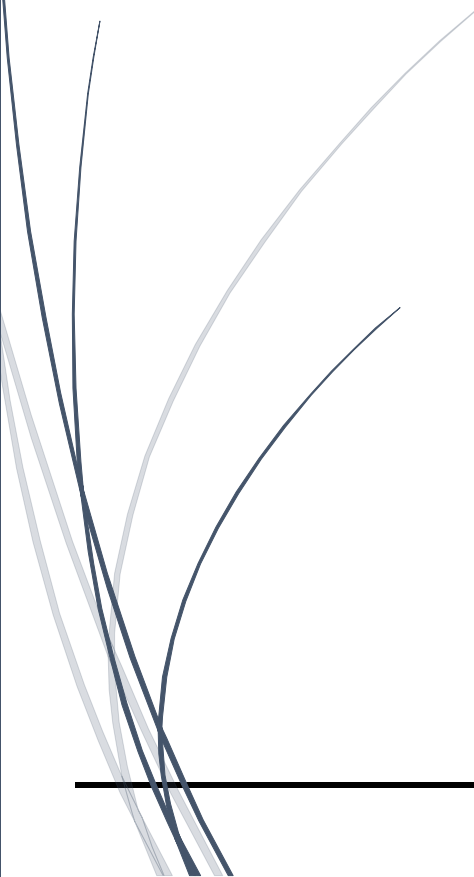
A dark blue vertical bar is on the left. A blue arrow points right from it, containing the date.

10/19/2017

ERLANG

programming language

Tìm hiểu ngôn ngữ lập trình

Several thin, curved lines in shades of blue and grey originate from the left side and curve upwards and to the right.

17CTT1TN - Faculty of Information Technology
HO CHI MINH UNIVERSITY OF SCIENCE

LỊCH SỬ PHIÊN BẢN

Phiên bản	Thực hiện bởi	Ngày sửa	Xác nhận bởi	Ngày xác nhận	Lý do
1.0	Nhóm: Cao Nhon Hưng Phan Đăng Hoài Bảo Lê Trần Hữu Đắc	10/19/2017	Võ Hoài Việt	<mm/dd/2017>	Bản thảo đầu tiên

NỘI DUNG

1. TỔNG QUAN	5
1.1 Lịch sử hình thành và phát triển ngôn ngữ erlang	5
1.2 Đặc tính của ngôn ngữ erlang	7
2. CÀI ĐẶT MÔI TRƯỜNG LẬP TRÌNH	7
3. CÚ PHÁP CƠ BẢN CỦA CHƯƠNG TRÌNH	8
3.1 Modules	8
3.2 Export	8
3.3 Nhập xuất	8
3.4 Chương trình cơ bản	8
4. BIẾN VÀ KIỂU DỮ LIỆU	9
4.1 Biến trong erlang	9
4.2 Kiểu dữ liệu trong erlang	11
4.2.1 ATOM	11
4.2.2 BOOLEAN	11
4.2.3 BIT STRING	11
4.2.4 TUPLE	11
4.2.5 LIST	12
5. TOÁN TỬ	13
5.1 Toán tử số học	13
5.2 Toán tử so sánh	14
5.3 Toán tử trên bit	16
5.4 Phép gán	17
6. SỬA LỖI CHƯƠNG TRÌNH	17

6.1	Debug	17
6.2	Một số lỗi cơ bản	17
6.2.1	Compile-time errors	17
6.2.2	Logical errors	18
6.2.3	Run-time errors	19
7.	CÁC CẤU TRÚC ĐIỀU KHIỂN VÀ VÒNG LẶP	22
7.1	Cấu trúc rẽ nhánh có điều kiện	22
7.2	Cấu trúc vòng lặp	24
8.	LẬP TRÌNH HÀM.....	28
8.1	Lập trình hàm.....	28
8.2	Đệ quy.....	29
9.	CÁC CẤU TRÚC DỮ LIỆU ĐẶC THÙ VÀ THAO TÁC.....	34
9.1	Mảng.....	34
9.2	Chuỗi.....	39
9.3	Dictionary.....	45
10.	TRỪU TƯỢNG HÓA DỮ LIỆU.....	47
11.	TẬP TIN - THAO TÁC VỚI FILE TRONG ERLANG.....	50
11.1.1	Đọc nội dung file 1 dòng 1 lần đọc	50
11.1.2	Đọc toàn bộ nội dung file 1 lần	51
11.1.3	Viết nội dung ra 1 file	51
11.1.4	Sao chép 1 file đã tồn tại	52
11.1.5	Xóa 1 file.....	53
11.1.6	Liệt kê nội dung 1 thư mục.....	53
11.1.7	Tạo thư mục mới	54
11.1.8	Đổi tên 1 file đã tồn tại.....	54

11.1.9	Xác định kích thước của file.....	55
12.	CHUẨN VIẾT CHƯƠNG TRÌNH.....	55
12.1	Cấu trúc tập tin	56
12.2	Sự thụt đầu dòng	58
12.3	Đặt tên	59
12.4	Kích thước của những tập hợp.....	59
13.	MỘT SỐ CHỦ ĐỀ NÂNG CAO	59
13.1	Web programing.....	59
13.2	Driver	62
14.	TỔNG KẾT	65
PHỤ LỤC A: TÀI LIỆU THAM KHẢO		67
PHỤ LỤC B: THUẬT NGỮ		68

1. TỔNG QUAN

1.1 LỊCH SỬ HÌNH THÀNH VÀ PHÁT TRIỂN NGÔN NGỮ ERLANG

Erlang được phát triển tại phòng thí nghiệm Ericsson Computer Science Laboratory vào những năm 1986. Erlang được thiết kế cho việc lập trình tương tranh, một chương trình trong Erlang được tạo thành bởi hàng nghìn, hàng vạn các tiến trình, các tiến trình này không chia sẻ bộ nhớ và giao tiếp với nhau thông qua các message. Erlang có cơ chế cho phép chương trình thay đổi code ngay khi chúng đang được thực thi (cơ chế “on the fly”), chính nhờ cơ chế này ta có thể xây dựng phần mềm cho các hệ thống non-stop.

Giai đoạn 1985-1988: Sự ra đời của Erlang

Vào những năm 1986, Joe Armstrong dự định phát triển một trình biên dịch cho phép lập trình các ứng dụng trên điện thoại một cách hiệu quả. Như chúng ta đã biết, các ứng dụng trên điện thoại có yêu cầu về khả năng tương tranh cao: một thao tác switch phải xử lý hàng trăm thậm chí hàng ngàn nghìn các giao dịch. Trong quá trình phát triển, ông bị ảnh hưởng khá nhiều bởi cấu trúc cú pháp của ngôn ngữ Prolog. Trình biên dịch của ông không những hỗ trợ được cho nhiều process cùng lúc mà còn có cơ chế trao đổi thông điệp giữa các process, cơ chế bất lỗi... Cùng với Robert Virding, ông đã nghiên cứu và phát triển ra 1 ngôn ngữ lập trình, đồng thời phát triển các nguyên lý cho ngôn ngữ này, ngày nay chúng ta gọi ngôn ngữ này là Erlang và nguyên lý mà 2 người phát triển là “Concurrency-Oriented Programming”. Như vậy, Erlang là 1 ngôn ngữ lai giữa ngôn ngữ lập trình tương tranh và ngôn ngữ lập trình hàm.

Vào những năm 1987, Erlang lần đầu tiên được sử dụng để thực hiện các ứng dụng thực tế, Erlang được sử dụng để tạo ra một kiến trúc phần mềm mới mang tên ACS3, thiết kế cho lập trình các dịch vụ điện thoại trên Ericsson MD110 PABX4, và dự án mang tên ACS/Dunder đã được triển khai, dựa trên ngôn ngữ Erlang để xây dựng kiến trúc ACS3. Chính nhờ dự án này, Erlang được phát triển một cách nhanh chóng, liên tục được cập nhật các tính năng mới, và được áp dụng ngay vào thực tế, những tính năng phù hợp sẽ được giữ lại, còn không sẽ được bỏ đi. Hầu hết các thay đổi của Erlang đều không được lưu lại. Dưới đây là những câu lệnh hiếm hoi của Erlang trong những ngày đầu phát triển.

erlang vsn 1.05	
h	help
⊗ reset	reset all queues
reset_erlang	kill all erlang definitions
load(F)	load erlang file <F>.erlang
load	load the same file as before
load(?)	what is the current load file
what_erlang	list all loaded erlang files
go	reduce the main queue to zero
send(A,B,C)	perform a send to the main queue
send(A,B)	perform a send to the main queue
cq	see queue - print main queue
wait_queue(N)	print wait_queue(N)
cf	see frozen - print all frozen states
eqns	see all equations
eqn(N)	see equation(N)
start(Mod,Goal)	starts Goal in Mod
top	top loop run system
q	quit top loop
open_dots(Node)	opens Node
talk(N)	N=1 verbose, =0 silent
peep(M)	set peeping point on M
no_peep(M)	unset peeping point on M
vsn(X)	erlang vsn number is X

Đến cuối năm 1988, hầu hết mọi ý tưởng trong Erlang đã định hình.

Giai đoạn 1989-1997:

Trong khoảng thời gian 8 năm này chính là giai đoạn phát triển chính của Erlang, từ thời gian ban đầu chỉ có 2 người phát triển, đến giai đoạn này đã có hàng trăm người nghiên cứu và sử dụng Erlang.

Kết quả của dự án ACS/Dunder: vào tháng 12 năm 1989, báo cáo cuối cùng của dự án được đưa ra, có khoảng 25 tính năng của điện thoại đã thực hiện thành công, các tính năng này đại diện cho khoảng 1/10 chức năng của MD110. Báo cáo cũng chỉ ra rằng, Erlang quá chậm để phát triển sản phẩm. Để có thể tạo ra các sản phẩm thực tế thì Erlang cần phải nhanh hơn 40 lần so với hiện tại, tuy nhiên Ericsson vẫn quyết định xây dựng một sản phẩm gọi là “Mobility server” dựa trên kiến trúc ACS/Dunder.

Mở rộng ra thế giới: năm 1989 cũng là lần đầu tiên Erlang có cơ hội được giới thiệu ra ngoài thế giới thông qua hội nghị SETSS tại Bournemouth. Chính nhờ hội nghị này mà các tác giả đã được mời đến Bellcore để nói về Erlang.

1.2 ĐẶC TÍNH CỦA NGÔN NGỮ ERLANG

Erlang đơn giản chỉ là một ngôn ngữ lập trình như bao ngôn ngữ khác. Điểm khác biệt duy nhất là nó được thiết kế với mục tiêu chạy song song để giúp tận dụng tối đa ưu thế của phần cứng máy tính. Điều này có nghĩa rằng, nếu máy tính của bạn có càng nhiều nhân (core) thì đồng nghĩa với việc chương trình của bạn sẽ chạy càng nhanh. Chính điều đó khiến cho nó trở nên mạnh mẽ hơn các ứng dụng được viết bởi những ngôn ngữ khác không hỗ trợ chạy song song.

Khả năng xây dựng ứng dụng chịu lỗi tốt, chúng có thể được chỉnh sửa mà không cần phải dừng ứng dụng lại.

