Бьярн Дэкер (Bjarne Däcker)
Роберт Вирдинг (Robert Virding)

Настольная книга по Erlang

Бьярн Дэкер (Bjarne Däcker) и Роберт Вирдинг (Robert Virding)

Версия:

Wed Sep 17 23:36:31 2014 +0200

Последняя версия этой книги находится по адресу:
http://opensource.erlang-solutions.com/erlang-handbook

Главный редактор

Омер Килич (Omer Kilic)

Прочие участники

Перевод на русский: Дмитрий Литовченко (Dmytro Lytovchenko)

Список людей, которые помогли с изменениями и исправлениями находится в репозитории проекта.

Соглашения

Спецификации синтаксиса задаются с помощью `этого моноширинного шрифта`. В квадратные скобки ([ ]) заключаются необязательные части. Термы, начинающиеся с заглавной буквы, как например *Integer* должны быть заменены на какое-нибудь подходящее значение. Термы, начинающиеся со строчной буквы, как например `end`, являются зарезервированными словами языка Erlang. Вертикальная черта (|) разделяет альтернативные варианты, как например Integer | Float.

Ошибки и улучшения

Это живой документ, пожалуйста присылайте исправления и рекомендации по улучшению содержимого используя систему учёта задач Github по адресу https://github.com/esl/erlang-handbook. Вы также можете создать свою ветвь репозитория (fork) и прислать нам запрос на соединение ветвей (pull request) с вашими предлагаемыми исправлениями и предложениями. Новые ревизии этого документа будут публиковаться по мере накопления больших изменений.

Оглавление

1 Вступление

или почему Erlang такой, как он есть

Erlang — это результат проекта по улучшению программирования приложений для телекоммуникаций в Лаборатории Компьютерных Наук (CSLab или Computer Science Lab) компании Ericsson. Критически важным требованием была поддержка характеристик этих приложений, таких, как:

- Массивная параллельность
- Устойчивость к сбоям
- Изоляция
- Динамическое обновление кода во время его исполнения
- Транзакционность

В течение всей истории Erlang процесс его разработки был исключительно прагматичным. Характеристики и свойства видов систем, в которых была заинтересована компания Ericsson, прямым образом влияли на ход разработки Erlang. Эти свойства считались настолько фундаментальными, что было решено поддержку для них встроить прямо в язык, вместо дополнительных библиотек. По причине прагматичного процесса разработки вместо предварительного планирования, Erlang «стал» функциональным языком — поскольку свойства функциональных языков очень хорошо подходили к свойствам систем, которые разрабатывались.

2 Структура Erlang-программы

2.1 Синтаксис модулей

Программа на Erlang состоит из **модулей** где каждый модуль является текстовым файлом с расширением **.erl**. Для небольших программ все модули обычно хранятся в одной директории. Модуль состоит из атрибутов модуля и определений функций.

```
-module(demo).
-export([double/1]).

double(X) -> times(X, 2).

times(X, N) -> X * N.
```

Показанный модуль `demo` состоит из функции `times/2`, которая является локальной для модуля и функции `double/1`, которая экспортирована и может быть вызвана снаружи модуля.

`demo:double(10)` ⇒ `20` (стрелка ⇒ читается как «даёт результат»)

`double/1` означает, что перед нами функция «double» с *одним* аргументом. Функция `double/2` принимает *два* аргумента и считается другой, отличной от первой, функцией. Количество аргументов называется **арностью** (arity) данной функции.

2.2 Атрибуты модулей

Атрибут модуля определяет некоторое свойство модуля и состоит из **метки** и **значения** в следующем виде:

```
-Метка(Значение).
```

`Метка` должна быть атомом, в то время, как значение `Значение` должно быть термом (смотрите главу 3). Можно указывать любое имя для атрибута модуля, также разрешены повторы. Атрибуты хранятся в скомпилированном коде и могут быть извлечены с помощью вызова функции `Модуль:module_info(attributes).`

2.2.1 Предопределённые атрибуты модулей

Предопределённые атрибуты модулей должны быть размещены в модуле до начала первого определения функции.

- `-module(Модуль).`
  Этот атрибут обязателен и должен идти первой строкой файла. Он определяет имя модуля. Имя Module, атом (смотрите раздел 3.1.1), должно соответствовать имени файла без расширения `.erl`.
- `-export([Функ1/Арность1, ..., ФункN/АрностьN]).`
  Этот атрибут указывает, какие функции модуля могут быть вызваны снаружи модуля. Каждое имя функции `ФункX` — это атом и `АрностьX` функции — целое число.
- `-import(Модуль,[Функ1/Арность1, ..., ФункN/АрностьN]).`
  Этот атрибут указывает модуль `Модуль`, из которого импортируется список функций. Например:
  - `import(demo, [double/1]).`
    Эта запись означает, что теперь можно писать `double(10)` вместо более длинной записи `demo:double(10)`, которая может быть непрактична, если функция используется часто.
- `-compile(Параметры).`
  Параметры компилятора для данного модуля.
- `-vsn(Vsn).`
  Версия модуля. Если не указать этот атрибут, то по умолчанию будет исполь-

зована контрольная сумма содержимого модуля.

- `-behaviour(Поведение).`
  Этот атрибут указывает поведение модуля, либо выбранное пользователем, либо одно из стандартных поведений OTP: `gen_server`, `gen_fsm`, `gen_event` или `supervisor`. Принимаются как британская (behaviour), так и американская запись (behavior).

2.2.2 Определения записей и макросов

Записи и макросы определяются так же, как и другие атрибуты модуля:

```
-record(Запись, Поля).
-define(Макро, Замена).
```

Записи и определения макросов также позволяются между функциями, если определение встречается раньше, чем его первое использование. (Подробнее о записях читайте секцию 3.2.2, а о макросах главу 11.)

2.2.3 Включение содержимого файлов

Включение содержимого файлов указывается аналогично другим атрибутам модуля:

```
-include(Файл).
-include_lib(Файл).
```

`File` — это строка, представляющая собой имя файла. Включаемые файлы обычно используются для определений записей и макросов, которые используются в нескольких модулях сразу. По договорённости для включаемых файлов используется расширение `.hrl`.

```
-include("my_records.hrl").
-include("incdir/my_records.hrl").
-include("/home/user/proj/my_records.hrl").
```

Если путь к файлу `File` начинается с компонента пути `$Var`, то значение переменной окружения `Var` (которое возвращается функцией `os:getenv(Var)`) будет подставлено вместо `$Var`.

```
-include("$PROJ_ROOT/my_records.hrl").
```

Директива `include_lib` подобна `include`, но считается, что первый компонент пути — имя приложения.

```
-include_lib("kernel/include/file.hrl").
```

Сервер кода (специальный процесс, управляющий загрузкой и версиями модулей в памяти) использует функцию `code:lib_dir(kernel)` для поиска директории текущей (свежайшей) версии модуля `kernel`, и затем в поддиректории `include` производится поиск нужного файла `file.hrl`.

2.3 Комментарии

Комментарии могут появляться в любом месте модуля, кроме как внутри строк и атомов, заключённых в кавычки. Комментарий начинается с символа процента (%) и действителен до конца строки но не включая символ конца строки. Завершающий строку символ конца строки, с точки зрения компилятора, действует как пробел.

2.4 Кодировка файлов

Erlang работает с полным набором символов Latin-1 (ISO-8859-1). Таким образом, можно использовать и выводить на экран все печатные символы из набора Latin-1 без цитирования с помощью обратной косой черты (\). Атомы и переменные могут использовать все символы из набора Latin-1.

Примечание: Начиная с версии R16 позволяется использовать кодировку UTF-8 для исходных файлов, но этот режим нужно включать параметром командной строки компилятора. Начиная с версии R17, исходные файлы имеют кодировку UTF-8 по умолчанию.

Классы символов в кодировке			
Восьме-ричное	Деся-тичное		Класс
40 - 57	32 - 47	! " # $ % & ' /	Символы пунктуации
60 - 71	48 - 57	0 - 9	Десятичные цифры
72 - 100	58 - 64	: ; < = > @	Символы пунктуации
101 - 132	65 - 90	A - Z	Заглавные буквы
133 - 140	91 - 96	[\ ] ^ _ `	Символы пунктуации
141 - 172	97 - 122	a - z	Строчные буквы
173 - 176	123 - 126	{ | } ~	Символы пунктуации
200 - 237	128 - 159		Управляющие символы
240 - 277	160 - 191	- ¿	Символы пунктуации
300 - 326	192 - 214	À - Ö	Заглавные буквы
327	215	×	Символ пунктуации
330 - 336	216 - 222	Ø - Þ	Заглавные буквы
337 - 366	223 - 246	ß - ö	Строчные буквы
367	247	÷	Символ пунктуации
370 - 377	248 - 255	ø - ÿ	Строчные буквы

2.5 Зарезервированные слова

Следующие ключевые слова в Erlang зарезервированы, и не могут использоваться в качестве атомов (для использования одного из ключевых слов в качестве атома, оно должно быть заключено в одиночные кавычки):

```
after and andalso band begin bnot bor bsl bsr bxor case
catch cond div end fun if let not of or orelse receive
rem try when xor
```

3 Типы данных (термы)

3.1 Унарные (одиночные) типы данных

3.1.1 Атомы

Атом — это символьное имя, также известное, как *литерал*. Атомы начинаются со строчной латинской буквы и могут содержать буквенно-цифровые символы, подчёркивания (_) и символ *at* (@). Как вариант, атомы могут быть указаны с помощью заключения в одиночные кавычки ('), это необходимо, если атом начинается с заглавной буквы, или содержит другие символы, кроме подчёркиваний и *at* (@). Например:

```
hello
phone_number
'Monday'
'phone number'
```

```
'Anything inside quotes \n\012'
```

(см. раздел 3.3)

3.1.2 Истина и ложь

В Erlang нет специального типа данных для **логических** значений. Эту роль выполняют атомы `true` и `false`.

```
2 =< 3  ⇒  true
true or false  ⇒  true
```

3.1.3 Целые числа

В дополнение к обычному способу записи **целых чисел** Erlang предлагает ещё ряд способов записи. Запись `$Char` даёт числовое значение для символа '`Char`' в кодировке Latin-1 (это также может быть и непечатный символ) и запись `Base#Value` — даёт целое число, записанное с основанием `Base`, основание должно быть целым числом в диапазоне $2..36$.

```
42   ⇒  42
$A   ⇒  65
$\n  ⇒  10                                   (see section 3.3)
2#101  ⇒  5
16#1f  ⇒  31
```

3.1.4 Действительные с плавающей точкой

Действительное число **с плавающей точкой** записывается, как `Основание["e"Экспонента]`, где `Основание` — это действительное число в диапазоне от 0.01 до 10000 и `Экспонента` (необязательное) — это целое число со знаком, указывающее экспоненту (степень десятки, на которую множится `Основание`). Например:

```
2.3e-3  ⇒  2.30000e-3
```
(соответствует $2.3 \cdot 10^{-3}$)

3.1.5 Ссылочные значения

Ссылка — это терм, значение которого является уникальным в системе времени исполнения Erlang, который создаётся встроенной функцией `make_ref/0`. (Для дополнительной информации по встроенным функциям, или *BIF*-ам, смотрите раздел 5.8.)

