая **pid** процесса-адресата, а ***B*** – переменная, содержащая текст сообщения. В качестве сообщения могут быть любые сущности языка Erlang (valid Erlang terms): атомы, числа, списки, closure и так далее.

Посмотрим на конкретном примере:

```
-module(prob).
-export([a/2,b/0,f/1]).
a(A, B) -> A ! B.
b() -> receive fin -> io:format("Stop~n");
                  C -> io:format("~p~n", [C])
       end.
f(X) ->  Bp = spawn(prob, b, []), spawn(prob, a, [Bp, X]).
```

Функция ***f*** создаёт два процесса на базе функций ***a*** и ***b***. Дальше будем называть процессы по имени используемых ими функций, то-есть наши процессы будем называть процесс ***a*** и процесс ***b***. Процесс ***b*** создаётся первым и его ***pid*** мы присвоили переменной ***Bp***. Эту переменную затем используем в качестве аргумента для функции ***a***. ***Pid*** процесса ***a*** нам не потребуется. Функция ***a*** имеет два аргумента: первый это ***pid*** процесса ***b***, второй — текст сообщения. Значит, функция ***a*** (или процесс ***a***) посылает сообщение процессу ***b***. Для получения и обработки сообщения процесс ***b*** использует блок, ограниченный служебными словами ***receive*** и ***end***. В этом блоке имеется возможность сопоставления полученного сообщения с образцом. Если сообщение будет содержать атом ***fin***, то процесс ***b*** выводит текст ***Stop***, а если сообщение будет каким-то другим, то процесс ***b*** выводит само это сообщение:

prob:f("Marta"). *→* ***"Marta" <0.522.0>***
prob:f(fin). *→* ***Stop <0.522.0>***

Как обычно, в Repl выводятся значения pid (***<0.522.0>*** и ***<0.522.0>***), дальше я их буду опускать. И надо сделать пару замечаний в качестве предостережения от ошибок. Процесс ***b*** создаётся первым и мы можем передать его ***pid*** процессу ***a***, то-есть имеет значение порядок создания процессов, их нельзя переставить местами. В блоке receive-end сопоставления разделяются точкой с запятой, а перед словом ***end*** этот знак не ставится. При этом объявленной здесь переменной ***C*** передается значение сопоставляемой величины, то-есть, текст сообщения. Переменная ***C*** локальна и вместо ***C*** можно использовать то же обозначение ***B***, ничего не изменится, всё-равно это будут разные переменные.

Посмотрим теперь, как будет процесс ***b*** обрабатывать серию сообщений от процесса ***a***:

```
-module(prob).
-export([a/2,b/0,f/1]).
a(A, []) -> A ! fin;
a(A, B) -> A ! hd(B), a(A, tl(B)).
b() -> receive fin -> io:format("Stop~n");
            C ->  io:format("~p~n", [C])
      end, b().
f(X) ->  Bp = spawn(prob, b, []), spawn(prob, a, [Bp, X]).
```

Здесь аргумент функции ***f*** представлен списком и процесс ***a*** посылает в качестве сообщений элементы этого списка. Когда список исчерпывается, посылается атом ***fin***. После блока receive-end необходимо выполнить рекурсивный вызов функции ***b*** для организации цикла, иначе будет выведен один только первый элемент исходного списка.

prob:f([one, two, three]). *→* ***one two three Stop***

Значит, когда приходит серия сообщений, блок receive-end обрабатывает их в цикле и этот цикл мы должны организовать сами. Функция ***b*** рекурсивно вызывает сама себя без аргументов, то-есть, создаётся бесконечный цикл. Интересно, как и в каком порядке сохраняются поступившие сообщения? В общем случае процесс может получать серию сообщений от какого-то другого процесса или от разных процессов. Все поступающие сообщения ставятся в

очередь (внешне она никак не видна), и очередное сообщение ставится в конец очереди. Принимающий процесс обрабатывает первое сообщение и удаляет его из очереди, затем обрабатываются следующие сообщение, и так далее до конца очереди.

Посмотрим теперь на пример, в котором два процесса посылают серию сообщений друг другу:

```
-module(prob).
-export([a/2,b/0,f/1]).
a(0, Bp) -> Bp ! fin, io:format("A stop~n");
a(N, Bp) -> Bp ! self(),
        receive mes -> io:format("A rec B~n") end, a(N - 1, Bp).
b() -> receive fin -> io:format("B stop~n");
                  Ap -> io:format("B rec A~n"), Ap ! mes, b()
        end.
f(N) -> Bp = spawn(prob, b, []), spawn(prob, a, [N, Bp]).
```

Новый элемент здесь встроенная функция ***self***. Эта функция возвращает ***pid*** того процесса, в теле которого она находится (в примере она возвращает ***pid*** процесса ***a***). Мы могли бы задать функцию **f** в таком виде:

```
f(N) -> Bp = spawn(prob, b, []), Ap = spawn(prob, a, [N, Bp]).
```

Но передать pid ***Ap*** процессу ***b*** здесь мы не можем, поскольку процесс ***a*** создаётся после процесса ***b*** (выше я уже обращал внимание на то, что порядок создания процессов имеет значение). Значит, процесс ***a*** в цикле посылает процессу ***b*** свой ***pid***, а процесс ***b*** посылает процессу ***a*** в цикле атом ***mes***. (На самом деле блок receive-end процесса ***a*** выполняет только одно сопоставление по атому ***mes***). Поскольку сам этот атом ***mes*** не используется, то при сопоставлении его можно было бы заменить на placeholder:

```
receive _ -> io:format("A rec B~n") end, a(N - 1, Bp).
```

Здесь нас интересует только сам факт получения сообщения, а не его содержание. После повторения сообщений заданное число раз, процесс ***a*** посылает атом ***fin*** и циклы останавливаются.

```
Prob:f(3). →
        B rec A
        A rec B
        B rec A
        A rec B
```

B rec A
A rec B
A stop
B stop

Регистрация имени процесса

В предыдущих примерах мы передавали pid процесса, как аргумент функции, или использовали функцию ***self***. Erlang также имеет встроенную функцию ***register***, позволяющую зарегистрировать имя pid процесса (или можно говорить имя самого процесса). Это имя процесса затем можно использовать всюду (глобально) в качестве pid, не заботясь о его передаче. Функция ***register*** принимает два аргумента: имя процесса, обязательно атом, и pid процесса. Изменим предыдущую программу:

```
-module(prob).
-export([a/1,b/0,f/1]).
a(0) -> bp ! fin, io:format("A stop~n");
a(N) -> bp ! ap,
        receive _ -> io:format("A rec B~n") end, a(N - 1).
b() -> receive fin -> io:format("B stop~n");
                  Ap -> io:format("B rec A~n"), Ap ! mes, b()
        end.
f(N) -> register(bp, spawn(prob, b, [])), register(ap, spawn(prob, a, [N])).
```

Здесь функция ***f*** при создании процессов одновременно регистрирует их имена: ***bp*** и ***ap***. Поскольку теперь не надо передавать функции ***a*** pid процесса ***b***, то у ***a*** остаётся только один аргумент. Не требуется теперь и применение функции ***self*** для получения pid процесса ***a***. Напоминаю, что функция ***spawn*** возвращает pid создаваемого процесса. Наверное, код будет более понятным, если использовать локальные переменные ***Ap*** и ***Bp***:

```
f(N) -> Bp = spawn(prob, b, []), register(bp, Bp),
    Ap = spawn(prob, a, [N]), register(ap, Ap).
```

Кстати, при использовании функции ***register*** уже не имеет значения порядок создания процессов функцией ***f***, их можно поменять местами:

```
f(N) -> Ap = spawn(prob, a, [N]), register(ap, Ap),
```

Bp = spawn(prob, b, []), register(bp, Bp).