Ngôn ngữ sử dụng mô hình lập trình hàm (functional programming).

Đây là một ngôn ngữ được kiểm nghiệm thực tế trong hệ thống sản xuất công nghiệp quy mô lớn, có các thư viện khổng lồ.

Các ứng dụng được viết với lượng dòng lên không quá nhiều.

2. CÀI ĐẶT MÔI TRƯỜNG LẬP TRÌNH

Tất cả những gì ta cần là cài đặt một Text editor và môi trường Erlang.

- Text editor: *Sublime text*, *Atom*, *Vim*, ... cài đặt tùy sở thích cá nhân.
- Erlang environment: Trong Window ta chỉ cần truy cập trang chủ của Erlang: <https://www.erlang.org>, vào mục DOWNLOADS và tải Binary file phù hợp với hệ điều hành về cài đặt.

Hoặc ta có thể cài đặt plugin Erlide (<http://erlide.org/>) vào Eclipse để sử dụng như một IDE cho việc lập trình Erlang.

Ngoài ra chúng ta cần lưu ý đến CEAN (The Comprehense Erlang Archive Network) – đây là sự tổng hợp tất cả các *ứng dụng* của Erlang vào một bộ cài đặt duy nhất, điều này giúp cho ta có thể khai thác được một lượng lớn các chương trình viết bằng Erlang. Để tải và sử dụng CEAN, truy cập link: <http://cean.process-one.net/downloads/>.

3. CÚ PHÁP CƠ BẢN CỦA CHƯƠNG TRÌNH

3.1 MODULES

Erlang code được chia thành các modules. Module là một cụm các functions được nhóm vào trong 1 file. Các hàm trong Erlang phải được defined trong 1 module nào đó. Có thể khai báo module theo cú pháp `-module(Name)`. và nó phải được khai báo đầu tiên trong file.

3.2 EXPORT

Trong chương trình ta cần phải cài đặt các hàm. Để định nghĩa các hàm trong module ta dùng cú pháp `-export([Function1/Arity, Function2/Arity, ..., FunctionN/Arity])`. nó định nghĩa danh sách các hàm cùng với từng arity riêng biệt. Arity của một hàm là số biến được truyền vào.

3.3 NHẬP XUẤT

Ta có thể dùng Erlang shell để gọi các hàm trong module và truyền dữ liệu vào hàm để xử lí.

Ta có hàm ***io:format*** dùng để xuất dữ liệu.

VD: `io:format("Hello~n")`. sẽ in ra dòng chữ Hello.

3.4 CHƯƠNG TRÌNH CƠ BẢN

`-module(cuphapcoban).`

`-export([xyz/0, add/2, add/3]).`

`xyz() ->`

```
io:format("Hello Erlang!\n").

% Cộng 2 số
add(A, B) ->
    S = A + B,
    io:format("~p + ~p = ~p\n", [A, B, S]).

% Cộng 3 số
add(A, B, C) ->
    S = A + B + C,
    io:format("~p + ~p + ~p = ~p\n", [A, B, C, S]).
```

4. BIẾN VÀ KIỂU DỮ LIỆU

4.1 BIẾN TRONG ERLANG

Trong erlang, biến không thể thay đổi giá trị khi lập trình. Tên biến phải bắt đầu bằng chữ in hoa. Ta xem các ví dụ dưới để hiểu rõ hơn:

```

Erlang/OTP 20 [erts-9.1] [64-bit] [smp:4:4] [ds:4:4:10] [async-threads:10]
Eshell V9.1 (abort with ^G)
1> One.
* 1: variable 'One' is unbound
2> One = 1.
1
3> Un = Uno = One = 1.
1
4> Two = One + One.
2
5> Two = 2.
2
6> Two = Two + 1.
** exception error: no match of right hand side value 3
7> two = 2.
** exception error: no match of right hand side value 2
8> █

```

Chúng ta có thể khai báo giá trị cho biến duy nhất 1 lần; sau đó ta có thể thử gán giá trị cho biến, nếu giá trị khác nhau, Erlang sẽ báo lỗi. Toán tử “=” có vai trò so sánh 2 giá trị và báo lại nếu khác nhau, nếu bằng nhau nó sẽ trả về giá trị đó:

```

8> 47 = 45 + 2.
47
9> 47 = 45 + 3.
** exception error: no match of right hand side value 48

```

Nếu vế trái là 1 biến và chưa có giá trị thì Erlang sẽ tự động gán giá trị của vế phải cho biến và sự so sánh vẫn tiếp tục được thực hiện và biến sẽ ghi nhớ giá trị đó.

Tên biến có thể bắt đầu bằng ký tự gạch dưới (“_”), nhưng theo quy ước chúng được dùng để lưu những giá trị mà ta không quan tâm.

Ta cũng có thể có 1 biến với tên chỉ là 1 dấu gạch dưới:

```

10> _ = 14 + 3.
17
11> _
* 1: variable '_' is unbound

```

Không như tất cả những biến khác, nó không hề lưu trữ bất kì giá trị nào.

4.2 KIỂU DỮ LIỆU TRONG ERLANG

4.2.1 ATOM

Atom có thể được xem như các hằng. Atom thường bắt đầu bằng chữ cái thường và chứa các ký tự như chữ, số, “_”, “@”. Atom có thể được đặt trong ngoặc đơn nếu bao gồm những ký tự khác.

```
1> atom.  
atom  
2> atoms_rule.  
atoms_rule  
3> atoms_rule@erlang.  
atoms_rule@erlang  
4> 'Atoms can be cheated!'.  
'Atoms can be cheated!'  
5> atom = 'atom'.  
atom
```

4.2.2 BOOLEAN

Boolean chính là 2 atoms đặt biệt: true và false.

```
6> 2 > 3.  
false  
7> 2 =< 3.  
true
```

4.2.3 BIT STRING

Bit string được dùng để lưu các dữ liệu không định kiểu.

```
13> Bin1 = <<10, 20>>.  
<<10,20>>  
14> X = binary_to_list(Bin1).  
[10,20]
```

4.2.4 TUPLE

Tuple là 1 cách để tổ chức dữ liệu. Nó dùng để nhóm các terms lại với nhau khi biết trước số lượng.

```
19> X = 10, Y = 4.  
4  
20> Point = {X, Y}.  
{10,4}
```

Chúng ta có thể gán dữ liệu bằng tuple và lấy nó.

```
23> Point = {4, 5}.
{4,5}
24> {X, Y} = Point.
{4,5}
25> X.
4
26> {X, _} = Point.
{4,5}_
```

Các phần tử của tuple có thể là bất kì kiểu dữ liệu nào, kể cả 1 tuple khác.

```
27> {point, {X, Y}}.
{point,{4,5}}
```

4.2.5 LIST

List là 1 cấu trúc dùng để lưu dữ liệu gồm nhiều phần tử, các phần tử có thể mang bất kì kiểu dữ liệu nào.

```
1> [1, 2, 3, {numbers, [4, 5, 6]}, 5.34, atom].
[1,2,3,{numbers,[4,5,6]},5.34,atom]
```

Ngoài ra, ta còn có thể thêm vào hoặc xóa bớt phần tử trong list bằng các toán tử “++” và “--”.

```
4> [1, 2, 3] ++ [4, 5].
[1,2,3,4,5]
5> [1, 2, 3, 4, 5] -- [1, 2, 3].
[4,5]
6> [2, 4, 2] -- [2, 4].
[2]
7> [2, 4, 2] -- [2, 4, 2].
[] _
```

Cả 2 toán tử trên đều được thực hiện từ phải sang trái:

```
8> [1, 2, 3] -- [1, 2] -- [3].
[3]
9> [1, 2, 3] -- [1, 2] -- [2].
[2,3]
```

Phần tử đầu tiên của list được gọi là Head, phần còn lại được gọi là Tail.

```

10> hd([1, 2, 3, 4]).
1
11> tl([1, 2, 3, 4]).
[2,3,4]
12> List = [2, 3, 4].
[2,3,4]
13> NewList = [1|List].
[1,2,3,4]
14> [Head|Tail] = NewList.
[1,2,3,4]
15> Head.
1
16> Tail.
[2,3,4]
17> [NewHead|NewTail] = Tail.
[2,3,4]
18> NewHead.
2
19> [1 | []].
[1]
20> [2 | [1 | []]].
[2,1]
21> [3 | [2 | [1 | []]]].
[3,2,1]

```

5. TOÁN TỬ

5.1 TOÁN TỬ SỐ HỌC

<i>Erlang</i>	<i>C</i>
+	+
-	-
*	*
/	/
div	/
rem	%

```

1> 32 + 37.
69
2> 32 * 3.
96
3> 1586 - 1517.
69
4> 576 / 6.
96.0
5> 7 div 3.
2
6> 7 rem 3.
1

```

Erlang không quan tâm bạn nhập số thực hay số nguyên: Cả 2 đều được hỗ trợ khi xử lý số học. Tuy nhiên, nếu bạn muốn chia số nguyên nhận được số nguyên, dùng div, và có số dư, dùng rem (remainder).

Ta cũng có thể dùng nhiều toán tử trong 1 biểu thức như trong các ngôn ngữ lập trình khác, và các phép tính tuân theo quy luật ưu tiên bình thường.

```

7> (50 * 100) - 4931.
69
8> -(50 * 100 - 4931).
-69
9> -50 * (100 - 4931).
241550

```

Để viết số nguyên ở hệ cơ số khác hệ cơ số 10, nhập dưới dạng: Base#value

(Base là hệ cơ số từ 2 -> 36):

```

10> 2#1000101.
69
11> 8#105.
69
12> 16#45.
69

```

5.2 TOÁN TỬ SO SÁNH

<i>Erlang</i>	<i>C</i>
,	&&
;	

xor	^
not	!
==, :=	==
!=, /=	!=
>	>
<	<
>=	>=
<=	<=

Như những ngôn ngữ khác, Erlang có phép toán logic và so sánh.

Phép toán logic:

```
1> true or false.  
true  
2> false and true.  
false  
3> true xor false.  
true  
4> not true.  
false  
5> not (true or false).  
false
```

Phép so sánh:

```
6> 69 == 69.  
true  
7> 69 == 96.  
false  
8> 69 /= 96.  
true  
9> 69 == 69.0.  
false  
10> 69 == 69.0.  
true  
11> 69 /= 69.0.  
false
```


3 dòng cuối (9 ->11) đưa ra những trường hợp đặc biệt: Erlang không quan tâm số thực và số nguyên trong tính toán, nhưng sẽ quan tâm khi so sánh. Do đó trong trường hợp so sánh số nguyên và số thực, dùng `==` thay vì `:=`, `/=` thay vì `=/=`.

Những toán tử so sánh khác là `<`, `>`, `>=`, `<=`. Toán tử cuối cùng viết ngược lại với hầu hết các ngôn ngữ lập trình, nên chú ý vì nó có thể trở thành nguồn gốc của những lỗi cú pháp trong code của bạn.

```
12> 69 < 96.
true
13> 69 < 69.
false
14> 69 >= 69.
true
15> 69 =< 69.
true
_
```

5.3 TOÁN TỬ TRÊN BIT

<i>Erlang</i>	<i>C</i>
<code>band</code>	<code>&</code>
<code>bor</code>	<code> </code>
<code>bxor</code>	<code>^</code>
<code>bnot</code>	<code>!</code>
<code>bsl</code>	<code><<</code>
<code>bsr</code>	<code>>></code>

```
1> 2#10001 bor 2#00101.
21
2> 2#10001010 bsr 1.
69
3> 2#110000 bsl 1.
96
_
```

5.4 PHÉP GÁN

Sử dụng dấu '=' :v

Ví dụ: Audi = 1.