3.1.6 Порты

Идентификатор порта указывает на открытый в системе порт (смотрите главу 9 про порты).

3.1.7 Идентификаторы процессов (Pid)

Идентификатор процесса, *или pid*, указывает на процесс в системе (смотрите главу 6 о процессах).

3.1.8 Анонимные функции

Тип данных *fun* идентифицирует *анонимную функцию* или **функциональный объект** (см. раздел 5.7).

3.2 Составные типы данных

3.2.1 Кортежи

Кортеж — это составной тип данных, который содержит **фиксированное количество термов**, заключённое {в фигурные скобки}. Например:

```
{Терм1,...,ТермN}
```

Каждый из `ТермX` в кортеже называется **элементом**. Количество элементов называется **размером** данного кортежа.

BIF-функции для работы с кортежами	
`size(Кортеж)`	Возвращает размер `Кортежа`
`element(N,Кортеж)`	Возвращает `N`^{ый} элемент в `Кортеже`
`setelement(N,Кортеж1,Выражение)`	Возвращает новый кортеж, скопированный из `Кортеж1`, в котором `N`^{ый} элемент заменён новым значением `Выражение`

```
P = {adam, 24, {july, 29}}  ⇒  P связано с {adam, 24, {july, 29}}
element(1, P)  ⇒  adam
element(3, P)  ⇒  {july,29}
P2 = setelement(2, P, 25)  ⇒  P2 связано с {adam, 25, {july, 29}}
size(P)  ⇒  3
size({})  ⇒  0
```

3.2.2 Записи

Запись это *именованный кортеж*, имеющий именованные элементы, которые называются **полями**. Тип записи определяется в виде атрибута модуля, например:

```
-record(Запись, {Поле1 [= Значение1],
                 ...
                 ПолеN [= ЗначениеN]}).
```

Здесь имя записи `Запись` и имена полей `ПолеX` — атомы и каждое `ПолеX` может получить необязательное значение по умолчанию `ЗначениеX`. Это определение может быть помещено в любом месте модуля между определениями функций, но обязательно до первого использования. Если тип записи используется в нескольких модулях, рекомендуется поместить его в отдельный файл для включения.

Новая запись с типом `Запись` создаётся с помощью следующего выражения:

```
#Запись{Поле1=Выражение1, ..., ПолеK=ВыражениеK [, _=ВыражениеL]}
```

Поля не обязательно должны идти в том же порядке, как и в определении записи. Пропущенные поля получат свои соответствующие значения по умолчанию. Если использована последняя запись (подчёркивание равно `ВыражениеL`), то все оставшиеся поля получат значение `ВыражениеL`. Поля без значений по умолчанию и пропущенные поля получат значение `undefined`.

Значение поля можно получить используя выражение `Переменная#Запись.Поле`.

```
-module(employee).
-export([new/2]).
-record(person, {name, age, employed=erixon}).

new(Name, Age) -> #person{name=Name, age=Age}.
```

Здесь функция `employee:new/2` может быть использована в другом модуле, который также должен включить файл, содержащий определение использованной здесь записи `person`.

```
{P = employee:new(ernie,44)}  ⇒  {person, ernie, 44, erixon}
P#person.age  ⇒  44
P#person.employed  ⇒  erixon
```

При работе с записями в интерпретаторе Erlang, можно использовать функции `rd(ИмяЗаписи, ОпределениеЗаписи)` и `rr(Модуль)` для того, чтобы определить или загрузить новые определения записей. Подробнее читайте в документации *Erlang Reference Manual*.

3.2.3 Списки

Список — это составной тип данных, который содержит *переменное* количество **термов**, заключённое в квадратных скобках.

```
[Терм1,...,ТермN]
```

Каждый `ТермX` в списке называется **элементом**. Количество элементов в списке называется **длиной списка**. Как это принято в функциональном программировании, первый элемент называется **головой** списка, а остаток (начиная с 2го элемента и до конца) называется **хвостом** списка. Заметьте, что отдельные элементы в списке не должны быть одинакового типа, хотя часто практикуется (и это, наверное, даже удобно), иметь в списке элементы одинакового типа — когда приходится работать с элементами разных типов, обычно используются **записи**.

Встроенные функции для работы со списками	
`length(Список)`	Возвращает длину `Списка`
`hd(Список)`	Возвращает 1^{й} элемент `Списка` (голову)
`tl(Список)`	Возвращает `Список` без 1го элемента (хвост)

Оператор «вертикальная черта» (|, также в некоторых книгах по ФП он называется `cons`), отделяет ведущие элементы списка (один или более) от остальных элементов. Например:

```
[H | T] = [1, 2, 3, 4, 5]  ⇒  H=1 и T=[2, 3, 4, 5]
[X, Y | Z] = [a, b, c, d, e]  ⇒  X=a, Y=b и Z=[c, d, e]
```

Список — рекурсивная структура. Неявно список завершается ссылкой на пустой список, то есть `[a,b]` это то же самое, как и `[a,b|[]]`. Список, выглядящий как `[a,b|c]` называется **плохо сформированным** (badly formed) и такой записи следует избегать (потому что атом 'c', завершающий структуру списка сам *не* является списком). Списки натурально способствуют рекурсивному функциональному программированию.

Например, следующая функция `sum` вычисляет сумму списка и функция `double` умножает каждый элемент списка на 2, при этом конструируя и возвращая новый

список по ходу выполнения.

```
sum([]) -> 0;
sum([H | T]) -> H + sum(T).

double([]) -> [];
double([H | T]) -> [H*2 | double(T)].
```

Определения функций выше представляют *сопоставление с образцом*, которое описано далее в главе 4. Образцы в такой записи часто встречаются в рекурсивном программировании, неявно предоставляя «базовый случай» (для пустого списка в этих примерах).

Для работы со списками, оператор `++` соединяет вместе два списка (присоединяет второй аргумент к первому) и возвращает список-результат. Оператор `--` создаёт список, который является копией первого аргумента, за тем исключением, что для каждого элемента во втором списке его первое вхождение в результат (если такое было) удаляется.

`[1,2,3] ++ [4,5]` ⇒ `[1,2,3,4,5]` `[1,2,3,2,1,2] -- [2,1,2]` ⇒ `[3,1,2]`

Подборка функций, работающих со списками может быть найдена в модуле стандартной библиотеки с названием `lists`.

3.2.4 Строки

Строки — это цепочки символов, заключённые между двойными кавычками, на самом деле они хранятся в памяти, как списки целых чисел — символов.

`"abcdefghi"` то же самое, как и `[97,98,99,100,101,102,103,104,105]`

`""` то же самое, как и `[]`

Две строки, записанные подряд без разделительных знаков и операторов будут соединены в одну во время компиляции и не принесут дополнительных затрат по соединению во время исполнения.

`"string42"` ⇒ `"string42"`

3.2.5 Двоичные данные

Двоичные данные — это блок нетипизированной памяти, по умолчанию двоичные данные являются последовательностью 8-битных элементов (байтов).

```
<<Элемент1,...,ЭлементN>>
```

Каждый `ЭлементX` указывается в виде

```
Значение[:Размер][/СписокСпецификацийТипа]
```

Спецификация элемента двоичных данных		
`Значение`	`Размер`	`СписокСпецификацийТипа`
Выражение должно вычисляться в целое, действительное число или двоичные данные.	Выражение должно вычисляться в целое число	Последовательность необязательных спецификаторов типа, в любом порядке, разделённых дефисами (-)

Спецификаторы типов		
Тип данных	`integer` | `float` | `binary`	По умолчанию `integer`
Наличие знака	`signed` | `unsigned`	По умолчанию без знака, `unsigned`
Порядок байтов (endianness)	`big` | `little` | `native`	Зависит от архитектуры процессора. По умолчанию `big`
Единица измерения (группа битов)	`unit:`*ЦелоеЧисло*	Разрешены значения от 1 до 256. По умолчанию 1 для целых и действительных чисел и 8 для двоичных данных

Значение `Размера` умножается на единицу измерения и даёт число бит, которые может занять данный сегмент двоичных данных. Каждый сегмент может состоять из нуля или более битов, но общая сумма битов должна делиться нацело на 8, иначе во время исполнения возникнет ошибка `badarg`. Также сегмент с типом `binary` должен иметь размер, делящийся нацело на 8.

Двоичные данные не могут вкладываться друг в друга.

```
<<1, 17, 42>>        % <<1, 17, 42>>
<<"abc">>            % <<97, 98, 99>> (То же, что и <<$a, $b, $c>>)
<<1, 17, 42:16>>     % <<1,17,0,42>>
<<>>                 % <<>>
<<15:8/unit:10>>     % <<0,0,0,0,0,0,0,0,0,15>>
<<(-1)/unsigned>>    % <<255>>
```

3.3 Escape-последовательности

Escape-последовательности разрешено использовать в строках и атомах, которые заключены в кавычки, они позволяют ввести в исходный текст программы символ, который другим способом ввести трудно или невозможно.