Экономия времени

Параллельные вычисления позволяют ускорять вычисления. Давайте поэкспериментируем. Возьмём самый простой пример. Предположим нам надо вычислить математическое выражение:
y = sin(x) + cos(x)

Вычисление тригонометрических функций занимает приблизительно одинаковое время и оно неизмеримо меньше времени выполнения операции сложения. Значит, время вычисления по формуле будет равно времени на вычисление ***sin*** плюс время на вычисление ***cos***. Для экономии времени распараллелим вычисления:

```
-module(prob).
-export([a/1,b/1,c/0,f/1]).
a(X) -> cp ! math:sin(X).
b(X) -> cp ! math:cos(X).
c() -> Y = receive Sn -> Sn end + receive Cs -> Cs end,
            io:format("~20w~n", [Y]).
f(X) -> register(cp, spawn(prob, c, [])), spawn(prob, a, [X]),
        spawn(prob, b, [X]).
```

Здесь процессы ***a*** и ***b*** вычисляют тригонометрические функции и посылают результаты процессу ***c*** в виде сообщений. Процесс с обрабатывает сообщения, суммирует числа и выдаёт результат:
prob:f(0.5). → 1.3570081004945758

Можно запрограммировать подсчёт времени, затрачиваемого на вычисления, Erlang имеет соответствующие средства. Чтобы результаты были более наглядными, добавим ещё к вычислению тригонометрических функций задержку времени с помощью функции ***sleep*** из модуля ***timer***:

```
-module(prob).
-export([a/1,b/1,c/0,f/1]).
a(X) -> timer:sleep(100), cp ! math:sin(X).
b(X) -> timer:sleep(100), cp ! math:cos(X).
c() -> T1 = erlang:monotonic_time(), Y = receive Sn -> Sn end +
       receive Cs -> Cs end, io:format("~20w~n", [Y]),
       T2 = erlang:monotonic_time(),
```

```
    io:format("~20w~n", [(T2-T1)/1000]).
f(X) -> register(cp, spawn(prob2, c, [])), spawn(prob, a, [X]),
        spawn(prob, b, [X]).
```

Функция ***erlang:monotonic_time*** вычисляет системное время. С помощью этой функции мы определяем системное время перед и после выполнения процесса ***c***. Затем находим их разность (делим на 1000 для перевода микросекунд в миллисекунды) и выводим результат:

```
prob:f(0.5). →
        1.3570081004945758
        111.616
```

Как видим, время вычисления оказалось чуть больше заданной нами задержки в ***100*** миллисекунд, а если бы распараллеливания не было, время должно бы составить больше ***200*** миллисекунд. Можно попробовать убрать одну задержку, например, у синуса и убедиться, что общее время всё-равно будет чуть больше 100 миллисекунд.

Ожидание сообщения

Иногда для передачи сообщения процессу требуется некоторое время: ввод сообщения с клавиатуры, получение его из интернета и так далее. При этом желательно, чтобы при отсутствии сообщения достаточно долго, процесс завершился, а программа как-то отреагировала на такую ситуацию. Для подобных случаев Erlang имеет встроенную функцию ***after***. Посмотрим на её применение в следующем примере:

```
-module(prob).
-export([a/0,f/0]).
a()  -> timer:sleep(100), X = io:read(""), timer:sleep(100), bp ! X, a().
f() -> spawn(prob, a, []), register(bp, spawn(prob1, b, [])).

-module(prob1).
-export([b/0]).
b() ->  receive Mes -> io:format("message: ~p~n", [ Mes]), b()
                after 15000 -> io:format("Stop~n") end.
```

Для большей наглядности передающий (***a***) и принимающий (***b***) процессы мы поместили в разные модули. Как видим, процесс ***b*** из

модуля ***prob1*** можно запустить из другого модуля (***prob***). Функция ***io:read*** позволяет ввести текст с клавиатуры, которым мы инициируем локальную переменную ***X***. Вместо (***""***) можно было бы указать какой-нибудь текст, который был бы выведен в командную строку. Далее переменная ***X*** посылается в качестве сообщения процессу ***b*** из модуля prob1. Процесс ***b*** ожидает сообщения ***15000*** миллисекунд и если сообщение не поступило за это время, функция ***after***, выводит слово ***Stop***. Если сообщение не запоздало, оно выводится на терминал. Поскольку функция ***a*** работает в рекурсивном цикле, можно ввести несколько сообщений друг за другом. Задержки времени (***timer:sleep(100)***) в процессе ***a*** введены для того, чтобы обеспечить нужную очерёдность вывода на экран. Запустим программу:

prob:f(). → ***true***
8>"Hello World" → ***message: {ok,"Hello World"}***
8> hello. → ***message: {ok,hello}***
Stop

Функция ***after*** выводит слово ***true***, сообщая тем самым, что ожидание сообщения началось и в командной строке выводится приглашение (***8>***). Печатаем нужное сообщение на клавиатуре и оно тут же возвращается на экран. (Функция ***io:read*** возвращает текст в форме кортежа, добавляя атом ***ok***). В примере ввели два сообщения. Если текст не вводить с клавиатуры, через 15000 миллисекунд получаем слово ***Stop:***

prob:f(). → ***true***
Stop

Для повторного вызова функции ***f*** требуется вернуть Repl в исходное состояние (***ctrl/G***). Напоминаю, что вводимое с клавиатуры сообщение надо закончить точкой.

К сожалению, применение встроенных функций по вводу с клавиатуры, по выводу на экран, по обработке времени и тому подобное, нарушает принцип чистоты функций, поскольку используются побочные эффекты. Иногда побочные эффекты приводят к непредсказуемому поведению программы. Однако, без связи с внешним миром программы не имеют смысла. Проблему можно решить привлечением средств других платформ и Erlang имеет такие возможности, правда всё это осложняет жизнь.

3.Разное

Компиляция

Компилятор можно использовать несколькими способами. Ранее мы уже много раз компилировали модули из Repl командой
c(prob).
или
compile:file(prob).
где ***prob*** – имя компилируемого модуля. При этом модуль должен быть в текущей директории. Repl позволяет выполнять команду ***cd(path).*** для смены текущей директории, где ***path*** – нужный путь полный или относительный, записывается в кавычках.

Можно команду ***compile:file(prob).*** использовать для компиляции данного модуля из другого модуля, например, так:
-module(prob).
-export([f/0]).
f() -> compile:file(prob1).

Теперь вызов функции ***f***:
prob:f().
выполнит компиляцию модуля ***prob1***.

Компилировать можно также и из командной строки с помощью команды:
erlc flag file.erl

Здесь ***flag*** – параметр, называемый флагом, который позволяет дополнительно управлять процессом компиляции. Существует целый набор таких флагов, перечислим наиболее востребованные:
-debug_info – добавляет в модуль отладочную информацию, необходимую для работы таких инструментов Erlang, как отладчик, утилита статического анализа и покрытия кода.
-{out_dir, Dir} – Здесь ***Dir*** задаёт директорию в которой будет сохранён скомпилированный файл (с расширение .beam), без флага файл сохраняется в текущей директории.
-export_all – флаг заставляет игнорировать атрибут модуля ***-export***. При этом будут экспортируемыми все функции модуля, которые в нём определены. Ясно, что этот флаг может быть полезен при разработке и тестировании, не следует применять его для рабочего режима.

{d,Macro} или ***{d,Macro, Value}*** Этот флаг определяет макрос, который можно будет использовать в модуле. Более подробно об этом можно прочитать в документации.

Пример использовании флагов при компиляции из Repl может выглядеть, например, так:

c(prob, [debug_info, export_all]).

Флаги можно передать компилятору и из компилируемого модуля. Для этого надо вставить в модуль, например, такой атрибут:

-compile([debug_info, export_all]).