6. SỬA LỖI CHƯƠNG TRÌNH

6.1 DEBUG

Erlang có nhiều cách để giúp tìm lỗi, bao gồm những test frameworks, TypER and Dialyzer (để hiểu rõ hơn: <http://learnyousomeerlang.com/types-or-lack-thereof#for-type-junkies>), debugger

(các bước debug: http://erlang.org/doc/apps/debugger/debugger_chapter.html),

và tracing module.

6.2 MỘT SỐ LỖI CƠ BẢN

6.2.1 Compile-time errors

Thường là lỗi cú pháp: kiểm tra tên hàm, dấu trong ngôn ngữ (ngoặc vuông, ngoặc đơn, chấm, phẩy), arity của function,... Đây là danh sách 1 vài compile-time errors và cách sửa lỗi nếu bạn gặp phải:

- module.beam: Module name 'madule' does not match file name 'module'

Tên module bạn nhập trong -module không trùng với filename.

- ./module.erl:2: Warning: function some_function/0 is unused

Chưa xuất ra một hàm, hay vị trí hàm được dùng bị sai tên hoặc arity. Cũng có thể bạn viết 1 hàm không còn cần thiết nữa.

- ./module.erl:2: function some_function/1 undefined

Hàm không tồn tại. Bạn viết sai tên hoặc arity trong -export hay khi gọi hàm. Lỗi này cũng xuất hiện khi không thể dịch hàm, thường là lỗi cú pháp như quên kết thúc hàm với dấu chấm.

- ./module.erl:5: syntax error before: 'SomeCharacterOrWord'

Lỗi này xảy bởi nhiều nguyên nhân, cụ thể như chưa đóng ngoặc đơn, tuples hay kết thúc dòng lệnh sai (như kết thúc nhánh cuối cùng của case bằng dấu phẩy).

- `./module.erl:5: syntax error before:`

Lỗi này thường xuất hiện khi bạn kết thúc dòng lệnh sai.

- `./module.erl:5: Warning: this expression will fail with a 'badarith' exception`

Lệnh tính toán sai (Vd: `5 + ba.`)

- `./module.erl:5: Warning: variable 'Var' is unused`

Bạn khai báo biến và sau đó không dùng nó. Điều này có thể là bug với code của bạn, nên kiểm tra lại những gì đã viết. Bạn có thể chuyển tên biến thành `'_'` hay gạch dưới trước tên biến đó nếu cảm thấy tên giúp ta có thể đọc được code.

- `./module.erl:5: Warning: a term is constructed, but never used`

Trong 1 trong những hàm, bạn đang làm gì đó như xây dựng 1 list, khai báo 1 tuple hay 1 hàm ẩn mà không ràng buộc với 1 biến hay trả về giá trị. Lỗi này cho biết bạn đang làm 1 việc vô nghĩa hay đã làm sai.

- `./module.erl:5: Warning: this clause cannot match because a previous clause at line 4 always matches`

1 hàm có 1 lệnh đặc biệt sau 1 điều kiện luôn đúng. Vì thế, trình biên dịch có thể cảnh báo bạn rằng bạn thậm chí không bao giờ cần các nhánh rẽ khác.

- `./module.erl:9: variable 'A' unsafe in 'case' (line 5)`

Bạn đang dùng 1 biến được khai báo ở 1 trong những nhánh của 1 case ... of ngoài nó. Điều này được xem là không an toàn. Nếu muốn dùng biến như thế, tốt hơn nên làm như sau:
`MyVar = case ... of ...`

6.2.2 Logical errors

Lỗi logic là những lỗi khó tìm và debug nhất. Chúng hầu như là lỗi của lập trình viên : các nhánh của câu lệnh điều kiện như if và case không xét hết trường hợp, nhầm lẫn phép nhân với phép chia,... Chúng không làm cho chương trình bị lỗi nhưng sẽ đưa ra dữ liệu sai hay khiến chương trình không thi hành như mong muốn.

Có vẻ như bạn phải tự tìm là sửa lỗi khi gặp phải lỗi này, nhưng như đã nói ở 6.1, Erlang có nhiều cách giúp tìm lỗi.

6.2.3 Run-time errors

function_clause

```
1> lists:sort([3,2,1]).
[1,2,3]
2> lists:sort(ffffffff).
** exception error: no function clause matching lists:sort(ffffffff) (lists.erl,
line 480)
```

Tất cả các guard clauses (câu lệnh điều kiện trên cùng của hàm trả về giá trị cho hàm ngay khi nó đúng) của hàm sai, hoặc không có mẫu lệnh của hàm trùng khớp.

case_clause

```
3> case "Unexpected Value" of
3>   expected_value -> ok;
3>   other_expected_value -> 'also ok'
3> end.
** exception error: no case clause matching "Unexpected Value"
```

Có vẻ như ai đó đã quên specific pattern trong case, gửi sai loại dữ liệu, hay cần 1 mệnh đề luôn đúng.

if_clause

```
4> if 2 > 4 -> ok;
4>    0 > 1 -> ok
4> end.
** exception error: no true branch found when evaluating an if expression
```

Giống với lỗi case_clause: không thể tìm nhánh rẽ có chân trị đúng. Đảm bảo rằng bạn đã xét đủ trường hợp hoặc thêm trường luôn đúng là điều bạn cần.

badmatch

```
5> [X,Y] = {4,5}.
** exception error: no match of right hand side value {4,5}
```

Lỗi xảy ra khi pattern matching hỏng. Có thể bạn đang cố làm pattern matches sai (như ví dụ trên), gán biến lần 2, hay bất cứ cái gì 2 vế ở 2 bên dấu '=' không tương đồng.

Chú ý rằng lỗi này thường xảy ra vì lập trình viên cho rằng 1 biến ở dạng `_MyVar` giống với `'_'`. Những biến với gạch dưới ở đầu là biến bình thường, chỉ là trình biên dịch sẽ không báo lỗi nếu bạn không dùng nó. Bạn không thể gán nó quá 1 lần.

badarg

```
6> erlang:binary_to_list("heh, already a list").  
** exception error: bad argument  
   in function  binary_to_list/1  
   called as binary_to_list("heh, already a list")
```

Lỗi này rất giống với `function_clause` vì nó cũng là gọi hàm sai tham số. Khác biệt chủ yếu ở đây là lỗi này thường được gây ra bởi lập trình viên sau khi làm cho biến có hiệu lực từ trong hàm, ngoài guard clauses.

undef

```
7> lists:random([1,2,3]).  
** _exception error: undefined function lists:random/1
```

Lỗi xảy ra khi bạn gọi 1 hàm không tồn tại. Hãy chắc chắn rằng hàm được xuất ra từ module với arity đúng (nếu bạn gọi nó ngoài module) và kiểm tra lại xem bạn đã viết đúng tên hàm và module chưa. 1 nguyên nhân khác của lỗi này là khi module không ở trong đường dẫn tìm kiếm của Erlang. Mặc định, đường dẫn tìm kiếm của Erlang được đặt ở thư mục hiện tại. Bạn có thể thay đổi đường dẫn bằng cách dùng [code:add_patha/1](#) hoặc [code:add_pathz/1](#).

badarith

```
8> 5 + ba.  
** exception error: an error occurred when evaluating an arithmetic expression  
   in operator  +/2  
   called as 5 + ba
```

Lỗi này xảy ra khi bạn thực hiện tính toán không tồn tại, như chia không hay giữa atoms và số.

badfun

```

hhfuns.erl
1 -module(hhfuns).
2 -compile(export_all).
3
4 one() -> 1.
5 two() -> 2.
6
7 add(X, Y) -> X() + Y().

```

```

14> c("hhfuns").
hhfuns.erl:2: Warning: export_all flag enabled - all functions will be exported
{ok, hhfuns}
15> hhfuns:add(one, two).
** exception error: bad function one
    in function  hhfuns:add/2 (hhfuns.erl, line 7)
16> hhfuns:add(1, 2).
** exception error: bad function 1
    in function  hhfuns:add/2 (hhfuns.erl, line 7)
17> hhfuns:add(fun hhfuns:one/0, fun hhfuns:two/0).
3
_

```

Lý do thường xuyên nhất khiến lỗi này xảy ra là khi bạn dùng biến là hàm, nhưng giá trị của biến không phải là 1 hàm. Trong ví dụ trên, dùng hàm hhfuns và 2 atoms là hàm. Không thể thực thi và lỗi xuất hiện. Cách dùng đúng ở lệnh 17.

badarity

```

10> F = fun(_) -> ok end.
#Fun<erl_eval.6.99386804>
11> F(a,b).
** exception error: interpreted function with arity 1 called with two arguments

```

Lỗi này là 1 trường hợp đặc biệt của badfun: Lỗi này xảy ra khi bạn dùng hàm có bậc cao hơn, bạn truyền cho hàm nhiều (hay ít) tham số hơn số lượng chúng có thể xử lý

7. CÁC CẤU TRÚC ĐIỀU KHIỂN VÀ VÒNG LẶP

7.1 CẤU TRÚC RỄ NHÁNH CÓ ĐIỀU KIỆN

Trong ngôn ngữ lập trình C/C++, cấu trúc rẽ nhánh có điều kiện thường có các dạng:

```
If (<biểu thức luận lý>
{
    <Khối các câu lệnh>;
}
else
{
    <Khối các câu lệnh>;
}
Switch (<biểu thức>)
{
    case <giá trị 1>: <lệnh1>;
                    break;
    case <giá trị 2>: <lệnh2>;
                    break;
    .....;
    default : <lệnh>;
}
```

Nhưng trong ngôn ngữ **Erlang**, cấu trúc như If khá khác lạ. Để thấy được điều này, ta cùng xét một ví dụ:

```
-module(what_the_if).
-export([eat_food/1]).

eat_food(N) ->
if N == 2 -> "might full"
```

```
; N > 2 -> "full"
; true  -> "still hungry"
end.
```

Thực hiện chương trình trên:

```
1> c(what_the_if).
2> what_the_if: eat_food(2).
"might full"
3> what_the_if: eat_food(3).
"full"
4> what_the_if: eat_food(1).
"still hungry"
```

Như đã thấy, với $N = 2$ thì chương trình in ra dòng "might full", $N > 2$: in "full", và khi N khác 2 trường hợp trên thì in "still hungry".