Escape-последовательности	
\b	Backspace (удаление слева)
\d	Delete (удаление справа)
\e	Escape
\f	Новая страница (при печати)
\n	Новая строка
\r	Возврат курсора в начало
\s	Пробел
\t	Табуляция
\v	Вертикальная табуляция
\XYZ, \XY, \X	Символ, записанный в восьмеричном представлении, как XYZ, XY или X
\^A .. \^Z	Комбинации клавиш от Ctrl-A до Ctrl-Z
\^a .. \^z	Комбинации клавиш от Ctrl-A до Ctrl-Z
\’	Единичная кавычка
\”	Двойная кавычка
\\	Обратная косая черта (\)

3.4 Преобразования типов

Erlang — строго типизированный язык, то есть неявные автоматические преобразования типов в нём не происходят. Но есть ряд встроенных в стандартную биб-

лиотеку функций, предназначенных для преобразования между типами данных при участии программиста:

Преобразования типов								
	atom	integer	float	pid	fun	tuple	list	binary
atom		-	-	-	-	-	X	X
integer	-		X	-	-	-	X	X
float	-	X		-	-	-	X	X
pid	-	-	-		-	-	X	X
fun	-	-	-	-		-	X	X
tuple	-	-	-	-	-		X	X
list	X	X	X	X	X	X		X
binary	X	X	X	X	X	X	X	

Встроенная функция `float/1` переводит целые числа в числа с плавающей точкой. Встроенные функции `round/1` и `trunc/1` переводят числа с плавающей точкой обратно в целые, округляя или отбрасывая дробную часть.

Функции `Тип_to_list/1` и `list_to_Тип/1` переводят различные типы в списки (строки) и из списков.

Функции `term_to_binary/1` и `binary_to_term/1` переводят любое значение в закодированную двоичную форму и обратно (Подробнее: http://erlang.org/doc/apps/erts/erl_ext_dist.html).

```
atom_to_list(hello)         % "hello"
list_to_atom("hello")       % hello
float_to_list(7.0)          % "7.00000000000000000000e+00"
list_to_float("7.000e+00") % 7.00000
integer_to_list(77)         % "77"
list_to_integer("77")       % 77
tuple_to_list({a, b ,c})    % [a,b,c]
list_to_tuple([a, b, c])    % {a,b,c}
pid_to_list(self())         % "<0.25.0>"
term_to_binary(<<17>>)      % <<131,109,0,0,0,1,17>>
term_to_binary({a, b ,c})   % <<131,104,3,100,0,1,97,100,0,1,98,
                            %   100,0,1,99>>
binary_to_term(<<131,104,3,100,0,1,97,100,0,1,98,100,0,1,99>>)
                            % {a,b,c}
term_to_binary(math:pi())   % <<131,99,51,46,49,52,49,53,57,50,...
```

4 Сопоставление с образцом

4.1 Переменные

Переменные представлены, как аргументы функции или как результат сопоставления с образцом. Переменные начинаются с заглавной буквы или символа подчёркивания (_) и могут содержить буквенно-цифровые символы, подчёркивания и символы *at* (@). Переменные могут быть связаны со значением (присвоены) только один раз.

```
Abc
A_long_variable_name
AnObjectOrientatedVariableName
_Height
```

Анонимная переменная объявляется с помощью одного символа подчёркивания (_) и может использоваться там, где требуется переменная, но её значение нас не интересует и может быть проигнорировано.

```
[H|_] = [1,2,3]          % H=1 и всё остальное игнорируется
```

Переменные, начинающиеся с символа подчёркивания, как, например, `_Height`, являются обычными не анонимными переменными. Однако они игнорируются компилятором в том смысле, что они не произведут предупреждений компилятора о неиспользуемых переменных. Таким образом, возможна следующая запись:

```
member(_Elem, []) ->
    false.
```

вместо:

```
member(_, []) ->
    false.
```

что улучшает читаемость кода.

Область видимости для переменной — это её уравнение функции. Переменные, связанные со значением в ветке `if`, `case` или `receive`, должны быть связаны с чем-нибудь во всех ветвях этого оператора, чтобы иметь значение за пределами выражения, иначе компилятор будет считать это значение *небезопасным* (unsafe) (вероятно, не присвоенным) за пределами этого выражения, и выдаст соответствующее предупреждение.

4.2 Сопоставление с образцом

Образец имеет такую же структуру, как и терм, но может содержать новые свободные переменные. Например:

```
Name1
[H|T]
{error,Reason}
```

Образцы могут встречаться в *заголовках функций*, выражениях *case*, *receive*, и *try* и в выражениях оператора сопоставления (`=`). Образцы вычисляются посредством **сопоставления образца** с выражением, и таким образом новые переменные определяются и связываются со значением.

```
Образец = Выражение
```

Обе стороны выражения должны иметь одинаковую структуру. Если сопоставление проходит успешно, то все свободные переменные (если такие были) в образце слева становятся связанными. Если сопоставление не проходит, то возникает ошибка времени исполнения `badmatch`.

```
1> {A, B} = {answer, 42}.
{answer,42}
2> A.
answer
3> B.
42
```

4.2.1 Оператор сопоставления (=) в образцах

Если `Образец1` и `Образец2` являются действительными образцами, тогда следующая запись тоже действительный образец:

```
Образец1 = Образец2
```

Оператор = представляет собой **подмену** (alias), при сопоставлении которой с выражением, оба и `Образец1` и `Образец2` также сопоставляются с ней. Цель этого — избежать необходимости повторно строить термы, которые были разобраны на составляющие в сопоставлении.

```
foo({connect,From,To,Number,Options}, To) ->
    Signal = {connect,From,To,Number,Options},
    fox(Signal),
    ...;
```

можно более эффективно записать, как:

```
foo({connect,From,To,Number,Options} = Signal, To) ->
    fox(Signal),
    ...;
```

4.2.2 Строковой префикс в образцах

При сопоставлении строк с образцом, следующая запись является действительным образцом:

```
f("prefix" ++ Str) -> ...
```

что эквивалентно и легче читается, чем следующая запись:

```
f([$p,$r,$e,$f,$i,$x | Str]) -> ...
```

Вы можете использовать строки только как префикс; варианты с постфиксом для образцов, такие как `Str ++ "postfix"` не разрешаются.

4.2.3 Выражения в образцах

Арифметическое выражение может быть использовано внутри образца, если оно использует только числовые, битовые операторы, и его значение является константой, которая может быть вычислена во время компиляции.

```
case {Value, Result} of
    {?Threshold+1, ok} -> ...   % ?Threshold - это макрос
```

4.2.4 Сопоставление двоичных данных

```
Bin = <<1, 2, 3>>       % <<1,2,3>> Все элементы - 8-битные байты
<<A, B, C>> = Bin       % A=1, B=2 и C=3
<<D:16, E>> = Bin       % D=258 и E=3
<<F, G/binary>> = Bin   % F=1 и G=<<2,3>>
```

В последней строке переменная `G` неуказанного размера сопоставляется с остатком двоичных данных `Bin`.

Всегда ставьте пробел между оператором (=) и (<<), чтобы избежать возможной путаницы с оператором (=<).

5 Функции

5.1 Определение функции

Функция определяется, как последовательность из одного или нескольких **уравнений функции**. Имя функции должно быть атомом.

```
Функция(Образец11,...,Образец1N)
  [when ОхранныеВыражения1] -> ТелоФункции1;
...;
...;
Функция(ОбразецK1,...,ОбразецKN)
  [when ОхранныеВыраженияK] -> ТелоФункцииK.
```

Уравнения функции разделены точками с запятой (`;`) и последнее уравнение завершается точкой (`.`). Уравнение функции состоит из **заголовка уравнения** и **тела уравнения функции**, разделённых стрелкой (`->`).

Заголовок уравнения состоит из имени функции (атома), списка аргументов, заключённого в скобки, и необязательного списка охранных выражений, начинающихся с ключевого слова `when`. Каждый аргумент функции — образец. Тело функции состоит из последовательности выражений, разделённых запятыми (`,`).

```
Выражение1,
...,
ВыражениеM
```

Количество аргументов `N` ещё называется **арностью** функции. Функция уникально определяется именем модуля, именем функции и своей арностью. Две разные функции в одном модуле, имеющие разную арность, могут иметь одно и то же имя. `Функция` в `Модуле` с арностью `N` часто может записываться так: `Модуль:Функция/N`.

```
-module(mathStuff).
-export([area/1]).

area({square, Side}) -> Side * Side;
area({circle, Radius}) -> math:pi() * Radius * Radius;
area({triangle, A, B, C}) ->
    S = (A + B + C)/2,
    math:sqrt(S*(S-A)*(S-B)*(S-C)).
```

5.2 Вызовы функций

Функция вызывается с помощью записи:

```
[Модуль:]Функция(Выражение1, ..., ВыражениеN)
```

Выражение `Модуль` должно вычисляться в имя модуля (или быть атомом) и выражение `Функция` должно вычисляться в имя функции или в *анонимную функцию*. При вызове функции в другом модуле, следует указать имя модуля и функция должна быть экспортирована. Такой вызов будет называться **полностью определённым вызовом функции**.

```
lists:keysearch(Name, 1, List)
```

Имя модуля может быть опущено, если `Функция` вычисляется в имя локальной функции, импортированной функции или авто-импортированной встроенной (BIF) функции. Такой вызов называется **неявно определённым вызовом функции**.

Перед тем, как вызвать функцию, вычисляются аргументы `ExprX`. Если функция не может быть найдена, то возникает ошибка времени исполнения `undef`. Если уравнений функции несколько, они последовательно сканируются до тех пор, пока не будет найдено подходящее уравнение, такое, что образцы в заголовке уравнения успешно могут быть сопоставлены с данными аргументами и что охранное выражение (если оно задано) равно `true`. Если такое уравнение не может быть найдено, возникает ошибка времени исполнения `function_clause`.

Если совпадающее уравнение найдено, то вычисляется соответствующее тело функции, то есть выражения в теле функции вычисляются одно за другим и возвращается результат последнего выражения.

Полностью определённое имя функции должно быть использовано, если вызывается встроенная функция с таким же именем (см. секцию 5.8 о встроенных BIF функциях). Компилятор не разрешит определить функцию с таким же именем, как другая импортированная функция. При вызове локальной функции есть разница между использованием неявно или полностью определённого имени функции, поскольку второе всегда относится к последней версии модуля (см. главу 10 о модулях и версиях).