Теперь команда ***c(prob).*** выполнит то же самое, что и в предыдущем примере.

Обмен информацией с файлами

Посмотрим на пример функции ***f*** осуществляющей запись в файл:

```
-module(prob).
-export([f/2]).
f(FName, S) -> E = file:native_name_encoding(),
      {ok, Fd} = file:open(FName, [write]),
      file:write(Fd, unicode:characters_to_binary(S,E,E)),
      file:close(Fd).
```

Функция ***f*** имеет два аргумента: первый ***Fname*** содержит имя записываемого файла, второй ***S*** – текст, записываемый в файл. Переменная ***E*** определяет кодировку. Функция ***native_name_encoding*** из модуля **file** задаёт ту же самую кодировку, в которой записана сама программа. Вместо этого можно, конечно, кодировку указать конкретно, например:

E = utf8

Строка

{ok, Fd} = file:open(FName, [write])

открывает файл с помощью функции ***file:open***. Атом ***write*** (в квадратных скобках) указывает, что файл открыт для записи. Переменной ***Fd*** — присваивается имя файла. Левая часть этого равенства представлена кортежем, и такое присваивание допустимо, если справа тоже кортеж, у которого первый элемент тоже ***ok***, а

второй — имя файла. То-есть, функция ***open*** не только открывает файл, но ещё и возвращает кортеж вида:
{ok, Fname}.

Атом ***ok*** сигнализирует, что файл найден и благополучно открыт.

Функция ***write*** из модуля ***file*** выполняет запись в файл, а функция ***characters_to_binary*** из модуля ***unicode*** принимает записываемый контент и кодировку для него. Наконец, функция ***close*** из модуля ***file*** закрывает файл.

Если файл с заданным именем не существовал до записи, он будет создан, а если файл существовал и имел какое-то содержимое, перед записью он будет очищен.

prob:f(myfil, "Hello, World!").

После этой команды можно прочитать содержимое файла ***myfil***, расположенного в текущей директории. Разумеется, есть возможность работать и с другими директориями.

Чтение из файла выполняется проще:

```
-module(prob).
-export([f/1]).
f(FName) → {ok, S} = file:read_file(FName),
           {ok, binary_to_list(S)}.
```

Здесь наша функция ***f*** имеет единственный аргумент — имя файла. Функция ***read_file*** из модуля ***file*** возвращает кортеж, у которого первый элемент - атом ***ok*** (сообщает, что всё в порядке), а второй — прочитанный контент, который трансформируется к списку (или к string):

prob:f(myfil). → {ok,"Hello, World!"}

Следовательно, при чтении из файла можно его не открывать и не закрывать, всё происходит автоматически.

В модуле ***file*** (расположен в папке ***kernel/src***) имеется много разных функций для работы с файлами.

Заголовочные файлы (header files)

Erlang позволяет применять файлы, имя которых имеет расширение ***.hrl***. В документации их называют header files (видимо, по аналогии с языком C), хотя по смыслу такое название ничему не соответствует. В этих файлах может содержаться программный код,

предназначенный для повторного применения, или этот код предполагается изменять, может быть неоднократно. Содержимое hrl-файла может быть включено (просто механически вставлено) в головную программу с помощью атрибута ***-include***. Покажем на самом простом примере. Пусть в файле ***prob.erl*** имеется такой модуль (головная программа):

```
-module(prob).
-export([f/1]).
f(X) -> io:format("res = ~w~n", [g(X)]).
-include("prob1.hrl").
```

Здесь использована функция ***g(X)***, которая в модуле ***prob*** не определена. Определим её в файле ***prob1.hrl***:

```
g(X) -> 1/X.
```

Здесь не требуется атрибуты -module и -export.

Файл ***prob.erl*** можно скомпилировать, как обычно:

```
c(prob).
```

При этом файл ***prob1.hrl*** не требует предварительной компиляции. Теперь можно вызвать функцию ***f***:

```
prob:f(3). → res = 0.3333333333333333
```

Значит, всё работает так, как если бы программный код из файла prob1.hrl изначально располагался в файле prob.erl. Обращаю внимание на то, что имя файла для атрибута -include указывается в двойных кавычках с обязательным указанием расширения .hrl, а сам этот атрибут может располагаться в любом месте программы. Понятно, что файл prob1.hrl должен быть в текущей директории. Но в общем случае здесь может быть указано и полное (квалифицированное) имя файла и тогда его можно располагать где угодно.

Рассмотрим ещё один пример, когда в заголовочном файле расположена анонимная функция, которую мы собираемся передавать в качестве аргумента итератору ***map***:

```
-module(prob).
-export([f/1]).
f(S) -> F = g(), f(F,S).
f(F,S) -> lists:map(F, S).
-include("prob1.hrl").
```

В файле ***prob1.hrl*** находится анонимная функция (closure):

g() -> fun(X) -> X*X end.

Вызываем функцию ***f*** после компиляции:

prob:f([7,2,5,9]). → [49,4,25,81]

Надо отметить два обстоятельства. Во-первых для передачи closure, как аргумента мы ввели локальную переменную ***F***. А во-вторых, использована конструкция, когда функция ***f*** с одним аргументом (арность равна единице) выражена через функцию с тем же именем ***f*** с двумя аргументами (арность равна двойке):

f(S) -> F = g(), f(F,S).

На самом деле это разные функции, при этом функция ***f(F,S)*** локальная (не видна за пределами модуля prob) и можно было бы применить для неё другое имя, но просто есть обычай в подобных ситуациях применять одинаковые имена. Компилятор различает функции по их арности.

Применяется ещё и такой синтаксис:

```
-module(prob).
-export([f/1]).
f(S) -> lists:map(fun g/1, S).
-include("prob1.hrl").
```

В файле ***prob1.hrl*** определяем функцию ***g*** так:

g(X) -> X*X.

То-есть, в этом варианте использована обычная (не анонимная) функция ***g***. При передаче такой функции, как параметра, надо добавить слово ***fun*** перед именем функции с указанием арности.

Кроме атрибута ***-include*** есть ещё атрибут ***-include_lib***, позволяющий использовать заголовочные файлы из стандартной библиотеки.

Атрибут ***-import*** позволяет импортировать функцию для того, чтобы при её использовании не надо было указывать имя модуля, например:

```
-module(prob).
-export([f/1]).
-import(math, [sin/1]).
f(X) -> sin(X).
```

Значит, при импорте надо указать имя модуля и имя функции с указанием арности.

prob:f(1.5). → 0.9974949866040544

Синтаксический сахар

Как и другие языки, Erlang имеет много «синтаксического сахара». Например, следующая синтаксическая конструкция делает то же самое, что и итератор ***map***:

[X*X || X <- [5,2,7,9]]. → [25,4,49,81]

Применим этот «сахар» в модуле:

```
-module(prob).
-export([f/1]).
g(X) -> X*X.
f(S) -> [g(X) || X <- S].
```

Локальная функция ***g*** здесь использована для работы с элементами списка. Вызываем функцию ***f***:

prob:f([5,2,7,9]). → [25,4,49,81]

Добавим какое-нибудь условие:

```
-module(prob).
-export([f/1]).
g(X) when X rem 2 == 0 -> X/2;
g(X) -> X*X.
f(S) -> [g(X) || X <- S].
```

Теперь чётные числа делим на два, а нечётные возводим в квадрат:

prob:f([5,2,7,8,16]). → [25,1.0,49,4.0,8.0]

Подобная конструкция работает не только, как итератор:

```
-module(prob).
-export([f/1]).
f(X) -> [X || is_tuple(X)].
```

Теперь функция ***f*** возвращает свой аргумент в форме списка с одним элементом, если этот элемент представлен кортежем. В противном случае возвращается пустой список:

prob:f({1,2}). → [{1,2}]
prob:f([1,2]). → []

Можно задавать произвольные условия:

```
-module(prob).
-export([f/1]).
f(X) -> [math:log(X) || X>0].
```

Логарифм вычисляется только для положительных чисел:

prob:f(2). → [0.6931471805599453]
prob:f(-2). → []

Условий может быть несколько:

[X || X <- [1,2,a,3,4,b,5,6], integer(X), X > 3].