Vậy ta có thể nhận ra, câu lệnh **if else** trong **Erlang** được viết thành **if true**. Và 1 điểm đặc biệt đó là thay vì trong C++ ta viết nhiều dòng **if**, **else** để xét với mỗi điều kiện:

```
if (N == 2) cout<<"might full";
else if (N > 2) cout<<"full";
else cout<<"still hungry";
```

Trong **Erlang** ta chỉ cần viết 1 lần **if** sau đó liệt kê các điều kiện và lệnh cần thực hiện ở từng dòng, khi đó dấu ";" đóng vai trò như **else if** hay **else** :

```
if N == 2 -> "might full"
; N > 2 -> "full"
; true  -> "still hungry"
end.
```

Ta có cú pháp **if** tổng quát:


```
if
    <biểu thức luận lý 1> -> <lệnh 1>
    <biểu thức luận lý 2> -> <lệnh 2>
    ; ...
    ; true -> <lệnh> %%Dòng này có thể có hoặc không
end
```

Tương tự ta cũng có cú pháp **case...of** (thay thế cho **switch case** trong C/C++) trong Erlang như sau:

```
case <biểu thức> of
    dạng 1 [when <các điều kiện>] -> <lệnh 1>
    ; ...
    ; dạng N [when <các điều kiện>] -> <lệnh N>
    ; _ -> <lệnh> %%Dòng này tương tự như default trong
switch case
end
```

Với **case..of** Chương trình thực hiện kiểm tra **biểu thức** khớp với **dạng** nào và nếu **các điều kiện** ở dạng đó là đúng thì sẽ thực thi **lệnh** tương ứng.

Ví dụ:

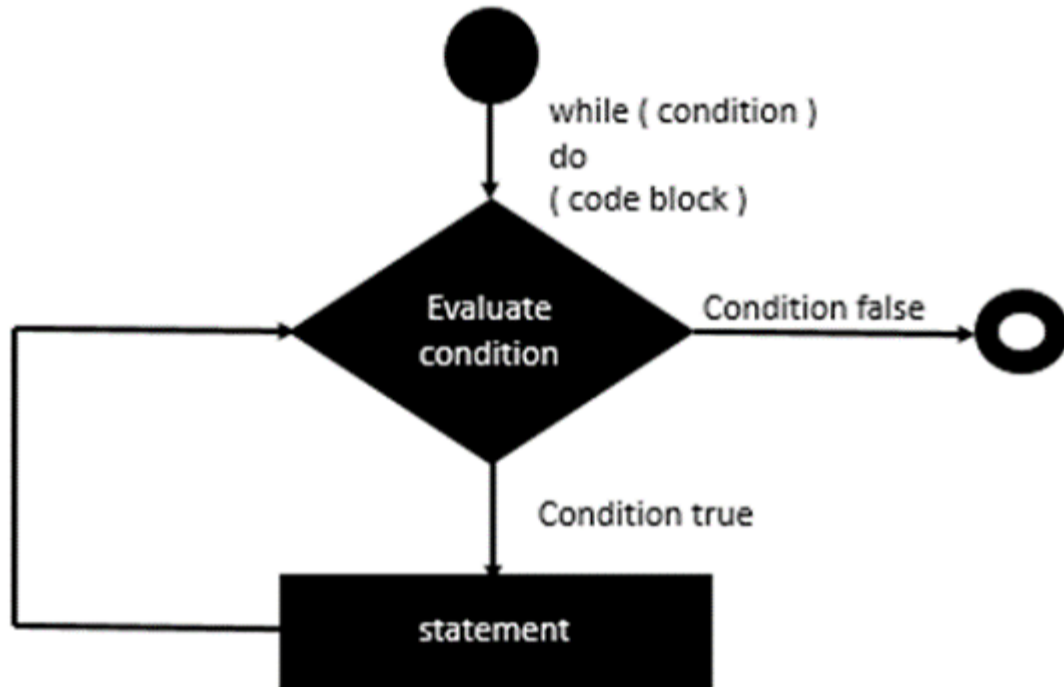
```
case a == b of
    true -> "yes"
    ; false -> "no"
end
```

7.2 CẤU TRÚC VÒNG LẶP

Erlang cũng như các ngôn ngữ lập trình hàm khác, không cung cấp bất kỳ cấu trúc vòng lặp nào như **for** hay **while**. Thay vào đó, ta vận dụng kỹ thuật **đệ quy** được cung cấp trong ngôn ngữ này để thực hiện các **vòng lặp** for hay while.

Vòng lặp WHILE

Chúng ta sẽ cố gắng trình bày biểu diễn vòng lặp **while** tương tự như trong các ngôn ngữ lập trình khác. Ta có lưu đồ thực hiện vòng lặp **while** như sau:



Bây giờ cùng xét một ví dụ về việc sử dụng **đệ quy** để biểu diễn vòng lặp **while** :

```

-module(example).
-export([while/1,while/2, start/0]).

while(L) -> while(L,0).
while([], Acc) -> Acc;

while([_|T], Acc) ->
    io:fwrite("~w~n",[Acc]),
    while(T,Acc+1).

start() ->
    X = [1,2,3,4],
  
```

`while(X).`

Một số điểm cần chú ý trong ví dụ trên:

- Định nghĩa hàm đệ quy tên “while” để mô phỏng việc thực hiện vòng lặp while.
- Đưa danh sách các giá trị được định nghĩa trong biến X (sẽ được khai báo sau) vào hàm “while” trên.
- Hàm “while” sẽ lấy mỗi giá trị trong danh sách này và lưu vào biến tạm ‘Acc’.
- Vòng lặp “while” khi đó sẽ được gọi đệ quy dần đến mỗi giá trị trong danh sách biến X.

Output của ví dụ trên sẽ là:

0

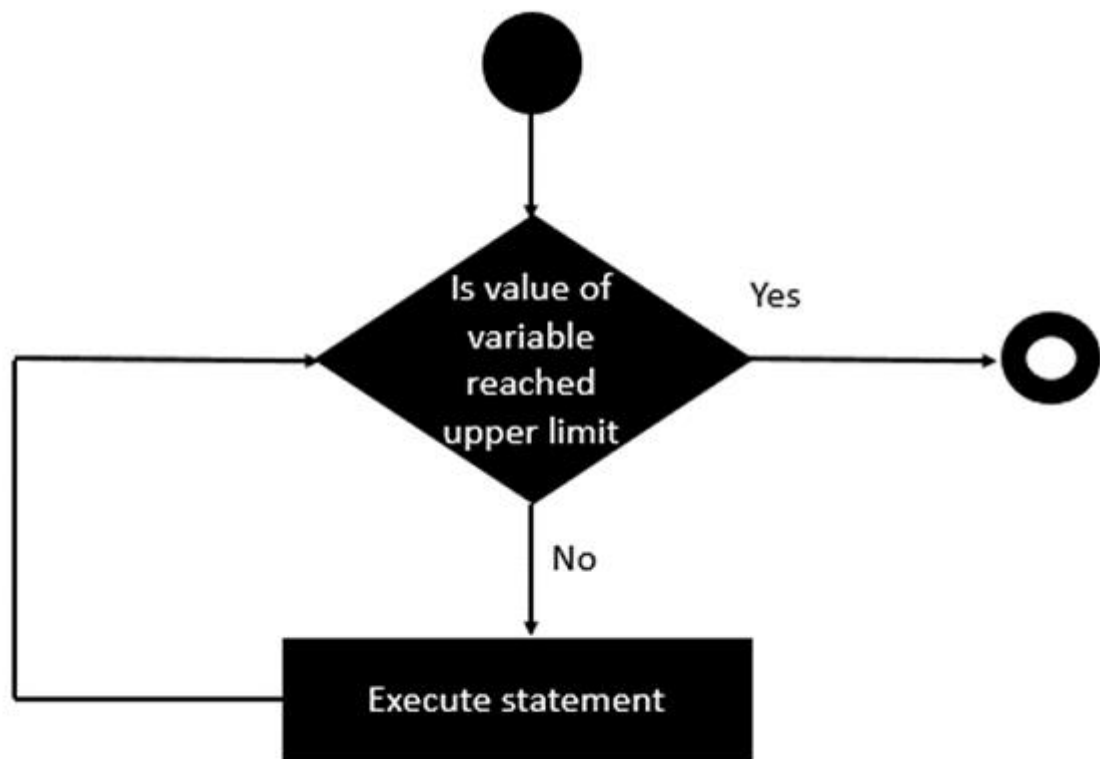
1

2

3

Vòng lặp FOR

Tương tự như vòng lặp while, với vòng lặp **for** ta có lưu đồ thuật toán sau:



Xét ví dụ in ra màn hình một số N lần từ “Hello” sử dụng **đệ quy** (cụ thể ở đây là 5 lần):

```
-module(helloworld).  
-export([for/2,start/0]).  
  
for(0,_) -> [];  
for(N,Term) when N > 0 ->  
    io:fwrite("Hello~n"),  
    [Term|for(N-1,Term)].  
start() -> for(5,1).
```

Các điểm lưu ý:

- Ta định nghĩa hàm đệ quy tên “for” để mô phỏng vòng lặp **for**.
- Trong hàm “for” ta kiểm tra để chắc chắn rằng giá trị đối số N hay giới hạn của nó là một số dương. (Điều này tương tự như kiểm tra điều kiện vòng lặp **for** trong C/C++)

- Sau đó thực hiện hàm đệ quy, với mỗi lần thực hiện ta giảm giá trị của N xuống 1 đơn vị bằng cách truyền giá trị giảm dần của N vào hàm **for**.

Output sẽ là:

Hello

Hello

Hello

Hello

Hello

8. LẬP TRÌNH HÀM

8.1 LẬP TRÌNH HÀM

Các hàm trong Erlang được nhóm lại trong 1 file gọi là Module. Tất cả các hàm phải được định nghĩa trong module. Các hàm thường được gọi theo cú pháp

`Module:Function(Arguments).`

Khi tạo 1 module trước tiên phải khai báo theo cú pháp **-module(Name).**

Tiếp đến cần khai báo các hàm trong module: **-export([Function1/Arity, Function2/Arity, ..., FunctionN/Arity]).**

Sau đó là các hàm chúng ta viết theo cú pháp: `Name (Args) -> Body.` Trong đó *Body* có thể là 1 hoặc nhiều biểu thức Erlang ngăn cách nhau bởi dấu phẩy. Hàm được kết thúc bởi dấu chấm. Erlang không sử dụng return, thay vào đó biểu thức cuối cùng được thực hiện sẽ trả về giá trị của hàm đó.

Ta có thể xem thử 1 module đơn giản:

```

1  -module(useless).
2  -export([add/2, hello/0, greet_and_add_two/1]).
3
4  add(A, B) ->
5      A + B.
6
7  %% Shows greetings.
8  %% io:format/1 is the standard function used to output text.
9  hello() ->
10     io:format("Hello, world!\n").
11
12 greet_and_add_two(X) ->
13     hello(),
14     add(X, 2).

```

Sau khi đã tạo được module thì ta có thể dùng nó bằng cách mở Erlang shell và gọi bằng lệnh:

```

1> cd("d:/erlang").
d:/erlang
ok
2> c("useless").
{ok,useless}

```

Sau đó có thể gọi các hàm theo cú pháp đã nói ở trên:

```

3> useless:add(7, 2).
9
4> useless:hello().
Hello, world!
ok
5> useless:greet_and_add_two(-3).
Hello, world!
-1
6> usesles:not_a_real_function().
** _exception error: undefined function usesles:not_a_real_function/0

```

8.2 ĐỆ QUY

Đệ quy là một phần quan trọng của Erlang (điển hình như việc sử dụng để tạo vòng lặp trong mục 7). Trước tiên ta cùng xét một ví dụ đệ quy đơn giản để tính giai thừa:

```

-module(factorial).

-export([fac/1, start/0]).