5.3 Выражения

Выражение это терм либо вызов оператора, результатом которого будет терм, например:

```
Терм
операция Выражение
Выражение1 операция Выражение2
(Выражение)
begin
   Выражение1,
   ...,
   ВыражениеМ            % нет запятой (,) перед end
end
```

В наличии имеются как *унарные* так и *бинарные* операторы. Простейшая форма выражения — это терм, например *целое число*, *число с плавающей точкой*, *атом*, *строка*, *список* или *кортеж*, и возвращаемое оператором значение тоже терм. Выражение может содержать *макрос* или операции над *записью*, которые будут развёрнуты во время компиляции.

Выражения в скобках полезны для изменения порядка вычисления операторов (см. раздел 5.3.5):

```
1 + 2 * 3           % 7
(1 + 2) * 3         % 9
```

Блочные выражения, заключённые в операторные скобки между `begin...end`, могут использоваться для группировки последовательности выражений и возвращают значение, равное значению последнего выражения внутри `ВыражениеМ`.

Все вложенные подвыражения вычисляются до главного выражения, но порядок, в котором происходит вычисление вложенных, не определён стандартом.

Многие операторы могут применяться только к аргументам определённого типа. Например, арифметические операторы могут только применяться к целым числам или числам с плавающей точкой. Аргумент неверного типа вызовет ошибку времени выполнения `badarg`.

5.3.1 Сравнение термов

```
Выражение1 оператор Выражение2
```

Сравнение термов возвращает *булево* (логическое) значение, в форме атомов `true` или `false`.

Операторы сравнения			
`==`	Равно (с приведением типа)	`=<`	Меньше или равно
`/=`	Не равно (с приведением типа)	`<`	Меньше, чем
`=:=`	Точно равно (с учётом типа)	`>=`	Больше или равно
`=/=`	Точно не равно (с учётом типа)	`>`	Больше, чем

```
1==1.0                % true
1=:=1.0               % false
1 > a                 % false
```

Аргументы оператора сравнения могут иметь разные типы данных. В таком случае действует следующий порядок сравнения:

```
число < атом < ссылка < анонимная функция < порт < pid < кортеж < список
< двоичные данные
```

Списки сравниваются поэлементно. Кортежи сравниваются по размеру, два кортежа одного размера сравниваются поэлементно. При сравнении целого числа и числа с плавающей точкой, целое сначала приводится к числу с плавающей точкой. В случае использования точного равенства `=:=` или `=/=` тип числа не изменяется и учитывается в равенстве.

5.3.2 Арифметические выражения

```
оператор Выражение
Выражение1 оператор Выражение2
```

Арифметическое выражение возвращает результат после применения оператора.

Арифметические операторы		
`+`	Унарный плюс	`Integer` | `Float`
`-`	Унарный минус	`Integer` | `Float`
`+`	Сложение	`Integer` | `Float`
`-`	Вычитание	`Integer` | `Float`
`*`	Умножение	`Integer` | `Float`
`/`	Деление с плавающей точкой	`Integer` | `Float`
`bnot`	Унарное битовое НЕ	`Integer`
`div`	Целочисленное деление	`Integer`
`rem`	Целочисленный остаток от деления X на Y	`Integer`
`band`	Битовое И	`Integer`
`bor`	Битовое ИЛИ	`Integer`
`bxor`	Исключающее битовое ИЛИ	`Integer`
`bsl`	Арифметический (с учётом знака) битовый сдвиг влево	`Integer`
`bsr`	Битовый сдвиг вправо	`Integer`

```
+1                  % 1
4/2                 % 2.00000
5 div 2             % 2
5 rem 2             % 1
2#10 band 2#01      % 0
2#10 bor 2#01       % 3
```

5.3.3 Логические (булевы) выражения

```
оператор Выражение
Выражение1 оператор Выражение2
```

Логическое выражение возвращает значение `true` или `false` после применения оператора.

Логические (булевы) операторы	
`not`	Унарное логическое НЕ (отрицание)
`and`	Логическое И
`or`	Логическое ИЛИ
`xor`	Логическое исключающее ИЛИ

```
not true            % false
true and false      % false
true xor false      % true
```

5.3.4 Умные логические выражения

```
Выражение1 orelse Выражение2
Выражение1 andalso Выражение2
```

В этих логические выражениях второй операнд вычисляется только в том случае, если его значение необходимо для конечного результата. В случае с `orelse` `Выражение2` будет вычислено, только если `Выражение1` равняется `false`. В случае с `andalso` `Выражение2` будет вычислено, только если `Выражение1` равняется `true`.

```
if A >= 0 andalso math:sqrt(A) > B -> ...

if is_list(L) andalso length(L) == 1 -> ...
```

5.3.5 Приоритет операторов

В выражении, состоящем из подвыражений, операторы будут применяться согласно определённому порядку, который называется **приоритетом операторов**:

Приоритет операторов (от высшего к низшему)	
`:`	
`#`	
`Unary + - bnot not`	
`/ * div rem band and`	Левоассоциативный
`+ - bor bxor bsl bsr or xor`	Левоассоциативный
`++ --`	Правоассоциативный
`== /= =< < >= > =:= =/=`	
`andalso`	
`orelse`	
`= !`	Правоассоциативный
`catch`	

Оператор с наивысшим приоритетом вычисляется первым. Операторы с одинаковым приоритетом вычисляются согласно их **ассоциативности**. Левоассоциативные операторы вычисляются слева направо, правоассоциативные — наоборот, справа налево:

```
6 + 5 * 4 - 3 / 2  ⇒ 6 + 20 - 1.5  ⇒ 26 - 1.5  ⇒ 24.5
```

5.4 Составные выражения

5.4.1 If

```
if
    ОхранноеВыражение1 ->
        Тело1;
    ...;
    ОхранноеВыражениеN ->
        ТелоN      % Заметьте, нет точки с запятой (;) перед end
end
```

Ветки выражения `if` сканируются последовательно сверху вниз, пока не встретится `ОхранноеВыражениеX`, результатом которого будет `true`. Соотвествующее `ТелоX` (последовательность выражений, разделённых запятыми) будет вычислено. Возвращаемое `ТеломX` значение будет результатом всего выражения `if`.

Если ни одно из охранных выражений не вычисляется в `true`, то возникнет ошибка времени выполнения `if_clause`. При необходимости можно использовать охранное выражение `true` в последней ветке `if`, поскольку такое охранное выражение всегда срабатывает, если другие не сработали и называется «срабатывающим для всех значений» (catch all).

```
is_greater_than(X, Y) ->
    if
        X>Y ->
            true;
        true ->                 % работает как ветка 'else'
            false
    end
```

Следует заметить, что сопоставление с образцом в уравнениях функций может почти всегда использоваться для замены `if`. Чрезмерное использование `if` внутри функций считается плохой практикой.

5.4.2 Case

Выражения `case` используются для сопоставления с образцом прямо в коде, подобно тому, как сопоставляются аргументы в заголовках функций.

```
case Выражение of
    Выражение1 [when ОхранноеВыражение1] ->
        Тело1;
        ...;
    ВыражениеN [when ОхранноеВыражениеN] ->
        ТелоN       % Заметьте, нет точки с запятой (;) перед end
end
```

`Выражение` вычисляется и `Образец1...ОбразецN` последовательно сопоставляются с результатом. Если сопоставление проходит успешно, и необязательное `ОхранноеВыражениеX` равно `true`, тогда вычисляется соответствующее `ТелоX`. Возвращаемое значение `ТелаX` является результатом всего выражения `case`.

В случае, если ни один из образцов и их охранных выражений не подходит, возникает ошибка времени выполнения `case_clause`.

```
is_valid_signal(Signal) ->
    case Signal of
        {signal, _What, _From, _To} ->
            true;
        {signal, _What, _To} ->
            true;
        _Else ->                % 'catch all'
            false
    end.
```

5.4.3 Генераторы списков

Генераторы списков аналогичны предикатам `setof` и `findall` в языке Prolog.

```
[Выражение || Квалификатор1,...,КвалификаторN]
```

`Выражение` — произвольное выражение, и каждый `КвалификаторX` — это либо **генератор** (источник данных) либо **фильтр**. Генератор записывается, как:

```
Образец <- ВыражениеСписок
```

где `ВыражениеСписок` должно быть выражением, результатом которого является список термов. Фильтр — это выражение, результатом которого является `true` или `false`. Переменные внутри генератора списков *затеняют* переменные с такими же именами, принадлежащие функции, окружающей генератор.

Квалификаторы вычисляются слева направо, генераторы создают значения и фильтры отбирают нужные из них. Генератор списков возвращает список, в котором элементы являются результатом вычисления `Выражения` для каждой комбинации результирующих значений.

```
1> [{X, Y} || X <- [1,2,3,4,5,6], X > 4, Y <- [a,b,c]].
[{5,a},{5,b},{5,c},{6,a},{6,b},{6,c}]
```

5.5 Охранные последовательности

Охранная последовательность — это набор **охранных выражений**, разделённых точками с запятой (`;`). Охранная последовательность равняется `true`, если как минимум одно из составляющих её охранных выражений равно `true`.

```
Охрана1; ...; ОхранаK
```

Охрана — это множество **охранных выражений**, разделённых запятыми (`,`). Результатом будет `true` если все охранные выражения равняются `true`.

```
ОхранноеВыражение1, ..., ОхранноеВыражениеN
```

В **охранных выражениях**, которые иногда ещё называются охранными тестами, разрешены не любые выражения Erlang, а ограниченный набор, выбранный авторами Erlang, поскольку важно, чтобы ход вычисления охранного выражения не имел побочных эффектов.