Здесь отбираются только целые числа и только те из них, которые больше ***3***. Результат будет равен: ***[4,5,6]*** .

Пусть элементы списка представлены кортежами. Выберем вторые члены только для тех кортежей, у которых на первом месте стоит заданный атом (в примере атом ***b***):

```
-module(prob).
-export([f/2]).
f(X, L) ->  [Y || {X1, Y} <- L, X == X1].
prob:f(b,[{a,1},{b,2},{c,3},{b,7}]). → [2,7]
```

На этой базе можно создать функцию быстрой сортировки списка:

```
-module(prob).
-export([f/1]).
f([A|T]) → f([ X || X <- T, X < A]) ++ [A] ++
           f([ X || X <- T, X >= A]);
f([]) -> [].
```

Голова ***A*** выделена из списка для использования в качестве опорной точки при начале сортировки. Не так просто проанализировать алгоритм работы этой функции сортировки.

prob:f([3,8,4,1,9,2]). → [1,2,3,4,8,9]

В модуле ***lists***, конечно, имеется готовая функция сортировки списков:

lists:sort([3,8,4,1,9,2]). → [1,2,3,4,8,9]

Как видим, подобная техника позволяет решать самые разнообразные задачи. В документации эта конструкция называется ***List Comprehensions***, обычно переводят, как «списочные выражения». Встречается этот List Comprehensions и в некоторых других современных языках программирования.

Ленивые вычисления

Erlang позволяет применять ленивый алгоритм, когда результат вычисляется только тогда, когда он потребовался для использования. В частности, есть возможность оперировать бесконечным списком,

который, естественно, не создаётся сразу, а только представляется возможность получения любой, как угодно длинной его части. Правда, реализуется такая возможность в Erlang весьма своеобразно. Посмотрим на такой пример:

-module(prob).
-export([f/1]).
f(X) -> fun() -> [X|f(X+1)] end.

Здесь функция ***f*** в качестве результата возвращает анонимную функцию. Эта функция реализует бесконечный цикл, в котором создаётся последовательность натуральных чисел, начиная с числа, представленного аргументом функции ***f***. Попробуем вызвать функцию ***f***, передав ей, например, тройку и инициируем переменную ***F*** анонимной функцией::

F=prob:f(3).

Теперь, если вызвать ***F***, как функцию, то получим несколько странный результат. С одной стороны полученный объект можно рассматривать, как список и, в частности, извлечь его голову, например, с помощью функции ***hd***:

hd(F()). → 3

Действительно, это первый элемент списка. Но если извлечь хвост, с помощью функции ***tl***, то получим не список (он был бы бесконечным), а новый объект подобный ***F***. Обозначим его ***F1***:

F1=tl(F()).

Снова можно извлечь голову и получим следующий элемент списка:

hd(F1()). → 4

Можно продолжать:

F2=tl(F1()).
F3=tl(F2()).
S=[hd(F()),hd(F1()),hd(F2()),hd(F3())].
S. → [3,4,5,6]

Для получения списка заданной длины из натуральных чисел придётся составить программу, например, такую:

-module(prob).
-export([f/1,g/2]).
f(X) -> fun() -> [X|f(X+1)] end.
g(N,N1)-> F=f(N1),h(N,F).

h(0,_)-> [];
h(N,F) -> [hd(F())]++h(N-1,tl(F())).

Здесь первый аргумент функции ***g*** задаёт количество элементов в списке, а второй — первое число списка:

prob:g(8,6). → [6,7,8,9,10,11,12,13]

Можно получать какие-то другие последовательности чисел, например, квадраты натуральных чисел:

-module(prob).
-export([f/1,g/2]).
f(X) -> fun() -> [X*X|f(X+1)] end.
g(N,N1)-> F=f(N1),h(N,F).
h(0,_)-> [];
h(N,F) -> [hd(F())]++h(N-1,tl(F())).
prob:g(8,6). → [36,49,64,81,100,121,144,169]

Проверим, работает ли эта техника ленивых вычислений, если список не бесконечный. Возьмём такой пример:

-module(prob).
-export([f/1]).
f([]) -> [];
f([X|Y]) -> fun() -> [X|f(Y)] end.

Здесь функция ***f*** принимает список и возвращает анонимную функцию, которая возвращает этот список, как результат. Проделаем с функцией ***f*** те же манипуляции, что и раньше:

F=prob:f([one,two,three,four,five]).
hd(F()). → one
F1=tl(F()).
hd(F1()). → two

Можно продолжать, всё работает точно также, только после получения атома ***five*** следующая попытка выделить хвост списка выдаст пустой список вместо ожидаемой анонимной функции. Значит, для реализации ленивых вычислений в общем случае, достаточно создать функцию, возвращающую в качестве результата анонимную функцию (closure).

Записи (records)

Записи в Erlang похожи на кортежи. Но если элемент кортежа можно извлечь по его порядковому номеру с помощью функции ***element***, то в записи используются именованные элементы. В Repl запись можно создать с помощью функции ***rd***. При этом применяется такой синтаксис:

rd(person, {name = "", age = 0, address}).

Здесь объявлена запись с именем ***person***, а после запятой, в фигурных скобках перечисляются поля записи: ***name, age*** и ***address***. Одновременно с объявлением поля можно инициировать или оставить неопределёнными. Далее значение записи можно присваивать переменным, а значения полей можно изменять следующим образом:

P = #person{name="Anna"}.

Переменная ***P*** инициирована записью ***person***, при этом изменено значение поля ***name***. Перед именем записи ставится знак ***#***.

P. → #person{name = "Anna",age = 0,address = undefined}

Не инициированное поле принимает значение ***undefined***. Для извлечения значения поля записи применяется такой синтаксис:

P#person.name. → "Anna"

P#person.address. → undefined

Таким образом, недостаточно указывать только идентификатор переменной, необходимо всегда ещё добавлять и имя записи.

В модуле запись создаётся с помощью атрибута ***-record***:

```
-module(prob).
-export([f/0]).
-record(person, {name = "",  age = 0, address=""}).
f() -> P = #person{name= "Robert", age=26, address = "Paris"},
        io:format("~p, ~p, ~p~n"[P#person.name,
        #person.age, P#person.address]).
```

Функция ***f*** создаёт запись, а затем извлекает и выводит на терминал значения полей. Вызвав функцию ***f***, получаем такой результат:

prob:f(). → "Robert", 26, "Paris"

Чтобы инициировать переменную значением нужного поля можно использовать такой странный синтаксис:

#person{name = X} = P.

Теперь переменная ***X*** получила значение поля name:

X. → "Anna"

Записи могут быть вложенными, то-есть, значениями полей могут быть записи, например:

```
-module(prob).
-export([f/2]).
-record(n, {first, last}).
-record(person, {name=#n{}, age, address}).
f(X,Y) -> P=#person{age=25, address="London",name=#n{first=X,
          last=Y}}, P1=P#person.name, [P1#n.first, P1#n.last].
```

Здесь поле ***name*** имеет значение, представленное записью ***n***. Функция ***f*** инициирует все поля, затем извлекает значение поля ***name***. При этом локальная переменная **P1** инициирована записью. Результат возвращается в форме списка:

prob:f("Robert","Ericsson"). → ["Robert","Ericsson"]

Работа с двоичными (битовыми) данными

При записи данных в файл или при отправке ещё куда-нибудь бывает выгодно преобразовать их в двоичный формат. Это позволяет предельно упростить структуру данных и существенно облегчить работу с ними. Erlang имеет богатый набор инструментов для распаковки данных в двоичный код и для восстановления их в исходном формате. Имеется несколько различных структур для представления информации в двоичном формате.