```

```

fac(N) when N == 0 -> 1;

```

```
fac(N) when N > 0 -> N*fac(N-1).
```

```
start() ->  
    X = fac(4),  
    io:fwrite("~w",[X]).
```

Trong ví dụ trên:

- Ta định nghĩa một hàm fac(N).
- Ta xác định hàm đệ quy bằng cách gọi đệ quy. Khi $N > 0$ hàm thực thi chính nó với $N-1$, cứ thế đến khi $N = 0$ thì hàm trả về 1.

Output: 24 (4!)

Tiếp cận thực tế Đệ quy

Trong phần này ta sẽ tìm hiểu chi tiết những loại **đệ quy** khác nhau và ứng dụng của chúng trong Erlang.

Length Recursion

Một cách tiếp cận thực tế hơn với đệ quy đó là thông qua ví dụ đơn giản về việc sử dụng đệ quy xác định độ dài của một danh sách. Một danh sách có thể có nhiều giá trị như [1,2,3,4]. Hãy cùng sử dụng đệ quy để có thể thấy được cách xác định độ dài của danh sách này.

```
-module(listlength).  
-export([len/1,start/0]).
```

```
len([]) -> 0;  
len([_|T]) -> 1 + len(T).
```

```
start() ->  
    X = [1,2,3,4],
```

```
Y = len(X),  
io:fwrite("~w",[Y]).
```

Trong ví dụ trên:

- Hàm `len([])` đầu tiên xét trường hợp đặc biệt là danh sách rỗng.
- Pattern `[H|T]` để match với danh sách của một hay nhiều phần tử, danh sách của độ dài một sẽ được định nghĩa `[X|[]]` và danh sách của độ dài hai sẽ được định nghĩa `[X|[Y|[]]]`. Chú ý rằng phần tử thứ 2 trong sách cũng là một danh sách của chính nó. Điều này có nghĩa ta chỉ cần đếm phần tử đầu tiên và hàm sẽ tự gọi chính nó ở phần tử thứ 2. Cho mỗi giá trị trong một danh sách được tính là 1.

Output: 4

Tail Recursion (Đệ quy đuôi)

Để hiểu cách tail recursion làm việc, trước hết ta cần hiểu được 2 dòng code sau của phần trước:

```
len([]) -> 0;  
len([_|T]) -> 1 + len(T).
```

Để thực hiện `1+len(T)`, cần phải thực hiện `len(T)`. Hàm `len(T)` sẽ tự gọi các hàm cần thiết để thực hiện chính nó. Những công việc này sẽ được thực hiện xếp chồng lên nhau cho đến công việc cuối cùng, và khi đó ta mới có được kết quả.

Đệ quy đuôi nhằm mục đích loại bỏ sự cài đặt nhiều lần một hàm bằng cách giảm chính hàm đó khi thực hiện.

Để làm được điều này ta cần phải giữ một biến tạm thời như một tham số trong hàm. Các biến tạm thời nói trên đôi khi được gọi là accumulator và hoạt động như một nơi để lưu trữ các kết quả tính toán khi chúng xảy ra để hạn chế sự tăng trưởng của các lần gọi hàm.

Ví dụ:

```
-module(helloworld).
```



```
-export([tail_len/1,tail_len/2,start/0]).

tail_len(L) -> tail_len(L,0).
tail_len([], Acc) -> Acc;
tail_len([_|T], Acc) -> tail_len(T,Acc+1).

start() ->
    X = [1,2,3,4],
    Y = tail_len(X),
    io:fwrite("~w",[Y]).
```

Output: 4

Duplicate (Nhân bản)

Xét một ví dụ của đệ quy. Lần này ta viết một hàm lấy số nguyên làm tham số đầu, theo sau đó bất kỳ term nào thì đều xem như là tham số thứ hai. Hàm này sẽ tạo một danh sách các bản sao của term này với số lượng bằng số nguyên đó.

```
-module(helloworld).
-export([duplicate/2,start/0]).

duplicate(0,_) ->
    [];
duplicate(N,Term) when N > 0 ->
    io:fwrite("~w,~n",[Term]),
    [Term|duplicate(N-1,Term)].

start() ->
    duplicate(5,1).
```

Output:

```
1,  
1,  
1,  
1,  
1,
```

List Reversal

Không có giới hạn nào khi sử dụng đệ quy trong Erlang. Hãy cùng xem cách mà ta có thể đảo ngược các phần tử của một danh sách sử dụng đệ quy.

Chương trình:

```
-module(helloworld).  
-export([tail_reverse/2,start/0]).  
  
tail_reverse(L) -> tail_reverse(L, []).  
  
tail_reverse([],Acc) -> Acc;  
tail_reverse([H|T],Acc) -> tail_reverse(T, [H|Acc]).  
  
start() ->  
    X = [1,2,3,4],  
    Y = tail_reverse(X),  
    io:fwrite("~w",[Y]).
```

Output: [4,3,2,1]

Trong ví dụ trên:

- Chúng ta lại sử dụng khái niệm các biến tạm thời để lưu trữ mỗi phần tử của danh sách trong một biến gọi là Acc.
- Sau đó gọi đệ quy `tail_reverse`, nhưng lần này ta đảm bảo rằng phần tử cuối cùng được đặt vào một danh sách mới.
- Sau đó, gọi đệ quy `tail_reverse` cho mỗi phần tử trong danh sách.

9. CÁC CẤU TRÚC DỮ LIỆU ĐẶC THÙ VÀ THAO TÁC

9.1 MẢNG

Mảng có thể được mở rộng (fixed size). Mảng được đánh số từ 0. Các phần tử trong mảng có thể được khởi tạo từ đầu hoặc không, nếu không được khởi tạo thì các phần tử nhận giá trị là `atom undefined`.

Ví dụ:

Tạo 1 mảng với 10 phần tử 0-9 không có giá trị khởi tạo nên mặc định là `undefined`:

```
1> A0 = array:new(10).  
{array,10,0,undefined,10}  
2> 10 = array:size(A0).  
10
```

Tạo 1 mảng và gán phần tử 17 giá trị `true`, các phần tử trước đó mang giá trị mặc định:

```
18> A1 = array:set(17, true, array:new()).  
{array,18,100,undefined,  
  {10,  
    {undefined,undefined,undefined,undefined,undefined,  
      undefined,undefined,true,undefined,undefined},  
    10,10,10,10,10,10,10,10,10,10}  
19> array:size(A1).  
18
```

Lấy giá trị của phần tử:

```
20> array:get(17, A1).  
true  
21> array:get(3, A1).  
undefined
```

Khi truy cập vào các phần tử nằm sau phần tử cuối cùng được khởi tạo trong mảng không định sẵn kích thước thì nó sẽ trả về giá trị mặc định:

```
22> array:get(18, A1).
undefined
```

Hàm “sparse” bỏ qua các phần tử mang giá trị mặc định:

```
24> A2 = array:set(4, false, A1).
{array,18,100,undefined,
  {{undefined,undefined,undefined,undefined,false,undefined,
    undefined,undefined,undefined,undefined},
   {undefined,undefined,undefined,undefined,undefined,
    undefined,undefined,true,undefined,undefined},
   10,10,10,10,10,10,10,10,10}}
25> array:sparse_to_orddict(A2).
[{4,false},{17,true}]
```

Mảng có kích thước không định sẵn có thể được đặt lại kích thước:

```
26> A3 = array:fix(A2).
{array,18,0,undefined,
  {{undefined,undefined,undefined,undefined,false,undefined,
    undefined,undefined,undefined,undefined},
   {undefined,undefined,undefined,undefined,undefined,
    undefined,undefined,true,undefined,undefined},
   10,10,10,10,10,10,10,10,10}}
_
```

Trong mảng có kích thước, ta không thể truy cập các phần tử ngoài mảng:

```
29> (catch array:set(18, true, A3)).
{'EXIT',{badarg,[{array,set,3,
  [{file,"array.erl"},{line,583}]},
  {erl_eval,do_apply,6,[{file,"erl_eval.erl"},{line,674}]},
  {erl_eval,expr,5,[{file,"erl_eval.erl"},{line,431}]},
  {shell,exprs,7,[{file,"shell.erl"},{line,687}]},
  {shell,eval_exprs,7,[{file,"shell.erl"},{line,642}]},
  {shell,eval_loop,3,[{file,"shell.erl"},{line,627}]}]}}
30> (catch array:get(18, A3)).
{'EXIT',{badarg,[{array,get,2,
  [{file,"array.erl"},{line,641}]},
  {erl_eval,do_apply,6,[{file,"erl_eval.erl"},{line,674}]},
  {erl_eval,expr,5,[{file,"erl_eval.erl"},{line,431}]},
  {shell,exprs,7,[{file,"shell.erl"},{line,687}]},
  {shell,eval_exprs,7,[{file,"shell.erl"},{line,642}]},
  {shell,eval_loop,3,[{file,"shell.erl"},{line,627}]}]}}
_
```

Một số hàm thường dùng liên quan đến mảng:

```
default(Array :: array(Type)) -> Value :: Type
```

lấy giá trị mặc định của các phần tử trong

```
4> A1 = array:set(17, true, array:new()).
{array,18,100,undefined,
  {10,
    {undefined,undefined,undefined,undefined,undefined,
      undefined,undefined,true,undefined,undefined},
    10,10,10,10,10,10,10,10,10}}
5> array:default(A1).
mảngundefined
```

fix(Array :: [array](#)(Type)) -> [array](#)(Type)

Cố định kích thước của mảng có thể xem ví dụ ở trên.

from_list(List :: [Value :: Type], Default :: term()) ->
[array](#)(Type)

Chuyển từ list sang mảng

```
6> L = [1, 2, 3, 4].
[1,2,3,4]
7> array:from_list(L, undefined).
{array,4,10,undefined,
  {1,2,3,4,undefined,undefined,undefined,undefined,undefined,
    undefined}}
8> array:from_list(L).
{array,4,10,undefined,
  {1,2,3,4,undefined,undefined,undefined,undefined,undefined,
    undefined}}
9> array:from_list(L, dacrom).
{array,4,10,dacrom,
  {1,2,3,4,dacrom,dacrom,dacrom,dacrom,dacrom,dacrom}}
10> array:from_list(L, 0).
{array,4,10,0,{1,2,3,4,0,0,0,0,0,0}}

from_orddict(Orddict :: indx pairs(Value :: Type),
  Default :: Type) ->
array(Type)
```

Chuyển 1 list dưới dạng các cặp {Index, Value} sang

mảng

```
11> OrdList = [{1, 1}, {6, 6}, {7, 7}, {8, 8}].
[{1,1},{6,6},{7,7},{8,8}]
12> array:from_orddict(OrdList).
{array,9,10,undefined,
  {undefined,1,undefined,undefined,undefined,undefined,6,7,8,
    undefined}}
13> array:from_orddict(OrdList, 0).
{array,9,10,0,{0,1,0,0,0,0,6,7,8,0}}
```

get(I :: [array indx\(\)](#), Array :: [array](#)(Type)) -> Value :: Type

Lấy giá trị của phần tử I trong mảng, có thể xem lại trong các ví dụ trên.