Разрешённые охранные выражения:	
Атом `true`; Другие константы (термы, связанные переменные), все считаются равными `false`;	
Сравнения термов; Арифметические и логические выражения; Вызовы встроенных (BIF) функций, перечисленные ниже:	
Функции проверки типа	Другие позволенные функции:
`is_atom/1`	`abs(Integer|Float)`
`is_constant/1`	`float(Term)`
`is_integer/1`	`trunc(Integer|Float)`
`is_float/1`	`round(Integer|Float)`
`is_number/1`	`size(Tuple|Binary)`
`is_reference/1`	`element(N, Tuple)`
`is_port/1`	`hd(List)`
`is_pid/1`	`tl(List)`
`is_function/1`	`length(List)`
`is_tuple/1`	`self()`
`is_record/2` Второй аргумент имя записи	`node()`
`is_list/1`	`node(Pid|Ref|Port)`
`is_binary/1`	

Небольшой пример

```
fact(N) when N>0 ->             % Заголовок первого уравнения
    N * fact(N-1);              % Тело первого уравнения
fact(0) ->                      % Заголовок второго уравнения
    1.                          % Тело второго уравнения
```

5.6 Хвостовая рекурсия

Если последнее выражение тела функции является вызовом функции, то выполняется **хвостовой рекурсивный** вызов так, что ресурсы системы (например, стек вызовов) не расходуются. Это означает, что можно создать бесконечный цикл, такой, как, например, сервер, если использовать хвостовые рекурсивные вызовы.

Функция `fact/1` выше может быть переписана для использования хвостовой рекурсии следующим образом:

```
fact(N) when N>1 -> fact(N, N-1);
fact(N) when N==1; N==0 -> 1.

fact(F,0) -> F;       % Переменная F используется как аккумулятор
fact(F,N) -> fact(F*N, N-1).
```

5.7 Анонимные функции

Ключевое слово **fun** определяет *функциональный объект*. Анонимные функции делают возможным передавать целую функцию, а не только её имя, в качестве аргумента. Выражение, определяющее анонимную функцию, начинается с ключевого слова `fun` и заканчивается ключевым словом `end` вместо точки (`.`). Между ними должно находиться обычное объявление функции, за тем исключением, что имя её не пишется.

```
fun
    (Образец11,...,Образец1N) [when Охрана1] ->
        ТелоУравнения1;
        ...;
    (ОбразецK1,...,ОбразецKN) [when ОхранаK] ->
        ТелоУравненияK
end
```

Переменные в заголовке анонимной функции *затеняют* переменные в уравнении функции, которое окружает `fun`, но переменные, связанные в теле `fun` являются для него локальными. Возвращаемое выражением `fun Имя/N` значение является функцией. Выражение `fun Имя/N` эквивалентно следующей записи:

```
fun (Аргумент1,...,АргументN) -> Имя(Аргумент1,...,АргументN) end
```

Выражение `fun Модуль:Функция/Арность` также разрешено, но только если `Функция` экспортирована из `Модуля`.

```
Fun1 = fun (X) -> X+1 end.
Fun1(2)          % результат: 3

Fun2 = fun (X) when X>=1000 -> big; (X) -> small end.
Fun2(2000)       % результат: big
```

Поскольку функция, созданная `fun` анонимна, то есть не имеет имени в определении функции, то для определения рекурсивной функции следует сделать два шага. Пример ниже показывает, как определить рекурсивную функцию `sum(List)` (смотрите раздел 3.2.3) как анонимную с помощью `fun`. Такой подход ещё называется *Y-комбинатор*.

```
Sum1 = fun ([], _Foo) -> 0;([H|T], Foo) -> H + Foo(T, Foo) end.
Sum = fun (List) -> Sum1(List, Sum1) end.
Sum([1,2,3,4,5])    % 15
```

Определение `Sum1` сделано так, что оно принимает *само себя* в качестве аргумента, сопоставляется с `_Foo` (пустым списком) или `Foo`, который затем рекурсивно вызывается. Определение `Sum` вызывает `Sum1`, также передавая `Sum1` в качестве аргумента.

Примечание: В Erlang версии R17 эта проблема устранена и анонимные функции могут ссылаться сами на себя.

5.8 Встроенные функции (BIF)

Встроенные функции, или BIF (built-in functions) — это функции, реализованные на языке C, на котором также написана система Erlang, и делают вещи, которые трудно или невозможно реализовать на языке Erlang. Большинство встроенных функций принадлежат модулю `erlang`, но есть некоторые, принадлежащие и другим модулям, таким как `lists` и `ets`. Используемые чаще всего встроенные функции, принадлежащие модулю `erlang`, импортируются во все ваши модули автоматически, то есть перед ними не требуется писать имя модуля.

Некоторые полезные встроенные функции	
`date()`	Возвращает сегодняшнюю дату в формате `{Год, Месяц, День}`
`now()`	Возвращает текущее время в микросекундах. Точность и реализация зависит от операционной системы
`time()`	Возвращает текущее время в формате `{Часы, Минуты, Секунды}`. Реализация зависит от операционной системы
`halt()`	Аварийно останавливает работу Erlang-системы
`processes()`	Возвращает список всех Erlang-процессов в системе
`process_info(Pid)`	Возвращает словарь, содержащий информацию о процессе `Pid`
`Модуль:module_info()`	Возвращает словарь, содержащий информацию о `Модуле`

Словарь — это список кортежей в формате `{Ключ, Значение}` (см. также раздел 6.8).

```
size({a, b, c})              % 3
atom_to_list('Erlang')       % "Erlang"
date()                       % {2013,5,27}
time()                       % {01,27,42}
```

6 Процессы

Процесс соответствует одному *потоку управления*. Erlang разрешает создавать очень большое количество параллельно работающих процессов, каждый из которых исполняется, как будто он имеет свой собственный виртуальный процессор. Когда процесс, исполняющийся внутри функции `functionA` вызывает другую функцию `functionB`, он будет ждать, пока `functionB` не завершится и затём извлечёт или получит результат. Если вместо этого он *породит* новый процесс, исполняющий ту же `functionB`, то оба процесса продолжат исполняться одновременно (конкурентно). `functionA` не будет ждать завершения `functionB` и единственный способ передать результат — это *передача сообщений*.

Процессы Erlang — очень лёгкие с малым расходом памяти, легко стартуют и легко завершают работу, и расходы на их планировку во время выполнения очень небольшие. **Идентификатор процесса**, или `Pid`, идентифицирует существующий или недавно существовавший процесс. Встроенная функция `self/0` возвращает `Pid` вызвавшего её процесса.

6.1 Создание процессов

Процесс создаётся с помощью встроенной функции `spawn/3`.

```
spawn(Модуль, Функция, [Аргумент1, ..., АргументN])
```

Аргумент `Модуль` должен быть равен имени модуля, который содержит нужную функцию, и `Функция` — имени экспортированной функции в этом модуле. Список `Аргумент1`…`АргументN` — параметры, которые будут переданы запущенной

в новом процессе функции. `spawn` создаёт новый процесс и возвращает его идентификатор, `Pid`. Новый процесс начинается с выполнения такого кода:

```
Модуль:Функция(Аргумент1, ..., АргументN)
```

`Функция` должна быть экспортирована, даже если процесс с ней порождается другой функцией в том же модуле. Есть и другие встроенные функции для порождения процессов, например `spawn/4` порождает процесс на другом узле Erlang.

6.2 Зарегистрированные процессы

Процесс может быть связан с некоторым именем. Имя процесса должно быть атомом и оно автоматически освобождается, если процесс завершает свою работу. Следует регистрировать только статические (постоянно живущие) процессы.

Встроенные функции для регистрации имён	
`register(Имя, Pid)`	Назначает атом `Имя` в качестве имени для процесса `Pid`
`registered()`	Возвращает список имён, которые были зарегистрированы
`whereis(Name)`	Возвращает `Pid`, который был зарегистрирован для имени `Имя` или атом `undefined` если имя не было зарегистрировано

6.3 Сообщения между процессами

Процессы сообщаются друг с другом посредством отправки и получения **сообщений**. Сообщения отправляются используя оператор отправки (`!`) и принимаются с помощью конструкции `receive`. Передача сообщений *асинхронная* (не блокирует отправителя до доставки сообщения) и *надёжная* (сообщение гарантированно достигает получателя, если он существует).

6.3.1 Отправка

```
Pid ! Выражение
```

Оператор отправки (`!`) посылает значение `Выражения` в форме сообщения процессу, указанному идентификатором `Pid`, где сообщение будет помещено в конец его **очереди сообщений**. Значение `Выражения` также будет значением, возвращённым оператором (`!`). `Pid` должен быть идентификатором процесса, зарегистрированным в системе именем или кортежем в виде `{Имя,Узел}`, где `Имя` — это зарегистрированное имя процесса на удалённом `Узле` (см. главу 8). Оператор отправки сообщения (`!`) не может возвратить ошибку, даже если в качестве получателя был указан несуществующий процесс.

6.3.2 Получение

```
receive
    Образец1 [when ОхранныеВыражения1] ->
        Тело1;
    ...
    ОбразецN [when ОхранныеВыраженияN] ->
        ТелоN      % Заметьте, нет точки с запятой (;) перед end
end
```

Это выражение принимает сообщения, отправленные процессу с помощью оператора отправки (`!`). `ОбразцыX` последовательно сопоставляются с первым сообщением в очереди сообщений, затем со вторым и так далее. Если сопоставление проходит успешно и необязательный список охранных выражений `ОхранныеВыраженияX` тоже равен `true`, то сообщение удаляется из очереди сообщений и соответствующая цепочка выражений `ТелоX` вычисляется. Именно порядок уравнений с образцами решает порядок получения сообщений, а не порядок, в котором они прибывают. Это называется *избирательным приёмом* сообщений. Значение, возвращаемое `ТеломX` и будет значением, возвращённым всем выражением `receive`.