Начнём со структуры, называемой ***binary***. Синтаксически binary представляет последовательность чисел или строк заключённых в двойные угловые скобки:

```
<<5,10,20>>. → <<5,10,20>>
<<"hello">>. → <<"hello">>
<<"hello","world">>. → <<"helloworld">>
```

Целые числа в ***binary*** должны принадлежать диапазону ***0...255***, то-есть это должны быть байты в числовом представлении. Для чисел, выходящих за этот диапазон, лишние двоичные разряды отсекаются и мы получим неожиданный результат. Знаки строковых величин автоматически преобразуются в числа в соответствии с кодировкой ASCII. Таким образом, выражение <<***"cat"***>> можно рассматривать просто, как краткую форму для <<***99, 97, 116***>>. Если содержимое

binary представлено печатаемыми знаками, то при выводе печатается строка, а в противном случае выводятся числа:

<<99, 97, 116>>. → <<"cat">>

<<99, 97, 116, 21>>. → <<99,97,116,21>>

Значит, если хотя бы один элемент не представляет печатаемый знак, все элементы будут выведены в форме чисел.

Имеется много встроенных функций для работы с binary. Функция ***list_to_binary*** преобразует произвольный список в binary:

B1=[5,6,7,<<"abc">>].

B=[<<2,13>>,"hello",B1].

list_to_binary(B). → <<2,13,104,101,108,108,111,5,6,7,97,98,99>>

Здесь список ***B*** содержит вложенный список ***B1***, а сами эти списки имеют вложенные binary. Кстати, списки с элементами в формате binary в документации называются ***iolist***. Функция ***list_to_binary*** всё это распаковывает в один binary.

Функция ***split_binary*** принимает два аргумента: binary и номер позиции. Функция разбивает binary на две части по заданной позиции и возвращает результат в формате кортежа.

B= <<1,2,3,4,5,6,7>>.

split_binary(B,3). → {<<1,2,3>>,<<4,5,6,7>>}

Функция ***term_to_binary*** преобразует любой объект Erlang (***term***) в binary, а функция ***binary_to_term*** выполняет обратное преобразование. Для примера применим эти функции к хешу:

H=#{a=>1,b=>2,c=>3}.

B=term_to_binary(H). →

<<131,116,0,0,0,3,100,0,1,97,97,1,100,0,1,98,97,2,100,0,1,99,97,3>>

binary_to_term(B). → #{a => 1,b => 2,c => 3}

Эти две функции особенно полезны, так как они позволяют любые комплексные (сложные) данные передавать в файл, в процесс, в базу данных и так далее в форме двоичного (битового) текста, а затем при необходимости восстанавливать информацию в исходном виде.

Функция ***byte_size*** возвращает размер binary в байтах:

byte_size(B). → 24

а функция ***bit_size*** — в битах:

bit_size(B). → 192

Кроме встроенных функций Erlang позволяет применять специальный синтаксис для работы с битовым кодом. В основном

этот синтаксис используется для извлечения и упаковки отдельных битов или последовательностей битов в двоичном коде. Рассмотрим примеры. Пусть у нас есть три переменных: ***X=2, Y=17 и Z=8***, которые мы хотим упаковать в 16-битную область памяти так, чтобы ***X*** занимала ***3*** бита, ***Y – 7*** битов, и ***Z – 6*** битов. Это будет выглядеть так:

[X,Y,Z]=[2,17,8].
M= <<X:3, Y:7, Z:6>>. → <<"DH">>

Этот код создаёт binary и запоминает его в переменной ***M***. Длина этого binary равна ***16*** битам, то-есть, мы имеем два байта, которые оказались «печатаемые» и потому мы получили два символа ***"DH"***. Но что получится, если длина не будет соответствовать целому числу байтов? Попробуем уменьшить область для ***X*** так, чтобы она составляла только два бита:

M1= <<X:2, Y:7, Z:6>>. → <<136,72:7>>

В результате имеем один полный байт (***136***) и число ***72***, для которого выделено только ***7*** битов. Такую структуру данных в Erlang называют ***bitstring***. Познакомимся на примерах с некоторыми свойствами ***bitstring***. Сначала создадим binary:

B1 = <<1:8>>. → <<1>>

B1 содержит в точности один байт:

byte_size(B1). → 1

и 8 битов:

bit_size(B1). → 8

Проверим, какой структуре принадлежит ***B1***:

is_binary(B1). → true
is_bitstring(B1). → true

Значит, ***B1*** одновременно считается и ***binary*** и ***bitstring***. Теперь создадим bitstring:

B2 = <<1:17>>. → <<0,0,1:1>>
byte_size(B2). → 3

В байтах его размер оказался равным ***3*** — два полных байта, равных нулю, и один неполный.

bit_size(B2). → 17
is_binary(B2). → false
is_bitstring(B2). → true

Значит, ***B2*** не является binary.

Работать с bitstring сложнее, чем с binary, в частности, их нельзя записывать в файл; там требуется байты.

Erlang позволяет извлекать отдельные биты из байтов. Для этого применяется ещё одна конструкция, называемая ***bit comprehension***. Создадим binary:

B = <<16#5f>>. → <<"_">>

Выражение ***16#5f*** — это число ***95*** в шестнадцатиричной системе. Перевести число в десятичную систему очень просто — достаточно написать:

X=16#5f.
X. → 95

Этот синтаксис позволяет представлять числа в любой системе счисления, например:

Y=9#45.
Y. → 41

Здесь код ***9#45*** представляет число ***41*** в девятиричной системе.

Для перевода чисел в двоичную систему счисления применяется такой, довольно замысловатый синтаксис:

C = [X || <<X:1>> <= B].
C. → [0,1,0,1,1,1,1,1]

Получили двоичный код в формате списка. При этом использована локальная переменная ***X***. Перевод можно выполнять из любой системы счисления:

Y= <<9#45>>. → <<")">>
D = [X || <<X:1>> <= Y].
D. → [0,0,1,0,1,0,0,1]

Чтобы получить вместо списка binary надо поступить так:

<< <<X>> || <<X:1>> <= B >>. → <<0,1,0,1,1,1,1,1>>

На самом деле это тот же самый List Comprehensions, о котором я упоминал ранее. Обращаю внимание на то, что в данном случае применяется знак ***(<=)*** вместо ***(<-).***

Рассмотрим теперь более содержательный пример. Для создания цвета применяются так называемые RGB-триплеты, определяющие интенсивность красного, зелёного и синего цветов. С помощью binary можно, например, создать 16-битную область памяти, содержащую RGB-триплет, например такой:

[Red, Green, Blue] = [2, 61, 20].

Выделим ***5*** бит для красного цвета, ***6*** бит — для зелёного и ***5*** бит для синего и создадим binary с идентификатором ***M***:

M= <<Red:5, Green:6, Blue:5>>. → <<23,180>>

Получили binary размером в два байта. Так же просто можно извлечь значения из триплета ***M***:

<<R1:5, G1:6, B1:5>> = M.
R1. → 2
G1. → 61
B1. → 20

Способ ссылки на функцию в модуле

На локальную функцию (да и не на локальную тоже) в модуле можно ссылаться так, как показано на следующем примере:

```
-module(m1).
-export([g/1]).
f(X) -> X*X.
g(L) -> lists:map(fun f/1, L).
```

Здесь используется служебное слово ***fun***, а для вызываемой функции указывается её арность.

m1:g([2,3,4,5]). → [4,9,16,25]

Если вызываемая функция находится в другом модуле, то вызывается она так же, только указывается модуль:

```
-module(m2).
-export([g/1]).
g(L) -> lists:map(fun m1:f/1,L).
```

Ясно, что теперь функция ***f*** должна быть экспортируемой.

m2:g([2,3,4,5]). → [4,9,16,25]

Функцию можно вызвать также с помощью директивы ***apply***. В общем виде это можно представить так:

apply(Mod, Func, [Arg1, Arg2, ..., ArgN]).