is_array(X :: term()) -> boolean()

trả về true nếu X là 1 mảng ngược lại ra false.

is_fix(Array :: [array\(\)](#)) -> boolean()

trả về true nếu mảng đã cố định kích thước.

new() -> [array\(\)](#)

tạo 1 mảng với kích thước không cố định, ban đầu kích thước là 0.

new(Options :: [array opts\(\)](#)) -> [array\(\)](#)

tạo mảng với những thiết lập riêng do ta quy định. Options có thể bao gồm 1 hoặc nhiều thành phần trong các thành phần sau:

`N::integer() >= 0 or {size, N::integer() >= 0}`

Kích thước ban đầu của mảng

`fixed or {fixed, true}`

cố định kích thước mảng

`{fixed, false}`

Không cố định kích thước

`{default, Value}`

Đặt giá trị mặc định cho mảng.

```
22> array:new(100).
{array,100,0,undefined,100}
23> array:new({default, 0}).
{array,0,10,0,10}
24> array:new([ {size, 10}, {fixed, false}, {default, -1} ]).
{array,10,10,-1,10}
```

relax(Array :: [array](#)(Type)) -> [array](#)(Type)

biến mảng thành mảng không cố định kích thước, ngược với fix/1.

reset(I :: [array indx\(\)](#), Array :: [array](#)(Type)) -> [array](#)(Type)

đặt lại giá trị mặc định cho phần tử I

```
resize(Array :: array(Type)) -> array(Type)
```

thay đổi kích thước bằng số phần tử sử dụng (tính từ phần tử 0 đến phần tử cuối cùng có giá trị khác mặc định).

```
resize(Size :: integer() >= 0, Array :: array(Type)) ->  
    array(Type)
```

thay đổi kích thước bằng Size.

```
set(I :: array\_indx\(\), Value :: Type, Array :: array(Type))  
->  
    array(Type)
```

đặt giá trị Value cho phần tử I trong mảng.

```
size(Array :: array()) -> integer() >= 0
```

lấy kích thước của mảng.

```
sparse_size(Array :: array()) -> integer() >= 0
```

lấy số phần tử của mảng tính từ phần tử 0 đến phần tử cuối cùng có giá trị khác mặc định.

```
sparse_to_list(Array :: array(Type)) -> [Value :: Type]
```

chuyển mảng sang list, lược bỏ các phần tử mang giá trị mặc định.

```
sparse_to_orddict(Array :: array(Type)) ->  
    indx\_pairs(Value :: Type)
```

chuyển mảng sang ordered list gồm các cặp giá trị {Index, Value}, lược bỏ các phần tử mang giá trị mặc định.

```
to_list(Array :: array(Type)) -> [Value :: Type]
```

chuyển mảng sang list.

```
to_orddict(Array :: array(Type)) -> indx\_pairs(Value ::  
Type)
```

chuyển mảng sang ordered list.

9.2 CHUỖI

Ta có module string cung cấp các hàm xử lý chuỗi.

Chuỗi được biểu diễn bởi `unicode:chardata()` là list các điểm mã (codepoint), binaries với UTF-8-encoded codepoints (UTF-8 binaries), hoặc kết hợp cả 2.

```
"abcd"           is a valid string
<<"abcd">>       is a valid string
["abcd"]         is a valid string
<<"abc..äö"/utf8>> is a valid string
<<"abc..äö">>     is NOT a valid string,
                  but a binary with Latin-1-encoded codepoints
[<<"abc">>, "..äö"] is a valid string
[atom]           is NOT a valid string
```

Module này hoạt động trên các grapheme cluster. Một grapheme cluster là một kí tự mà ta nhận thấy được, nó có thể được biểu diễn bởi một số codepoints.

```
"ä"  [229] or [97, 778]
"e°" [101, 778]
```

Độ dài của chuỗi "β↑ẻ" là 3 nhưng nó có biểu diễn dưới dạng codepoints là [223,8593,101,778] hoặc dạng UTF-8 binary là <<195, 159, 226, 134, 145, 101, 204, 138>>.

Cắt hay ghép chuỗi được thực hiện trên kiểu grapheme cluster. Không có sự kiểm tra rằng chuỗi kết quả là hợp lệ không.

Đa số các hàm yêu cầu input phải thỏa mãn một kiểu nào đó.

Hàm có thể ngưng nếu chuỗi input không hợp lệ.

Có những trường hợp giá trị trả về có cùng kiểu với input. Nghĩa là binary input trả về binary output, list input trả về list output, và kết hợp cả 2 thì trả về kết hợp cả 2.

```
1> string:trim(" sarah ").
"sarah"
2> string:trim(<<" sarah ">>).
<<"sarah">>
3> string:lexemes("foo bar", " ").
["foo", "bar"]
4> string:lexemes(<<"foo bar">>, " ").
[<<"foo">>, <<"bar">>]
```


Một số hàm thông dụng:

casefold(String :: [unicode:chardata\(\)](#)) -> [unicode:chardata\(\)](#)

trả về 1 chuỗi với các kí tự trong chuỗi ban đầu được viết thường.

```
1> string:casefold("Q and ß SHARP S").
"q and ss sharp s"
```

chomp(String :: [unicode:chardata\(\)](#)) -> [unicode:chardata\(\)](#)

trả về chuỗi sau khi xóa phần dư thừa \n hoặc \r\n ở cuối.

```
182> string:chomp(<<"\nHello\n\n">>).
<<"\nHello">>
183> string:chomp("\nHello\r\r\n").
"\nHello\r"
```

equal(A, B) -> boolean()

equal(A, B, IgnoreCase) -> boolean()

equal(A, B, IgnoreCase, Norm) -> boolean()

trong đó

A = B = [unicode:chardata\(\)](#)

IgnoreCase = boolean()

Norm = none | nfc | nfd | nfkc | nfkd

Hàm trả về true nếu A và B bằng nhau, ngược lại ra false. Nếu IgnoreCase là true thì hàm sẽ [casefold](#)ing trước khi so sánh. Nếu Norm là none thì hàm sẽ so sánh theo kiểu thông thường.

```
1> string:equal("äö", <<"äö"/utf8>>).
true
2> string:equal("äö", unicode:characters_to_nfd_binary("äö")).
false
3> string:equal("äö", unicode:characters_to_nfd_binary("ÄÖ"), true, nfc).
true
```

find(String, SearchPattern) -> [unicode:chardata\(\)](#) | nomatch

find(String, SearchPattern, Dir) -> [unicode:chardata\(\)](#) |

nomatch

trong đó

String = **SearchPattern** = [unicode:chardata\(\)](#)

Dir = [direction\(\)](#)

Xóa những kí tự trước SearchPattern trong String và trả về phần còn lại của chuỗi hoặc nomatch nếu không tìm thấy. Dir có thể là leading hoặc trailing cho biết chiều duyệt là từ đầu về cuối hay từ cuối về đầu.

```
1> string:find("ab..cd..ef", "..").
"..cd..ef"
2> string:find(<<"ab..cd..ef">>, "..", trailing).
<<"..ef">>
3> string:find(<<"ab..cd..ef">>, "x", leading).
nomatch
4> string:find("ab..cd..ef", "x", trailing).
nomatch
```

is_empty(String :: [unicode:chardata\(\)](#)) -> boolean()

trả về true nếu String là chuỗi rỗng, ngược lại ra false.

```
1> string:is_empty("foo").
false
2> string:is_empty(["",<<>>]).
true
```

length(String :: [unicode:chardata\(\)](#)) -> integer() >= 0

trả về số kí tự dạng grapheme cluster của String.

```
1> string:length("ð;e").
3
2> string:length(<<195,159,226,134,145,101,204,138>>).
3
```

lowercase(String :: [unicode:chardata\(\)](#)) -

> [unicode:chardata\(\)](#)

chuyển thành chuỗi viết thường

```
2> string:lowercase(string:uppercase("Michał")).
"michał"
```

**prefix(String :: [unicode:chardata\(\)](#), Prefix
:: [unicode:chardata\(\)](#)) ->**
nomatch | [unicode:chardata\(\)](#)

nếu Prefix là tiền tố của String thì xóa và trả về phần còn lại của String, ngược lại trả về nomatch.

```
1> string:prefix(<<"prefix of string">>, "pre").
<<"fix of string">>
2> string:prefix("pre", "prefix").
nomatch
```

replace(String, SearchPattern, Replacement) ->
[\[unicode:chardata\(\)\]](#)

replace(String, SearchPattern, Replacement, Where) ->
[\[unicode:chardata\(\)\]](#)

trong đó

String = SearchPattern = Replacement = [unicode:chardata\(\)](#)

Where = [direction\(\)](#) | all

thay thế SearchPattern trong String bằng Replacement. Where có thể là leading, trailing hay all cho biết vị trí thay thế là đầu tiên từ trái qua, phải qua hay tất cả.

```
1> string:replace(<<"ab..cd..ef">>, "..", "*").
[<<"ab">>, "*", <<"cd..ef">>]
2> string:replace(<<"ab..cd..ef">>, "..", "*", all).
[<<"ab">>, "*", <<"cd">>, "*", <<"ef">>]
```

reverse(String :: [unicode:chardata\(\)](#)) ->
[\[grapheme cluster\(\)\]](#)

trả về 1 list đảo ngược của các kí tự dạng grapheme cluster trong String.

```
1> Reverse = string:reverse(unicode:characters_to_nfd_binary("ÄÄÖ")).
[[79,776],[65,776],[65,778]]
2> io:format("~ts~n",[Reverse]).
ÄÄÖ
```

`split(String, SearchPattern) -> [unicode:chardata\(\)]`

`split(String, SearchPattern, Where) -> [unicode:chardata\(\)]`

Chia String thành nhiều phần tại các vị trí mà SearchPattern xuất hiện. Where có thể là leading, trailing hay all.

```
0> string:split("ab..bc..cd", "..").
["ab","bc..cd"]
1> string:split(<<"ab..bc..cd">>,"..",trailing).
[<<"ab..bc">>,<<"cd">>]
2> string:split(<<"ab..bc...cd">>,"..",all).
[<<"ab">>,<<"bc">>,<<"">>,<<"cd">>]
```

`to_float(String) -> {Float, Rest} | {error, Reason}`

trong đó

String = [unicode:chardata\(\)](#)

Float = `float()`

Rest = [unicode:chardata\(\)](#)

Reason = `no_float` | `badarg`

String phải có phần đầu là một số thực hợp lệ, phần còn lại không phải số thực được đưa vào Rest.

```
> {F1,Fs} = string:to_float("1.0-1.0e-1"),
> {F2,[]} = string:to_float(Fs),
> F1+F2.
0.9
> string:to_float("3/2=1.5").
{error,no_float}
> string:to_float("-1.5eX").
{-1.5,"eX"}
```

`to_integer(String) -> {Int, Rest} | {error, Reason}`

trong đó

```
String = unicode:chardata\(\)
```

```
Int = integer()
```

```
Rest = unicode:chardata\(\)
```

```
Reason = no_integer | badarg
```