`receive` никогда не приводит к возникновению ошибки. Процесс может быть поставлен на паузу во время ожидания, возможно навсегда, до тех пор, пока не появится сообщение, отвечающее одному из образцов с охранной последовательностью, равной `true`.

```
wait_for_onhook() ->
    receive
        onhook ->
            disconnect(),
            idle();
        {connect, B} ->
            B ! {busy, self()},
            wait_for_onhook()
    end.
```

6.3.3 Получение с таймаутом

```
receive
    Образец1 [when ОхранныеВыражения1] ->
        Тело1;
        ...;
    ОбразецN [when ОхранныеВыраженияN] ->
        ТелоN
after
    ВыражениеТ ->
        ТелоТ
end
```

ВыражениеТ должно вычисляться в целое число между `0` и `16#ffffffff` (значение должно помещаться в 32 бита). Если ни одно подходящее сообщение не прибыло в течение ВыражениеТ миллисекунд, то выражение ТелоТ вычисляется и его возвращаемое значение становится результатом всего выражения `receive`.

```
wait_for_onhook() ->
    receive
        onhook ->
            disconnect(),
            idle();
        {connect, B} ->
            B ! {busy, self()},
            wait_for_onhook()
    after
        60000 ->
            disconnect(),
            error()
    end.
```

Выражение `receive...after` без образцов может быть использовано для реализации простых таймаутов.

```
receive
after
    ВыражениеТ ->
        ТелоТ
end
```

Два особых случая для значения таймаута `ВыражениеТ`	
`infinity`	Является эквивалентом бесконечного ожидания и может пригодиться, если значение таймаута вычисляется во время исполнения и передаётся параметром в функцию
`0`	Если в почтовом ящике нет подходящего сообщения, таймаут произойдёт немедленно

6.4 Завершение работы процесса

Процесс всегда завершается по некоторой **причине выхода** (exit reason), которая может быть любым термом Erlang. Если процесс завершился нормально, то есть его код исполнился до конца, то причиной выхода будет атом `normal`. Процесс может завершить себя сам вызывая одну из следующих встроенных функций:

```
exit(Причина)
erlang:error(Причина)
erlang:error(Причина, Аргументы)
```

Процесс завершается с причиной выхода `{Причина,СтекВызовов}`, когда происходит ошибка времени исполнения.

Процесс также может быть завершён, если он получает сигнал выхода с любой другой причиной кроме `normal` (см. раздел 6.5.3).

6.5 Связи между процессами

Два процесса могут быть **связаны** друг с другом. Связи двунаправленны и может существовать только одна связь между любыми двумя процессами (имеются в виду уникальные идентификаторы процессов). Процесс с идентификатором `Pid1` может связаться с процессом, имеющим идентификатор `Pid2` используя встроенную функцию `link(Pid2)`. Функция `spawn_link(Модуль, Функция, Аргументы)` порождает и сразу же связывает процессы одной атомарной операцией.

Связь между процессами можно убрать используя функцию `unlink(Pid)`.

6.5.1 Обработка ошибок между процессами

Когда процесс завершает работу, он отправляет **сигналы выхода** всем процессам, с которыми он связан. Они в свою очередь также завершают работу *или обрабатывают сигнал выхода каким-либо способом*. Эта возможность может использоваться для построения иерархических программных структур, где некоторые из процессов присматривают за другими процессами, например рестартуя их, если они завершаются аварийно.

6.5.2 Отправка сигналов выхода

Процесс всегда завершает работу с причиной выхода, которая отправляется в виде сигнала выхода всем связанным процессам. Встроенная функция `exit(Pid, Причина)` посылает сигнал выхода другому процессу `Pid` по указанной `Причине`, не влияя на процесс-отправитель.

6.5.3 Получение сигналов выхода

Если процесс получает сигнал выхода с причиной выхода другой, кроме как `normal`, он также завершит свою работу и отправит сигналы выхода с той же причиной всем своим связанным процессам. Сигнал выхода с причиной `normal` игнорируется и не приводит к такому поведению. Это поведение можно изменить с помощью вызова встроенной функции `process_flag(trap_exit, true)`.

Процесс после этого способен **перехватывать сигналы выхода**. Это означает, что сигнал выхода трансформируется в обычное сообщение: `{'EXIT', PidОтправителя, Причина}`, который помещается в почтовый ящик процесса и может быть принят и обработан, как обычное сообщение используя `receive`.

Однако, вызов встроенной функции `exit(Pid, kill)` завершает работу процесса `Pid` безусловно, независимо от того, способен ли он перехватывать сигналы выхода или нет.

6.6 Мониторы

Процесс `Pid1` может создать **монитор** для процесса `Pid2` используя встроенную функцию:

```
erlang:monitor(process, Pid2)
```

которая возвращает ссылочное значение (`Ссылку`). Если после этого `Pid2` завершит работу с `Причиной` выхода, то следующее сообщение будет отправлено процессу `Pid1`:

```
{'DOWN', Ссылка, process, Pid2, Причина}
```

Если процесс `Pid2` не существует, то сообщение `'DOWN'` будет отправлено немедленно и поле `Причина` будет установлено равным `noproc`. Мониторы однонаправлены, то есть если `Pid1` следит за `Pid2`, то он получит сообщение о смерти `Pid2`, но `Pid2` **не** получит сообщение о смерти `Pid1`. Повторные вызовы функции `erlang:monitor(process, Pid)` создадут несколько независимых мониторов и каждый из них отправит сообщение `'DOWN'`, когда процесс `Pid` завершит работу.

Монитор можно удалить, вызывая функцию `erlang:demonitor(Ссылка)`. Возможно создание мониторов для процессов с зарегистрированными именами, а также запущенных на других узлах.

6.7 Приоритетность процессов

Встроенная функция `process_flag(priority, Приоритет)` определяет приоритет текущего процесса. `Приоритет` может быть одним из следующих значений `normal` (по умолчанию), `low`, `high` или `max`.

Изменение приоритета процесса не рекомендуется и должно производиться только в особых случаях. Проблема, которая требует смены приоритета процесса, чаще всего может быть решена и другим подходом.

6.8 Словарь процесса

Каждый процесс имеет собственный словарь, являющийся списком пар термов в форме `{Ключ, Значение}`.

Встроенные функции для работы со словарём процесса	
`put(Ключ, Значение)`	Сохраняет `Значение` с `Ключом` или заменяет уже существующее
`get(Ключ)`	Извлекает значение, сохранённое с `Ключом`, иначе возвращает `undefined`
`get()`	Возвращает весь словарь процесса целиком, в виде списка пар `{Ключ, Значение}`
`get_keys(Значение)`	Возвращает список ключей, которые имеют `Значение`
`erase(Ключ)`	Удаляет пару `{Ключ, Значение}`, если она была, и возвращает `Ключ`
`erase()`	Возвращает словарь процесса целиком и удаляет его содержимое

Словари процессов могут использоваться для того, чтобы хранить глобальные переменные в приложении, но их слишком активное использование обычно усложняет отладку и считается плохим стилем программирования.

7 Обработка ошибок

Эта глава описывает обработку ошибок внутри процесса. Такие ошибки известны ещё под названием **исключения**.

7.1 Классы исключений и причины ошибок

Классы исключений	
`error`	Ошибка времени выполнения, например, при применении оператора к недопустимым типам аргументов. Ошибки времени выполнения могут быть вызваны программистом с помощью встроенной функции `erlang:error(Причина)` или `erlang:error(Причина, Аргументы)`
`exit`	Процесс вызвал `exit(Причина)`, см. раздел 6.4 о завершении процессов
`throw`	Процесс вызвал `throw(Выражение)`, см. раздел 7.2 о бросках исключений

Появление исключения приводит к аварийной остановке процесса, то есть его исполнение останавливается и он и его данные удаляются из системы. Также это действие называется *уничтожением* (termination). После этого сигналы выхода посылаются всем связанным процессам. Исключение состоит из класса, причины выхода и копии стека вызовов. Стек вызовов можно сформировать и получить в удобном виде с помощью функции `erlang:get_stacktrace/0`.

Ошибки времени исполнения и другие исключения могут не приводить к смерти процесса, если использовать выражения `try` и `catch`.

Для исключений, имеющих класс `error`, например для обычных ошибок времени выполнения, **причиной выхода** будет кортеж `{Причина, Стек}` где `Причина` — это терм, указывающий более точно на тип ошибки.

Причины выхода	
`badarg`	Передан аргумент недопустимого типа.
`badarith`	Аргумент в арифметическом выражении имеет недопустимый тип.
`{badmatch, Значение}`	Вычисление сопоставления с образцом прошло неудачно. `Значение` не совпало с образцом.
`function_clause`	Не найдено ни одного подходящего уравнения функции по образцам аргументов и охранных выражений при вычислении вызова функции.
`{case_clause, Значение}`	При вычислении выражения `case Значение` не совпало ни с чем.
`if_clause`	Ни одна из веток выражения `if` не оказалась равна `true`.
`{try_clause, Value}`	При вычислении секции выражения `try` со `Значением` не совпала ни одна из веток.
`undef`	Функция не была найдена при попытке исполнить вызов функции
`{badfun, Fun}`	Что-то не так с анонимной функцией `Fun`
`{badarity, Fun}`	Функция вызвана с неверным количеством аргументов. Значение `Fun` описывает и её и переданные аргументы
`timeout_value`	Значение таймаута в `receive…after` было вычислено и оказалось не целым 32-битным числом и не атомом `infinity`
`noproc`	Попытка создать связь с несуществующим процессом
`{nocatch, Значение}`	Попытка выполнить `throw` за пределами кода, защищённого оператором `catch`. `Значение` — терм, который был брошен
`system_limit`	Сработало одно из системных ограничений, заданных реализацией виртуальной машины или операционной системы

`Стек` — это цепочка вызовов функций, которые исполнялись в тот момент, когда произошла ошибка, даётся в виде списка кортежей `{Модуль, Имя, Арность}`, где первым идёт самый недавний вызов. Иногда самый последний вызов вместо

Арности будет содержать Аргументы: {Модуль, Имя, Аргументы}.

7.2 Catch и throw

```
catch Выражение
```

Такая запись возвращает результат вычисления Выражения если во время вычисления не возникло исключение. В случае исключения возвращаемое значение будет кортежем с информацией о нём.

```
{'EXIT', {Причина, Стек}}
```

Такое исключение считается *пойманным*. Непойманное исключение приведёт к уничтожению процесса. Если исключение было вызвано вызовом функции exit(Терм) то будет возвращён кортеж с причиной в виде {'EXIT',Терм}. Если исключение возникло при вызове throw(Терм), тогда будет возвращено значение Терма.

```
catch 1+2  ⇒ 3
catch 1+a   ⇒ {'EXIT',{badarith,[...]}}
```

Оператор catch имеет низкий приоритет и выражения, использующие его часто требуют заключения в блок begin..end или круглые скобки.

```
A = (catch 1+2)  ⇒ 3
```

Встроенная функция throw(Выражение) используется для *нелокального* выхода из функций. Её можно вызывать только под защитой catch, что возвратит результат вычисления Выражения.

```
catch begin 1,2,3,throw(four),5,6 end  ⇒ four
```

Если throw/1 вычисляется за пределами оператора catch, то возникнет ошибка времени выполнения nocatch.