Здесь ***Mod*** - имя модуля, ***Func*** — имя функции, ***[Arg1, Arg2, ..., ArgN]*** — список аргументов функции. Например:

```
-module(prob).
-export([g/2,f/2]).
f(X,Y) -> X*Y.
g(X,Y) -> apply(prob,f,[X,Y]).
```

Если вызывается функция из того же модуля, как в этом случае, она должна быть экспортируемой.

prob:g(2,3). → 6

Фактически вызов с помощью ***apply*** эквивалентен обычному:

prob:f(X,Y).

Тем не менее, применение ***apply*** иногда бывает полезно и даже необходимо. В частности, при таком вызове возможно динамическое вычисление имён модуля и функции. Например, можно программировать так:

```
-module(prob).
-export([g/3,f1/2,f2/2]).
f1(X,Y) -> X*Y.
f2(X,Y) -> X+Y.
g(X,Y,R) -> apply(prob,R,[X,Y]).
```

Третий аргумент функции ***g*** представляет имя вызываемой функции.

prob:g(2,3,f1). → 6

prob:g(2,3,f2). → 5

Директиву ***apply*** можно применять и для вызова библиотечных функций, например:

apply(erlang, atom_to_list, [hello]). → "hello"

Здесь вызывается функция ***atom_to_list*** из модуля ***erlang***. Кстати, при обычном вызове имя модуля ***erlang*** можно опускать.

Локальная переменная в списке аргументов функции

Посмотрим на такой пример:

```
F = fun([[A,B,C]|T]) -> lists:sum([A,B,C]) end.
F([[1,2,3],4,5,6]). → 6
```

Здесь при использовании головы списка, которая сама является списком, в теле функции мы просто повторили текст. Но Erlang позволяет ввести локальную переменную в списке аргументов. При этом используется необычный синтаксис, когда идентификатор переменной находится справа от знака равенства:

```
F = fun([[A,B,C] = R|T]) -> lists:sum(R) end.
F([[1,2,3],4,5,6]). → 6
```

При этом надо отметить, что в данном случае работает и обычный синтаксис:
F = fun([R = [A,B,C]|T]) -> lists:sum(R) end.
F([[1,2,3],4,5,6]). → 6

Конечно, можно сделать и так:
F = fun([Z|T]) -> lists:sum(Z) end.
F([[1,2,3],5,6,7,8]). → 6

Разница, наверное, только в читабельности кода.

4.Клиент-Сервер (Client-Server)

Обычно словом сервер называют мощный компьютерный центр, с соответствующим программным обеспечением. Клиенты имеют возможность обращаться к серверу с запросами по сети интернета. Здесь мы несколько изменим трактовку этих слов. В Erlang архитектура клиент-сервер включает только два различных процесса, которые могут быть запущены на разных компьютерах, или даже на одном и том же компьютере. Инициатором связи всегда выступает процесс-клиент, посылающий запросы процессу-серверу. При этом сервер обеспечивает соответствующую реакцию на запрос клиента. Для того, чтобы более детально разобраться, как это работает, вернёмся снова к теме параллельного программирования на Erlang.

Spawn и *receive – end*

Как мы уже знаем, процесс запускает функция ***spawn***, обращение к которой в общем виде можно представить так:
Pid = spawn(Mod, Func, Args).

Здесь ***Pid*** – это обозначение для идентификатора процесса, возвращаемого функцией spawn в качестве результата. Функция ***Func*** должна быть экспортирована из модуля ***Mod*** и ей должен быть передан список аргументов ***Args***. Процессу, имеющему идентификатор ***Pid***, можно послать сообщение с помощью оператора ***(!)*** (в докуметации он называется ***send-onepamop***):
Pid ! Message.

Одновременно можно послать сообщение сразу нескольким процессам с идентификаторами ***Pid1, Pid2, Pid3,*** и так далее. Делается это так:

Pid1 ! Pid2 ! Pid3 ! … ! Message.

Процесс получатель должен иметь некий отрезок программного кода, ограниченный словами ***receive – end***. Для определённости будем дальше называть его ***receive – end*** блоком. В этом блоке программируется выбор реакции на полученное сообщение. Для этого используется обычное сопоставление с образцом (pattern matching) и в общем виде применяется следующий синтаксис:

r***eceive***

Pattern1 [when Guard1] –> Expression1;
Pattern2 [when Guard2] –> Expression2;
…

end.

Значит, дополнительно к образцу может быть добавлено (необязательное) условие (охрана, Guard), на которое проверяется полученное сообщение. Если сопоставление сообщения с образцом удачно и Guard возвращает ***true***, выполняется соответствующее выражение (***Expression***). Если же ни одно сопоставление не было удачно, сообщение сохраняется и может быть использовано в дальнейшем. Позднее рассмотрим, как это можно сделать.

В общем виде процесс передачи сообщения можно описать так. Когда команда ***spawn*** выполняется, система создаёт новый процесс. Каждый такой процесс имеет свой почтовый ящик, создаваемый одновременно с процессом. Отправленное сообщение помещается в почтовый ящик, а его содержимое проверяется, когда программа выполняет ***receive-end*** блок. Фактически конструкция receive-end блока не отличается от функции Erlang. Покажем это на простейшем примере функции ***area***, вычисляющей площадь прямоугольника и квадрата. Она может выглядеть, например, так:

area({rectangle, W, H}) –> W * H;
area({square, S}) –> S * S.

Перепишем теперь эту функцию, как процесс и создадим модуль:

-module(prob).
-export([f/0]).
f() -> receive

```
    {rectangle, W, H} -> io:format("Area is ~p~n", [W * H]), f();
    {square, S} -> io:format("Area is ~p~n", [S * S]), f()
  end.
```

Компилируем модуль:

c(prob).

Теперь в Repl можно создать процесс, выполняющий то же самое, что и функция ***area***:

Pid = spawn(prob1, f, []). → <0.93.0>

<0.93.0> - это значение ***Pid***.

Посылаем процессу сообщения:

Pid ! {rectangle, 6, 10}. → Area is 60 {rectangle,6,10}
Pid ! {square, 12}. → Area is 144 {square,12}

Процесс выводит результат вычислений, а send-оператор ***(!)*** возвращает переданное сообщение, которое Repl также выводит на экран.

Функцию ***f***, выполняющую блок receive-end, мы каждый раз вызываем рекурсивно, создавая бесконечный цикл. Тем самым обеспечивается возможность получения и обработки сколько угодно новых сообщений.

Простой пример клиент-сервер

В рассмотренном выше примере мы из Repl посылаем сообщение процессу, который выполняет вычисления и выводит результат с помощью функции вывода io:format. При этом обратная связь между процессом и Repl отсутствует, процесс даже не знает, кто послал ему сообщение. Для того, чтобы создать эту обратную связь необходимо сделать некоторые дополнения к нашей программе. Дальше будем называть словом клиент тот процесс, который посылает сообщение и является инициатором связи, а словом сервер процесс, реагирующий на запрос и возвращающий ответ.

Итак, перепишем текст программы из предыдущего примера в другом виде. Для конкретности назовём модуль словом ***server*** и, значит, поместим текст в файл ***server.erl***:

```
-module(server).
-export([f/0,g/2]).
g(P, R) -> P ! {self(), R},
```

```
    receive
       X -> X
    end.
f() -> receive
          {P, {rectangle, W, H}} -> P ! W * H, f();
          {P, {square, S}} -> P ! S * S, f()
          {P, X} -> P ! {error,X}, f()
        end.
```

Таким образом, здесь мы добавили ещё одну функцию ***g*** и несколько изменили функцию ***f***. Далее, после компиляции программы, с помощью функции ***f*** создаём сервер с идентификатором ***Pid***:

Pid = spawn(server, f, []).