String phải có phần đầu là một số nguyên hợp lệ, phần còn lại không phải số nguyên được đưa vào Rest.

```
to_graphemes(String :: unicode:chardata\(\)) ->
\[grapheme cluster\(\)\]
```

chuyển String thành 1 list gồm các grapheme cluster.

```
uppercase(String :: unicode:chardata\(\)) -
> unicode:chardata\(\)
```

chuyển String thành chuỗi in hoa.

```
1> string:uppercase("Michał").
"MICHAŁ"
```

```
sub_string(String, Start) -> SubString
```

```
sub_string(String, Start, Stop) -> SubString
```

trong đó

```
String = SubString = string()
```

```
Start = Stop = integer() >= 1
```

Trả về chuỗi con của String lấy từ vị trí Start đến cuối hoặc đến vị trí Stop.

```
sub_string("Hello World", 4, 8).
"lo Wo"
```

```
substr(String, Start) -> SubString
```

```
substr(String, Start, Length) -> SubString
```

trong đó

```
String = SubString = string()
```

```
Start = integer() >= 1
```

```
Length = integer() >= 0
```


append(Key, Value, Dict1) -> Dict2

trong đó

Dict1 = Dict2 = [dict](#)(Key, Value)

Thêm giá trị mới Value vào danh sách các giá trị liên kết với Key.

```
5> D0 = dict:new(),
5> D1 = dict:store(1, [], D0),
5> D2 = dict:append(1, 1, D1),
5> D3 = dict:append(1, 2, D2).
{dict,1,16,16,8,80,48,
  {[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[]},
  _ {[[],[],[],[],[],[],[],[],[],[],[],[],[[1,1,2]],[],[],[],[]]]}}
```

append_list(Key, ValList, Dict1) -> Dict2

tương tự trên nhưng có thể thêm vào giá trị là list.

```
6> D4 = dict:append_list(1, [3, 4], D3).
{dict,1,16,16,8,80,48,
  {[[],[],[],[],[],[],[],[],[],[],[],[],[],[],[]},
  _ {[[],[],[],[],[],[],[],[],[],[],[],[[1,1,2,3,4]],[],[],[]]]}}
```

fetch(Key, Dict) -> Value

trả về các giá trị ứng với Key trong Dict.

```
7> dict:fetch(1, D4).
[1,2,3,4]
```

fetch_keys(Dict) -> Keys

trả về list gồm các key có trong Dict.

```
8> dict:fetch_keys(D4).
[1]
```

take(Key, Dict) -> {Value, Dict1} | error

trả về giá trị ứng với Key và 1 dictionary mới không chứa giá trị đó.

```
9> dict:take(1, D4).
{[1,2,3,4],
 {dict,0,16,16,8,80,48,
  {[[],[],[],[],[],[],[],[],[],[],[],[],[],[],[]},
  _ {[[],[],[],[],[],[],[],[],[],[],[],[],[],[],[]]]}}}
10> dict:take(2, D4).
error
```

find(Key, Dict) -> {ok, Value} | error

tìm giá trị ứng với Key trong Dict.

```
11> dict:find(1, D4).  
{ok,[1,2,3,4]}
```

```
is_key(Key, Dict) -> boolean()
```

kiểm tra Key có phải là 1 khóa trong Dict không.

```
12> dict:is_key(1, D4).  
true  
13> dict:is_key(2, D4).  
false
```

10. TRỪU TƯỢNG HÓA DỮ LIỆU

Trong C, ta có kiểu cấu trúc với cú pháp khai báo sau:

```
struct <tên cấu trúc>  
{  
    <Khai báo thành phần 1>;  
    <Khai báo thành phần 2>;  
    ...  
};
```

Hay khái báo hằng enum:

```
enum <tên enum>  
{  
    //liệt kê các giá trị  
    //Mỗi enum được phân cách với nhau bởi dấu phẩy, không phải chấm  
    //phẩy  
};
```

Có nhiều sự lựa chọn để sử dụng Enum trong Erlang:

Macros

Nhiều lập trình viên định nghĩa macros với những cái tên có nghĩa và giá trị duy nhất.


```

1 | -define(AUTH_ERROR, 404).
2 | -define(FORMAT_ERROR, 400).
3 | -define(INTERNAL_ERROR, 500).

```

Lập trình viên trình độ cao thậm chí định nghĩa macro kiểm tra 1 giá trị có thuộc về enum hay không.

```

1 | -define(IS_ERROR(V), (
2 |     V == ?AUTH_ERROR orelse
3 |     V == ?FORMAT_ERROR orelse
4 |     V == ?INTERNAL_ERROR
5 | )).
6 | handle_error(E) when ?IS_ERROR(E) ->
7 |     case E of
8 |         ?AUTH_ERROR -> abort();
9 |         ?FORMAT_ERROR -> abort();
10 |        ?INTERNAL_ERROR -> retry()
11 |     end.

```

Vấn đề với cách dùng này là sự duy nhất của giá trị không được kiểm soát bởi trình biên dịch. Có thể dễ dàng thêm 1 lỗi vào enum với cùng 1 giá trị gán cho 2 phần tử được liệt kê.

```

1 | -define(NOT_FOUND, 404).

```

Nó phá vỡ logic của enum.

Atoms

Tập hợp các atom có thể được dùng để liệt kê enum. Các atom là duy nhất trong toàn bộ hệ thống.

```

1 | handle_error(E) ->
2 |     case E of
3 |         auth_error -> abort();
4 |         format_error -> abort();
5 |         internal_error -> retry()
6 |     end.

```

Vấn đề duy nhất là kiểm tra xem 1 atom có phải 1 phần tử của enum. Vấn đề này xảy ra nếu ta viết sai tên atom, ví dụ auth_eror thay vì auth_error. Có thể giải quyết bằng cách định nghĩa 1 hàm macro kiểm tra atom có thuộc enum không.

```
1  -define(IS_ERROR(V), (  
2      V := auth_error orelse  
3      V := format_error orelse  
4      V := internal_error  
5  )).  
6  handle_error(E) when ?IS_ERROR(E) ->  
7      ...,  
8      ok.  
9      ...  
10 handle_error(auth_error).
```

Records

Định nghĩa record

```
1  -record(http_errors,{  
2      auth_error,  
3      format_error,  
4      internal_error  
5  }).
```

Mỗi phần tử của enum có thể được xem là 1 phần tử của record.

```
1  handle_error(E) ->  
2      case E of  
3          #http_errors.auth_error -> abort();  
4          #http_errors.format_error -> abort();  
5          #http_errors.internal_error -> retry()  
6      end.  
7  handle_error(#http_errors.auth_error).
```

Rõ ràng tất cả các phần tử enum là duy nhất vì mỗi phần tử của record là duy nhất, và nó được đảm bảo bởi trình biên dịch.

Bất kỳ sai sót nào trong tên sẽ gây ra lỗi biên dịch.

Nếu dòng định nghĩa record ("record(http_errors,{") nằm trên dòng đầu tiên của file, và mỗi phần tử nằm trên 1 dòng mới, mỗi enum sẽ có 1 giá trị nguyên, bằng với số của dòng ở mã nguồn, ở nơi nó được định nghĩa.

```
1  -record(http_errors,{  
2      auth_error,  
3      format_error,  
4      internal_error  
5  }).
```

Ví dụ, trong enum http_errors phần tử auth_error nằm ở dòng 2 nên giá trị nguyên của nó là 2.

Thực thi enum dùng records rất giống với C. Nó thể hiện 1 hằng với 1 số nguyên. Nó tự động gán 1 giá trị nguyên cho mỗi phần tử trong enum. Giới hạn duy nhất là giá trị nguyên cho phần tử đầu tiên luôn bằng 2 và không thể thay đổi.

11. TẬP TIN - THAO TÁC VỚI FILE TRONG ERLANG

Để ví dụ, giả sử có 1 file là NewFile.txt chứa những dòng sau:

Vidu1

Vidu2

Vidu3

11.1.1 Đọc nội dung file 1 dòng 1 lần đọc

Thao tác trên file được thực hiện bằng cách dùng thư viện file. Giống như trong C/C++, để đọc file, ta cần mở file ra trước và đọc. Sau đây là **cú pháp**:

Mở 1 file – *open(File, Mode)*.

Đọc 1 file – *read(FileHandler, Numberofbytes)*.

Những tham số

File – Địa chỉ của file cần mở.

Mode – Trạng thái mở file.

Sau đây là vài mode có sẵn

Read, Write, Append giống như C/C++.

Exclusive – Mở file để viết, file được tạo ra nếu nó chưa có. Nếu file đã tồn tại, mở file sẽ trả về {error, exist}.

FileHandler – tên stream.

Numberofbytes – Số bytes của thông tin cần đọc từ file.

Giá trị trả về

open(File, Mode) – Trả về stream của file, nếu thành công.

`read(FileHandler, NumberOfBytes)` – Trả về thông tin muốn đọc từ file.

```
1 -module(helloworld).
2 -export([start/0]).
3
4 start() ->
5     {ok, File} = file:open("Newfile.txt",[read]),
6     Txt = file:read(File,1024 * 1024),
7     io:fwrite("~p~n",[Txt]).
```

Khi chạy chương trình trên sẽ được kết quả sau:

```
12> helloworld:start().
{ok,"Vidu1\r\nVidu2\r\nVidu3"}
ok
```

11.1.2 Đọc toàn bộ nội dung file 1 lần

Cú pháp:

`read_file(filename).`

Tham số

`filename` – Tên file cần đọc.

Giá trị trả về

Toàn bộ nội dung file.

Ví dụ:

```
1 -module(helloworld).
2 -export([start/0]).
3
4 start() ->
5     Txt = file:read_file("Newfile.txt"),
6     io:fwrite("~p~n",[Txt]).
```

Khi chạy chương trình trên, ta được kết quả sau:

```
3> helloworld:start().
{ok,<<"Vidu1\r\nVidu2\r\nVidu3">>}
ok
```

11.1.3 Viết nội dung ra 1 file

Cú pháp:

`write(FileHandler, text)`

Tham số

FileHandler – tên stream.

Text – Nội dung muốn thêm vào file.

Giá trị trả về

None

Ví dụ:

```
1  -module(helloworld).
2  -export([start/0]).
3
4  start() ->
5      {ok, Fd} = file:open("Newfile.txt", [write]),
6      file:write(Fd,"New Line").
```

Output

Khi chạy chương trình, dòng “New Line” sẽ được viết vào file. Vì mode là “write” nên nếu trong file có sẵn nội dung, chúng sẽ bị ghi đè.

Để thêm vào sau nội dung đã có trong file, đổi mode thành ‘append’ như chương trình sau:

```
1  -module(helloworld).
2  -export([start/0]).
3
4  start() ->
5      {ok, Fd} = file:open("Newfile.txt", [append]),
6      file:write(Fd,"New Line").
```

11.1.4 Sao chép 1 file đã tồn tại

Cú pháp:

copy(source, destination)

Tham số

Source – tên file nguồn cần sao chép.

Destination – Đường dẫn và tên file.

Giá trị trả về

None

Ví dụ

```
1 -module(helloworld).  
2 -export([start/0]).  
3  
4 start() ->  
5   file:copy("Newfile.txt","Duplicate.txt").|
```

Output

1 file tên Duplicate.txt sẽ được tạo ở cùng địa chỉ với NewFile.txt và có cùng nội dung với NewFile.txt.