Оператор catch не спасёт процесс от завершения по сигналу выхода из другого связанного с ним процесса (если только не был включен режим перехвата сигналов выхода, trap_exit).

7.3 Try

Выражение `try` способно различить различные классы исключений. Следующий пример эмулирует поведение описанного чуть выше `catch Выражение`:

```
try Expr
catch
    throw:Term -> Term;
    exit:Reason -> {'EXIT', Reason};
    error:Reason -> {'EXIT',{Reason, erlang:get_stacktrace()}}
end
```

Полное описание `try` следующее:

```
try Выражение [of
    Образец1 [when ОхранныеВыражения1] -> Тело1;
    ...;
    ОбразецN [when ОхранныеВыраженияN] -> ТелоN]
[catch
    [КлассОшибки1:]ОбразецИсключения1 [when ОхранаИсключения1] ->
        ТелоИсключения1;
    ...;
    [КлассОшибкиN:]ОбразецИсключенияN [when ОхранаИсключенияN] ->
        ТелоИсключенияN]
[after ТелоПосле]
end
```

В наличии должны обязательно быть как минимум одна строка `catch` или одно предложение `after`. Может дополнительно использоваться оператор `of` после `Выражения`, который добавляет вычисление `case` к значению `Выражения`.

`try` возвратит значение `Выражения`, если только не произойдёт исключение во время его вычисления. Тогда исключение *ловится* и ряд `ОбразцовИсключения` с подходящим `КлассомОшибки` сопоставляются один за другим с пойманным исключением. Если `КлассОшибки` не указан, то подразумевается `throw`. Если сопоставление удачно проходит и необязательные выражения `ОхраныИсключения` тоже равны `true`, то вычисляется соответствующее `ТелоИсключения` и его результат становится возвращаемым значением.

Если не найдено подходящего `ОбразцаИсключения` с таким же `КлассомОшибки` и выражениями `ОхраныИсключения`, равными `true`, то исключение передаётся дальше, как если бы начальное `Выражение` не было заключено в `try`. Исключение, происходящее во время вычисления `ТелаИсключения` не будет поймано.

Если ни один из `Образцов` не совпал, то произойдёт ошибка времени исполнения `try_clause`.

Если определено, то `ТелоПосле` всегда вычисляется **после** всех операций в `try..catch`, независимо от возникновения ошибки. Его возвращаемое значение игнорируется и не влияет на значение всего оператора `try` (как если бы `after` не было). `ТелоПосле` вычисляется даже если исключение произошло в `Теле` или `ТелеИсключения`, в этом случае исключение передаётся выше по коду.

Исключение, которое происходит во время вычисления `ТелаПосле` не ловится, то есть если `ТелоПосле` вычислилось по причине исключения в `Выражении`, `Теле` или `ТелеИсключения`, то оригинальное исключение будет потеряно и вместо него полёт вверх по стеку вызовов продолжит уже новое исключение.

8 Распределённый Erlang

Распределённая Erlang-система состоит из некоторого количества систем времени исполнения Erlang, которые сообщаются друг с другом. Каждая такая система называется **узлом** (node). Узлы могут находиться на одной физической машине или на разных и быть соединёнными посредством сети. Стандартный механизм распределения реализован на основе TCP/IP сокетов, но могут быть реализованы и другие механизмы.

Передача сообщений между процессами на разных узлах, так же как и связи между процессами и мониторы, прозрачно реализована используя идентификаторы процессов (pid). Однако, зарегистрированные имена локальны для каждого из узлов. На зарегистрированный процесс на конкретном узле можно ссылаться с помощью кортежа `{Имя,Узел}`.

Служба отображения портов Erlang (Erlang Port Mapper Daemon, или **epmd**) автоматически стартует на каждом компьютере, где имеется запущенный узел Erlang. Он отвечает за отображение имён узлов в сетевые адреса компьютеров.

8.1 Узлы

Узел — это исполняемая в данный момент Erlang-система, которой было назначено имя, используя параметр командной строки `-name` (длинное имя) или `-sname` (короткое имя).

Формат имени узла — атом вида `Имя@Адрес` (помните, `@` является допустимым в атомах символом), где `Имя` задаётся пользователем, запустившим узел, а `Адрес` — полное имя сервера, если были включены длинные имена, или первая часть

имени сервера (если были использованы короткие имена). Функция `node()` возвращает имя узла. Узлы, использующие длинные имена не могут связываться с узлами, использующими короткие имена.

8.2 Соединение между узлами

Узлы распределённой Erlang-системы полностью соединены (каждый с каждым). Первый раз, когда используется новое имя узла, производится попытка подключения к этому узлу. Если узел `A` подключается к узлу `B`, и узел `B` имел открытое подключение к узлу `C`, то узел `A` тоже попытается подключиться к узлу `C`. Эта возможность может быть отключена используя параметр командной строки:

```
-connect_all false
```

Если узел прекращает работу или теряет сеть, все подключения к нему удаляются. Встроенная функция:

```
erlang:disconnect(Узел)
```

отключает заданный `Узел`. Встроенная функция `nodes()` вернёт список подключенных в данный момент (видимых) узлов.

8.3 Скрытые узлы

Иногда полезно подключиться к нужному узлу, не инициируя веер подключений ко всем остальным узлам. Для этой цели можно использовать **скрытый узел**. Скрытый узел это узел, запущенный с параметром командной строки `-hidden`. Подключения между скрытыми узлами и другими узлами должны устанавливаться вручную и явно. Скрытые узлы не видны в списке узлов, возвращаемом функцией `nodes()`. Вместо этого следует использовать `nodes(hidden)` или `nodes(connected)`. Скрытый узел не будет включён в набор узлов, за которыми следит модуль `global`.

Узел на языке C это C-программа, написанная, чтобы действовать и выглядеть, как скрытый узел в распределённой Erlang-системе. Библиотека `erl_interface` содержит необходимые для этого функции.

8.4 Секретный куки (cookie)

Каждый узел имеет свой собственный ключ, ещё называемый **магический куки** (cookie), который является атомом. Сервер сетевой аутентикации Erlang (под названием `auth`) читает содержимое куки из файла `$HOME/.erlang.cookie`. Если файл не существовал, он будет создан и в него будет записана случайная строка.

Права доступа к файлу должны быть установлены в восьмеричное 0400 (только для чтения владельцем). Куки локального узла также можно установить с помощью встроенной функции `erlang:set_cookie(node(), Куки)`.

Текущему узлу позволяется подключаться к другому узлу `Узел2`, если он знает значение его куки. Если оно отличается от куки текущего узла (чей куки будет использован по умолчанию), то его надо явно установить с помощью встроенной функции `erlang:set_cookie(Узел2, Куки2)`.

8.5 Встроенные функции для распределения

Встроенные функции для распределения	
`node()`	Возвращает имя текущего узла. Позволяется использовать в охранных выражениях
`is_alive()`	Возвращает `true` если система является узлом и может подключаться к другим узлам, иначе `false`
`erlang:get_cookie()`	Возвращает магический куки текущего узла
`set_cookie(` `  Узел, Куки)`	Устанавливает магический `Куки`, который будет использован при подключении к `Узлу`. Если `Узел` — текущий узел, то `Куки` будет использован для всех подключений к новым узлам
`nodes()`	Возвращает список всех видимых узлов, к которым подключен текущий
`nodes(connected)` `nodes(hidden)`	Возвращает список не только видимых, но и скрытых и ранее известных узлов, и т.д.
`monitor_node(Узел,` `  true|false)`	Отслеживает статус `Узла`. Сообщение `{nodedown, Узел}` будет прислано процессу, если подключение к узлу потеряно
`node(Pid|Ref|Port)`	Возвращает имя узла, на котором зарегистрирован аргумент
`erlang:disconnect_node` `  (Узел)`	Принудительно отключает `Узел` от кластера

`spawn[_link|_opt](`
`Узел, Модуль,`
`Функция, Аргументы)`
	Создаёт процесс на другом (удалённом) узле

`spawn[_link|_opt](` `Узел, Функция)`	Создаёт процесс на удалённом узле

8.6 Параметры командной строки

Параметры командной строки для распределённого Erlang	
`-connect_all false`	Подключение новых узлов только вручную и явно перечисляется каждый узел
`-hidden`	Стартует узел как скрытый
`-name Имя`	Превращает систему Erlang в узел кластера, используя длинные имена узлов
`-setcookie Куки`	Аналогично вызову `erlang:set_cookie(node(), Куки))`
`-sname Имя`	Превращает систему Erlang в узел кластера, используя короткие имена узлов

8.7 Модули с поддержкой распределённых систем

Есть несколько доступных модулей, которые пригодятся при программировании распределённых систем:

Модули с поддержкой распределённых систем	
`global`	Глобальное средство регистрации имён
`global_group`	Соединение узлов в глобальные группы регистрации имён
`net_adm`	Различные функции для управления сетью в Erlang-системе
`net_kernel`	Ядро работы с сетью
Модули стандартной библиотеки, полезные для разработки распределённых систем	
`slave`	Запуск и управление ведомыми узлами

9 Порты и драйверы портов

Порты предоставляют байто-ориентированный интерфейс к внешним программам и связывается с процессами Erlang посылая и принимая сообщения в виде списков байтов. Процесс Erlang, который создаёт порт, называется **владельцем порта** или **подключенным к порту процессом**. Все коммуникации в и из порта должны пройти через владельца порта. Если владелец порта завершает работу, порт тоже закроется (а также и внешняя программа, подключенная к порту, если она была правильно написана и среагирует на закрытие ввода/вывода).

Внешняя программа является другим процессом операционной системы. По умолчанию, она должна считывать данные из стандартного входа (файловый дескриптор 0) и отвечать на стандартный вывод (файловый дескриптор 1). Внешняя программа должна завершать свою работу когда порт закрывается (ввод/вывод закрывается).