Теперь вместо отправки сообщения непосредственно серверу, воспользуемся функцией ***g***, передав ей идентификатор сервера и аргументы для вычисления площади:

server:g(Pid,{rectangle,6,8}). → 48

Как видим, функция ***g*** сразу возвращает результат. Разберёмся детально, как это работает. Получив идентификатор сервера (переменная ***P***), функция ***g*** посылает серверу идентификатор клиента, который эту функцию вызвал. Этот идентификатор определяется функцией ***self***. Затем сервер, получив идентификатор клиента, посылает ему результат вычисления площади. При этом роль приёмника клиента выполняет блок receive-end функции ***g***, запущенной клиентом, а в роли самого клиента здесь выступает Repl. Блок receive-end просто возвращает то, что поступило ему на вход. В итоге и сама функция ***g*** возвращает результат вычисления площади геометрической фигуры.

В список вариантов сопоставления с образцом мы добавили ещё строку

{P, X} -> P ! {error,X}, f()

В этой строке сопоставление будет удачно при любой информации, поступившей на вход. Такую строку следует добавлять всегда, чтобы исчерпать все альтернативные варианты, включая любые ошибочные данные. Получив такие данные, сервер передаст клиенту слово ***error*** и вернёт то, что клиент передал серверу:

prob:g(Pid,{aaa}). → {error,{aaa}}

Хотя созданная нами конструкция клиент-сервер работает, однако имеется один недостаток. Отправив свой запрос, клиент ожидает ответа (блок receive-end функции ***g***), но ждёт он ответ не конкретно от того, к кому обратился, а от кого угодно. Если в период этого ожидания клиенту будет передано сообщение от какого-то другого процесса, то оно может быть ошибочно принято за ответ сервера. Чтобы избежать такой ошибки, ещё раз несколько изменим функции ***g*** и ***f***:

```
-module(server).
-export([f/0,g/2]).
g(P, R) -> P ! {self(), R},
       receive
         {P,X} -> X
       end.
f() -> receive
           {P, {rectangle, W, H}} -> P ! {self(),W * H}, f();
           {P, {square, S}} -> P ! {self(),S * S}, f();
           {P, Other} -> P ! {self(),{error,Other}}, f()
        end.
```

Теперь сервер вместе с ответом на запрос посылает ещё и свой идентификатор (он определяется функцией self). В результате клиент будет получать ответ только от нужного сервера, а все другие сообщения будут проигнорированы.

Наконец, можно ещё спрятать вызов функции spawn в модуле server и ввести для получения площади фигуры функцию с названием ***area***, превратив функцию ***g*** в локальную:

```
-module(server).
-export([f/0,area/2,start/0]).
start() -> spawn(server,f,[]).
area(Pid,Y) -> g(Pid,Y).
g(P, R) -> P ! {self(), R},
       receive
         {P,X} -> X
       end.
f() -> receive
           {P, {rectangle, W, H}} -> P ! {self(),W * H}, f();
           {P, {square, S}} -> P ! {self(),S * S}, f();
```

```
        {P, Other} -> P ! {self(),{error,Other}}, f()
    end.
```

Теперь работа клиента будет выглядеть так:

```
Pid=server:start().
server:area(Pid,{rectangle,10,8}). → 80
```

Ограничение числа работающих процессов

Посмотрим на ещё один пример работы с процессами на Erlang:

```
-module(prob).
-export([max/1]).
max(N) -> Max = erlang:system_info(process_limit),
    io:format("Maximum allowed processes:~p~n",[Max]),
    statistics(runtime), statistics(wall_clock),
L = for(1, N, fun() -> spawn(fun() -> wait() end) end),
    {_, Time1} = statistics(runtime),
    {_, Time2} = statistics(wall_clock),
    lists:foreach(fun(Pid) -> Pid ! die end, L),
    U1 = Time1 * 1000 / N, U2 = Time2 * 1000 / N,
    io:format("Process spawn time=~p (~p) microseconds~n", [U1,
U2]).
wait() -> receive die -> void end.
for(N, N, F) -> [F()];
for(I, N, F) -> [F()|for(I+1, N, F)].
```

Здесь использована функция ***erlang:system_info(process_limit)***, позволяющая определить количество одновременно работающих процессов. Это ограничение установлено самой системой Erlang. Применена также функция ***statistics***, позволяющая контролировать время работы программы. Все остальные действия нам уже знакомы. При запуске 20000 процессов, получим:

```
prob:max(20000).
Maximum allowed processes:262144
Process spawn time=7.0 (6.25) microseconds
```

Запуск процессов в количестве больше допустимого предела, вызывает сообщение об ошибке:

```
prob:max(300000).
Maximum allowed processes:262144
```

=ERROR REPORT==== 6-Jun-2019::19:29:12 ===
Too many processes

Число дозволенных процессов можно установить самостоятельно, запустив систему Erlang с ключом ***P***:

erl +P 3000000

Теперь система допускает запуск 3000000 процессов (на самом деле даже больше) одновременно:

prob:max(500000).
Maximum allowed processes:4194304
Process spawn time=9.47 (10.75) microseconds

Кстати, имеются и другие ключи, позволяющие управлять некоторыми параметрами системы.

Динамическая загрузка кода

Erlang позволяет при работающем процессе динамически загружать программный код. Если, например, процесс использует функцию из какого-то модуля, то можно не останавливая процесс изменить и перекомпилировать используемый модуль. При этом процесс всегда будет вызывать последнюю версию модуля. Покажем на простейшем примере. Пусть в модуле ***b*** имеется такая функция ***g***:

```
-module(b).
-export([g/0]).
g() -> 1.
```

Создадим модуль ***a*** позволяющий запускать процесс, в котором вызывается функция ***b:g()***:

```
-module(a).
-export([start/1,f/1]).
start(T) -> spawn(a,f,[T]).
f(T) -> sleep(), V = b:g(),
      io:format("f (~p) b:g() = ~p~n",[T, V]), f(T).
sleep() -> receive after 10000 -> true end.
```

Здесь процесс на базе функции ***f*** имеет бесконечный цикл, в котором вызывается функция ***b:g()*** и выводится на экран её результат, то-есть цифра ***1***. В цикле вставлена задержка, определяемая локальной функцией ***sleep***, созданной с помощью блока ***receive-end*** и

оператора ***after*** с заданным временем ожидания ***10*** секунд.
Компилируем модули ***a*** и ***b*** и запускаем процесс:
a:start(aaa).

Получаем вывод:
f (aaa) b:g() = 1
f (aaa) b:g() = 1
f (aaa) b:g() = 1

Теперь не останавливая процесс изменим модуль ***b***:
-module(b).
-export([g/0]).
g() -> 2.

В период 10-секундного ожидания перекомпилируем модуль ***b***:
c(b).

Выводимый результат изменится:
f (aaa) b:g() = 2
f (aaa) b:g() = 2

Более того, также не останавливая процесс в период ожидания можно запустить его ещё раз с другим аргументом:
a:start(bbb).

Получим два параллельных процесса:
f (aaa) b:g() = 2
f(bbb) b:g() = 2
f (aaa) b:g() = 2
f(bbb) b:g() = 2

Если изменить и перекомпилировать модуль ***b*** ещё раз:
-module(b).
-export([g/0]).
g() -> 3.

то оба процесса будут выводить новый результат:
f (bbb) b:g() = 3
f (aaa) b:g() = 3

Кстати, в модуле ***timer*** есть уже готовая функция ***sleep*** и мы её уже использовали ранее. Так что модуль ***a*** может быть таким:
-module(a).
-export([start/1,f/1]).
start(T) -> spawn(a,f,[T]).
f(T) -> timer:sleep(10000), V = b:g(),

io:format("f (~p) b:g() = ~p~n",[T, V]), f(T).

При этом всё будет работать также.

Динамическая загрузка возможна только в процессах При использовании функций этот трюк не работает. Например, создадим два таких модуля:

```
-module(m2).
-export([g/0]).
g() -> aaa.

-module(m1).
-export([f/0]).
f() -> timer:sleep(10000), io:format("~p~n", [m2:g()]), f().
```

В этом случае изменить динамически модуль ***m2*** невозможно.

5. ETS и DETS таблицы

Общие сведения

Erlang имеет два системных модуля ***ets*** и ***dets***, предназначенных для работы по сохранению и просмотру больших объёмов данных в памяти или на диске. Сохраняемая информация представляется в форме специальных таблиц, которые, соответственно, называются ***ETS*** (erlang tertm storage) и ***DETS*** (disk ETS). Сохраняемые данные могут быть представлены любыми типами (потому они и называются словом ***tertm***): числа, строки, коллекции и так далее. Модуль ***ets*** позволяет сохранять данные в оперативной памяти (при наличии свободной памяти достаточного объёма), а модуль ***dets*** - на диске. Модули имеют средства, как для записи данных (insertion), так и для извлечения и просмотра (lookup). Оба модуля имеют аналогичный интерфейс и отличаются только тем, что модуль ***ets*** более быстрый (естественно), но зато он не гарантирует сохранение данных при непредвиденных сбоях (crash) программы. Ну, а более медленный ***dets*** имеет специальные средства для защиты и восстановления данных на диске в подобных случаях.

Фактически ETS и DETS таблицы являются коллекциями, элементы которых представлены кортежами. Таблицы ETS временные

(transient), помещённые в них данные пропадают, когда таблица удаляется, или когда соответствующий процесс Erlang заканчивается. Таблицы DETS постоянны (persistent) и сохраняются в подобных ситуациях.

Разновидности таблиц ETS и DETS

Сохраняемые в таблицах кортежи имеют специальную структуру. Один из элементов кортежа (первый по умолчанию) выполняет роль ключа (***key***) кортежа. Соответственно, все остальные элементы кортежа называются значениями (***value***). Вставка и извлечение данных выполняются по ключу. Кортежи группируются в таблице по ключу и с учётом типа таблицы. Имеется четыре разновидности таблиц, которым присвоены специальные названия: ***set, ordered_set, bag, duplicate_bag***. В таблице ***set*** ключи всех кортежей должны быть различными (уникальными). Таблица ***ordered_set*** отличается от таблицы ***set*** только тем, что кортежи в ней отсортированы по ключам. В таблице ***bag*** допустимы кортежи с одинаковыми ключами, но составы остальных элементов кортежей должны отличаться. Наконец, в таблице ***duplicate_bag*** допустимы полностью совпадающие (по ключам и по значениям) кортежи. Каждый раз тип таблицы выбирается программистом исходя из конкретных обстоятельств.

Имеется четыре основных операции над ETS и DETS таблицами: ***ets:new*** – создаёт новую таблицу, а ***dets:open_file*** – открывает существующую таблицу.
insert(Id, X) – вставка одного или нескольких кортежей в таблицу. Здесь ***X*** – кортеж или список кортежей, а ***Id*** – идентификатор таблицы. Операция одинакова для таблиц ETS и DETS.
lookup(Id, Key) – просмотр кортежа в таблице. Если в таблице несколько кортежей с одинаковыми ключами, результат будет представлен списком этих кортежей. Если кортежа с указанным ключом в таблице нет, возвращается пустой список. Операция одинакова для таблиц ETS и DETS.
ets:delete(Id) или ***dets:close(Id)*** – удаление таблицы с заданным идентификатором.

Модули ets и dets содержат, конечно, и другие функции для работы с таблицами.

Посмотрим на примере, как можно работать с таблицами. Пока ограничимся таблицей ETS. Применим одновременно все четыре вида таблиц, а для их перебора используем цикл с итератором ***foreach***:

```
-module(prob).
-export([start/0]).
start() -> lists:foreach(fun f/1, [set, ordered_set, bag, duplicate_bag]).
f(Mode) -> Id = ets:new(t, [Mode]),
        ets:insert(Id, {a,1}),
        ets:insert(Id, {b,2}),
        ets:insert(Id, {a,1}),
        ets:insert(Id, {a,3}),
        List = ets:tab2list(Id),
        io:format("~-13w  => ~p~n", [Mode, List]), ets:delete(Id).
```

Функция start организует цикл, в котором функции ***f*** поочерёдно передаются названия четырёх видов таблиц (для вызова функции ***f*** использован способ с применением слова ***fun***). Локальная функция ***f*** создаёт таблицы и вставляет в них кортежи, потом извлекает эти кортежи в форме списка ***List*** с помощью встроенной функции ***ets:tab2list*** и выводит эти списки на экран. В конце все таблицы удаляются. На месте первого аргумента функции ***new*** указан атом ***t***, который нужен в специальных случаях, но в нашей программе он не используется. Идентификатор таблиц ***Id*** создаёт функция ***ets:new***. Применим простейшие кортежи, содержащие только ключ и одно значение: ***{a,1}, {b,2}, {a,1}, {a,3}.*** Три кортежа из четырёх имеют одинаковый ключ и при этом два кортежа полностью совпадают. Получим следующий результат:

```
prob:start(). →
set           => [{b,2},{a,3}]
ordered_set   => [{a,3},{b,2}]
bag           => [{b,2},{a,1},{a,3}]
duplicate_bag => [{b,2},{a,1},{a,1},{a,3}]
```

Анализ результата позволяет установить, что при вставке в таблицу ***set*** кортежа с ключом, встречавшимся ранее, предыдущий кортеж удаляется. Таким образом, в таблице ***set*** остаётся только последний из кортежей с одинаковыми ключами. Таблица ***ordered_set*** это отсортированная по ключам таблица ***set***. В таблице ***bag*** все

кортежи с одинаковыми ключами сохраняются, но если встречаются полностью одинаковые кортежи, то сохраняется только один. Наконец, в таблице ***duplicate_bag*** сохраняются все кортежи.

Практически, конечно, бывает нужен какой-то один вид таблицы. В этом случае функция ***start*** не нужна:

```
-module(prob).
-export([f/1]).
f(Mode) -> Id = ets:new(t, [Mode]),
        ets:insert(Id, {a,1}),
        ets:insert(Id, {b,2}),
        ets:insert(Id, {a,1}),
        ets:insert(Id, {a,3}),
        List = ets:tab2list(Id),
        io:format("~-13w  => ~p~n", [Mode, List]), ets:delete(Id).
```

Теперь функция ***f*** экспортируемая и в качестве аргумента принимает название вида таблицы.

```
prob:f(set). →                  set  => [{b,2},{a,3}]
prob:f(duplicate_bag). → duplicate_bag  => [{b,2},{a,1},{a,1},{a,3}]
```

Пока мы ещё не затонули многие аспекты языка Erlang. Не знаю, смогу ли я закончить эту работу...

math:sqrt(-4). → exception rror: an error occurred when evaluating an arithmetic expression in function math:sqrt/1 called as math:sqrt(-4)

```
-module(prob).
-export([sqr/1]).
sqr(X) when X<0 -> error({"Neganive Argument in sqrt(X):",X});
sqr(X) -> math:sqrt(X).
```

```
prob:sqr(-4). → exception error: {"Neganive Argument in sqrt(X):",-4}
    in function  prob:sqr/1 (prob.erl, line 3)
```