9.1 Драйверы портов

Драйверы портов обычно пишутся на языке C и динамически подключаются к системе исполнения Erlang. Встроенный драйвер ведёт себя как порт и называется **драйвером порта**. Однако, ошибка в драйвере порта может привести к нестабильности во всей системе Erlang, утечкам памяти, зависаниям и краху системы.

9.2 Встроенные функции для портов

Функция для создания порта	
`open_port(ИмяПорта, НастройкиПорта)`	Возвращает **идентификатор порта** `Порт`, как результат открытия нового Erlang-порта. Сообщения могут быть отправлены в и получены через идентификатор порта, так же как это можно делать с идентификаторами процессов. Идентификаторы портов могут быть связаны с процессами, или зарегистрированы под каким-либо именем с помощью `link/1` и `register/2`.

`ИмяПорта` обычно является кортежем вида `{spawn,Команда}`, где строка `Команда` является именем внешней программы. Внешняя программа выполняется за пределами Erlang-системы, если только не найден драйвер порта с именем `Команда`. Если драйвер найден, он будет активирован вместо команды.

`НастройкиПорта` — это список настроек (опций) для порта. Список обычно содержит как минимум один кортеж `{packet,N}`, указывающий, что данные, пересылаемые между портом и внешней программой, предваряются N-байтовым индикатором длины. Разрешённые значения для `N` — 1, 2 или 4. Если двоичные данные должны использоваться вместо списков байтов, то должна быть включена опция `binary`.

Владелец порта Pid связывается с `Портом` с помощью отправки и получения Erlang-сообщений. (Любой процесс может послать сообщение в порт, но ответы от порта всегда будут отправлены только владельцу порта).

Сообщения, отсылаемые в порт	
`{Pid, {command, Данные}}`	Посылает `Данные` в порт.
`{Pid, close}`	Закрывает порт. Если порт был открыт, он отвечает сообщением `{Порт, closed}`, когда все буферы были сброшены и порт закрылся.
`{Pid, {connect, НовыйPid}}`	Устанавливает владельца `Порта` равным `НовомуPid`. Если порт был открыт, он отвечает сообщением `{Порт, connected}` старому владельцу. Заметьте, что старый владелец порта остаётся связанным с портом, тогда как новый — нет.

Данные должны быть списком ввода-вывода. **Список ввода-вывода** (iolist) — это либо двоичные данные, либо смешанный (возможно вложенный) список двоичных данных и целых чисел в диапазоне от 0 до 255.

Сообщения, получаемые из порта	
`{Порт, {data, Данные}}`	Данные получены от внешней программы
`{Порт, closed}`	Ответ на команду `Порт ! {Pid,close}`
`{Порт, connected}`	Ответ на команду `Порт ! {Pid,{connect, NewPid}}`
`{'EXIT', Порт, Причина}`	Присылается, если порт был отключен по какой-либо причине.

Вместо того, чтобы отправлять и получать сообщения, имеется ряд встроенных функций, которые можно использовать. Они могут быть вызваны любым процессом, а не только владельцем порта.

Встроенные функции для работы с портами	
`port_command(Порт, Данные)`	Отправляет `Данные` в `Порт`
`port_close(Порт)`	Закрывает `Порт`
`port_connect(Порт, НовыйPid)`	Устанавливает владельца `Порта` равным `НовомуPid`. Старый владелец остаётся связанным с портом и должен сам вызвать `unlink(Порт)` если связь не требуется.
`erlang:port_info(Порт, Элемент)`	Возвращает информацию о `Порте` с ключом `Элемент`
`erlang:ports()`	Возвращает список всех открытых портов на текущем узле

Есть несколько дополнительных встроенных функций, которые применимы только к драйверам портов: это `port_control/3` и `erlang:port_call/3`.

10 Загрузка кода

Erlang поддерживает обновление кода во время работы без остановки системы. Замена кода выполняется на уровне модулей.

Код модуля может существовать в системе в двух версиях: **текущая** и **старая**. Когда модуль загружается в систему в первый раз, код становится *текущим*. Если загружается новая версия модуля, то код предыдущей версии, уже имеющийся в памяти, становится *старым* и новая загруженная версия становится *текущей*. Обычно модуль автоматически загружается, когда вызвана одна из находящихся в нём функций. Если модуль уже загружен, то он должен быть явно перезагружен в новую версию.

И старый и текущий код полностью функциональны и могут использоваться одновременно. Полностью определённые вызовы функций (с именем модуля) всегда ссылаются на текущую версию кода. Однако старый код может продолжать исполняться другими процессами.

Если загрузить третью версию загруженного модуля, то сервер кода удалит (операция называется *purge*) старый код и все процессы, всё ещё использующие его, будут принудительно завершены. Затем третья версия становится *текущей* и предыдущий текущий код становится *старым*.

Чтобы перейти от старого кода к текущему, процесс должен выполнить один полностью определённый вызов функции (с именем модуля).

```
-module(mod).
-export([loop/0]).

loop() ->
    receive
        code_switch ->
            mod:loop();
        Msg ->
            ...
            loop()
    end.
```

Чтобы заставить процесс (в этом примере) сменить версию кода, отправьте ему сообщение `code_switch`. Процесс после этого выполнит полностью определённый вызов `mod:loop()` и это переключит его на текущую версию кода. Заметьте, что `mod:loop/0` должна быть для этого экспортирована.

11 Макросы

11.1 Определение и использование макросов

```
-define(Константа, Замена).
-define(Функция(Переменная1, ..., ПеременнаяN), Замена).
```

Макрос должен быть определён перед тем, как он используется, но определение макроса можно поместить где угодно среди атрибутов и определений функций в модуле. Если макрос используется в нескольких модулях, рекомендуется поместить его определение во включаемый файл. Макрос используется так:

```
?Константа
?Функция(Переменная1,...,ПеременнаяN)
```

Макросы разворачиваются во время компиляции на самом раннем этапе. Ссылка на макрос `?Константа` будет заменена на текст `Замена` так:

```
-define(TIMEOUT, 200).
...
call(Request) ->
    server:call(refserver, Request, ?TIMEOUT).
```

разворачивается перед компиляцией в:

```
call(Request) ->
    server:call(refserver, Request, 200).
```

Ссылка на макрос `?Функция(Аргумент1, ..., АргументN)` будет заменена на `Замену`, где все вхождения переменной `ПеременнаяX` из определения макроса будут заменены на соответствующий `АргументX`.

```
-define(MACRO1(X, Y), {a, X, b, Y}).
...
bar(X) ->
    ?MACRO1(a, b),
    ?MACRO1(X, 123).
```

будет развёрнуто в:

```
bar(X) ->
    {a, a, b, b},
    {a, X, b, 123}.
```

Для просмотра результата разворачивания макросов, модуль можно скомпилировать с параметром `'P'` таким образом:

```
compile:file(Файл, ['P']).
```

Это производит распечатку разобранного кода после применения к нему предварительной обработки и трансформаций разбора (parse transform), в файле с именем `Файл.P`.

11.2 Предопределённые макросы

Предопределённые макросы	
`?MODULE`	Атом, имя текущего модуля
`?MODULE_STRING`	Строка, имя текущего модуля
`?FILE`	Имя исходного файла текущего модуля
`?LINE`	Текущий номер строки
`?MACHINE`	Имя виртуальной машины, 'BEAM'

11.3 Управление исполнением макросов

```
-undef(Macro).      % Это отменяет определённый ранее макрос

-ifdef(Macro).
    %% Строки, которые будут скомпилированы, если Macro существует
-else.
    %% Иначе будут скомпилированы эти строки
-endif.
```

`ifndef(Macro)` можно использовать вместо `ifdef` и имеет обратный смысл.

```
-ifdef(debug).
-define(LOG(X), io:format("{~p,~p}:~p~n",[?MODULE,?LINE,X])).
-else.
-define(LOG(X), true).
-endif.
```

Если макрос `debug` определён в то время, когда идёт компиляция модуля, то макрос `?LOG(Arg)` развернётся в вызов печати текста `io:format/2` и обеспечит пользователя отладочным выводом в консоль.

11.4 Превращение аргументов макроса в строку

Запись вида `??Аргумент`, где `Аргумент` — это параметр, передаваемый в макрос, развернётся в представление аргумента в строковом виде.

```
-define(TESTCALL(Call), io:format("Call ~s: ~w~n", [??Call, Call])).

?TESTCALL(myfunction(1,2)),
?TESTCALL(you:function(2,1)),
```

Разворачивается в:

```
io:format("Call ~s: ~w~n",
          ["myfunction(1,2)", m:myfunction(1,2)]),
io:format("Call ~s: ~w~n",
          ["you:function(2,1)", you:function(2,1)]),
```

Таким образом, получается отладочный вывод как вызванной функции так и результата.

12 Дальнейшие материалы для чтения

Следующие вебсайты предлагают более подробное объяснение тем и концепций, кратко описанных в данном документе:

12.1 Русскоязычные ресурсы

- Русские новости из мира Erlang http://erlanger.ru
- Группа `erlang-russian` и `erlang-in-ukraine` на сервере Google Groups.
- Книга Ф. Чезарини «Программирование в Erlang», русский перевод http://dmkpress.com/catalog/computer/programming/functional/978-5-94074-617-1/

Также обратите внимание на русскоязычный канал `erlang` на сервере http://jabber.ru

12.2 Англоязычные ресурсы

- Официальная документация по Erlang: http://www.erlang.org/doc/
- Learn You Some Erlang for Great Good: http://learnyousomeerlang.com/
- Раздел с уроками на Erlang Central: https://erlangcentral.org/wiki/index.php?title=Category:HowTo

Ещё вопросы? Список рассылки `erlang-questions` (адрес для подписки http://erlang.org/mailman/listinfo/erlang-questions) является хорошим местом

для неспешных общих дискуссий об Erlang/OTP, языке, реализации, использовании и вопросы новичков. Если вы не планируете писать в рассылку, её можно прочесть без подписки на сервере Google Groups, группа `erlang-programming`.

Также обратите внимание на англоязычный IRC канал `#erlang` в сети Freenode.