11.1.5 Xóa 1 file

Cú pháp:

delete(filename)

Tham số

Filename – Tên và địa chỉ của file cần xóa.

Giá trị trả về

None

Ví dụ

```
1 -module(helloworld).  
2 -export([start/0]).  
3  
4 start() ->  
5   file:delete("Duplicate.txt").|
```

Output

Nếu file Duplicate.txt tồn tại, nó sẽ bị xóa.

11.1.6 Liệt kê nội dung 1 thư mục

Cú pháp:

list_dir(directory)

Tham số

Directory – Thư mục chứa nội dung cần liệt kê

Giá trị trả về

Danh sách các file trong thư mục

Ví dụ:

```
1 -module(helloworld).
2 -export([start/0]).
3
4 start() ->
5   io:fwrite("~p~n",[file:list_dir(".")]).
```

Output

```
i3> helloworld:start().
{ok,["cau1.beam","cau1.erl","cau2.beam","cau2.erl","foo.txt",
    "helloworld.beam","helloworld.erl","input.txt","NewFile.txt","test.erl"]}
ok
... ■
```

11.1.7 Tạo thư mục mới

Cú pháp:

make_dir(directory)

Tham số

Directory – Tên thư mục cần được tạo

Giá trị trả về

Trạng thái của quá trình tạo thư mục. Nếu thành công, tin nhắn {Ok} sẽ hiện ra.

Ví dụ:

```
1 -module(helloworld).
2 -export([start/0]).
3
4 start() ->
5   io:fwrite("~p~n",[file:make_dir("newdir")]).
```

Output

Thư mục mới tên 'newdir' sẽ được tạo.

11.1.8 Đổi tên 1 file đã tồn tại

Cú pháp:

rename(oldfilename, newfilename)

Tham số

Oldfilename – tên file muốn đổi tên.

Newfilename – tên file sẽ đổi thành.

Giá trị trả về

Trạng thái của quá trình đổi tên. Nếu thành công, tin nhắn {Ok} sẽ hiện ra.

Ví dụ:

```
1 -module(helloworld).  
2 -export([start/0]).  
3  
4 start() ->  
5   io:fwrite("~p~n",[file:rename("Newfile.txt","Renamedfile.txt"))].
```

Output

File NewFile.txt sẽ được đổi tên thành Renamedfile.txt.

11.1.9 Xác định kích thước của file

Cú pháp:

file_size(filename)

Tham số

Filename – Tên của file muốn xác định kích thước.

Giá trị trả về

1 số cho biết kích thước của file

Ví dụ:

```
1 -module(helloworld).  
2 -export([start/0]).  
3  
4 start() ->  
5   io:fwrite("~w~n",[filelib:file_size("Renamedfile.txt"))].
```

Output

```
19> helloworld:start().  
8  
ok
```

Kích thước file Renamedfile.txt với 1 dòng "New Line" là 8.

12. CHUẨN VIẾT CHƯƠNG TRÌNH

12.1 CẤU TRÚC TẬP TIN

Headers

File header nên có ngày code, 1 đoạn mô tả ngắn mục đích của file và tag những thứ cần thiết

```
%%%=====
%%% @doc One-line description of this module.
%%% More detailed commentary here.
%%% ...
%%%
%%% @copyright 2012 %%%
%%% @reference
%%% ...
%%%
%%% @end
%%%=====
```

Thuộc tính biên dịch

Những file thường có module, export, khai báo thư viện.

Những tên hàm nên được liệt kê theo thứ tự chữ cái. Nếu cần thiết, chia khai báo hàm ra cho rõ ràng.

```
--module(quux).

%% API
-export([ bar/2
         , baz/2
         , frob/4
         ...
        ]).

%% RPC
-export([twiddle/1]).

--include_lib("...").
```

Khai báo record

Dùng kiểu chú thích cho record

```
-record(quux,  
    { foo :: [integer()]  
    , bar :: ok | error  
    , baz :: _  
    })).
```

Định nghĩa hàm

Hàm nên được chú thích (mục đích của hàm)

```
-spec frob(...) -> ...  
%% @doc ...  
frob() ->  
    ...
```

Comments

Số dấu '%' trước 1 comment chỉ ra ý định của comment.

3 dấu '%' được dùng cho comments áp dụng đối với cả file nguồn, thông thường là file header.

```
%%%=====
%%% @doc One-line description of this module.
%%% More detailed commentary here.
%%% ...
%%%
%%% @copyright 2012 %%%
%%% @reference
%%% ...
%%%
%%% @end
%%%=====
```

2 dấu '%' được dùng nhiều. Những dòng comment này nên đặt bên trên hàm, lệnh, hay biểu thức mà nó nói đến, không đặt ở sau. Những comment này được dùng để giải thích tại sao code được viết như thế, hay chỉ người đọc cách thực thi chi tiết.

```
%% wrapper for calling bar/2  
foo(X) ->  
    bar(self(), X).
```

1 dấu '%' dùng làm nổi bật vài đặc điểm của 1 dòng lệnh, và ko nên đặt nó trên 1 dòng riêng.

```
foo(X) ->
...
frob(..., X+1), % X must be adjusted by 1 for calling frob()
...
```

12.2 SỰ THỤT ĐẦU DÒNG

Những dòng lệnh đơn [Hàm|Case|If|...] nên được điều chỉnh thẳng theo cột mũi tên

```
map(F, Xs)                -> map(F, Xs, []).
map(_, [], Acc)           -> {ok, lists:reverse(Acc)};
map(F, [X|Xs], Acc)       -> map_(F(X), F, Xs, Acc).
map_({error, _} = E, _, _, _) -> E;
map_({ok, Y}, F, Xs, Acc) -> map(F, Xs, [Y|Acc]).
```

Dùng kiểu ‘dấu phẩy viết trước’ cho những chữ không vừa trên 1 dòng

```
snarfs() ->
[ snarf
, snorf
, snurf
...
].
```

Cách ra sau mỗi dấu phẩy

```
{1, 2, 3} %yes
{1,2,3}   %no
```

Cách ra 2 bên toán tử.

```
1 + 2      %yes
1+2        %no
```

Hàm hay chữ được dùng như bảng tra cứu nên có cột được điều chỉnh đồng dạng.

```
%%      balance  where      balance  whole tree  to be
%%      before   inserted   after   increased  rebalanced
table('-', left) -> {'<', yes, no };
table('-', right) -> {'>', yes, no };
table('<', left) -> {'-', no, yes};
table('<', right) -> {'-', no, no };
table('>', left) -> {'-', no, no };
table('>', right) -> {'-', no, yes}.
```

12.3 ĐẶT TÊN

Tên biến nên viết theo kiểu

```
FromDate, ToDate
```

CamelCase

Tên hàm nên viết thường và các từ nên viết tách ra bằng dấu '_'

```
get_pay_dates
```

Đặt tên ngắn (dưới 20 kí tự), nhưng 1 cái tên có tính mô tả quan trọng hơn 1 cái tên ngắn.

12.4 KÍCH THƯỚC CỦA NHỮNG TẬP HỢP

Modules

Đặc tính quan trọng nhất của module là nó chứa các hàm liên quan. Kích thước của 1 module không quá quan trọng, miễn là nó không chứa code không liên quan.

Hàm

Hầu hết các hàm chỉ có vài dòng; hàm không nên vượt quá 60 dòng.

13. MỘT SỐ CHỦ ĐỀ NÂNG CAO

13.1 WEB PROGRAMING

Trong Erlang có sẵn một thư viện để tạo web servers là **inets library**. Hãy cùng xem xét một số hàm có sẵn từ đây dành cho lập trình web. Một hàm có thể thực hiện như một máy chủ HTTP, hay còn gọi là httpd để xử lý các yêu cầu về HTTP.

Máy chủ thực hiện được nhiều tính năng, chẳng hạn như:

- Secure Sockets Layer (SSL)
- Erlang Scripting Interface (ESI)
- Common Gateway Interface (CGI)

- User Authentication (sử dụng Mnesia, Dets hay plain text database)
- Common Logfile Format (có hoặc không có disk_log(3) support)
- URL Aliasing
- Action Mappings
- Directory Listings

Công việc đầu tiên cần làm là khởi động web library bằng lệnh:

```
inets:start()
```

Bước tiếp theo là chạy các hàm start của thư viện inets để web server có thể được thực thi.

Sau đây là ví dụ việc tạo quy trình web server trong Erlang.

```
-module(helloworld).  
-export([start/0]).  
  
start() ->  
    inets:start(),  
    Pid = inets:start(httpd, [{port, 8081}, {server_name,"httpd_test"},  
    {server_root,"D://tmp"},{document_root,"D://tmp/htdocs"},  
    {bind_address, "localhost"}}], io:fwrite("~p",[Pid])).
```

Các điểm cần lưu ý:

- port number là duy nhất và không được sử dụng bởi các chương trình khác. **httpd service** sẽ được khởi động trên port này.
- **server_root** và **document_root** là các thông số bắt buộc.

Output:

```
{ok,<0.42.0>}
```

Để thực thi **Hello world web server** trong Erlang, thực hiện các bước sau

Bước 1 – Viết các dòng code sau vào editor:

```
-module(helloworld).  
-export([start/0,service/3]).  
  
start() ->  
    inets:start(httpd, [  
        {modules, [  
            mod_alias,  
            mod_auth,  
            mod_esi,  
            mod_actions,  
            mod_cgi,  
            mod_dir,  
            mod_get,  
            mod_head,  
            mod_log,  
            mod_disk_log  
        ]},  
  
        {port,8081},  
        {server_name,"helloworld"},  
        {server_root,"D://tmp"},  
        {document_root,"D://tmp/htdocs"},  
        {erl_script_alias, {"/erl", [helloworld]}},  
        {error_log, "error.log"},  
        {security_log, "security.log"},  
        {transfer_log, "transfer.log"},  
  
        {mime_types,[
```

```

        {"html","text/html"}, {"css","text/css"}, {"js","application/x-javascript"}
    ]}

    }).

service(SessionID, _Env, _Input) -> mod_esi:deliver(SessionID, [
    "Content-Type: text/html\r\n\r\n", "<html><body>Hello, World!</body></html>" ]).
```

Bước 2 – Chạy dòng code sau trong erlang shell.

```
c(helloworld).
```

Ta sẽ nhận được output.

```
{ok,helloworld}
```

Lệnh tiếp theo:

```
inets:start().
```

Có được output:

```
ok
```

Tiếp:

```
helloworld:start().
```

Output:

```
{ok,<0.50.0>}
```

Bước 3 – Bây giờ bạn đã có thể truy cập vào

url - http://localhost:8081/erl/hello_world:service

13.2 DRIVER

Thỉnh thoảng ta muốn chạy 1 chương trình viết bằng ngôn ngữ khác trong Erlang Runtime System. Trong trường hợp này, chương trình đó sẽ được viết như 1 thư viện chia sẻ mà được liên kết động vào Erlang Runtime System. Driver liên kết đóng vai trò như 1 port program và tuân theo chính xác giao thức của 1 port program.

Tạo 1 driver

Tạo 1 driver kết nối là cách hiệu quả nhất để giao tiếp mã của ngôn ngữ lập trình khác với Erlang, nhưng cũng là cách nguy hiểm nhất. Bất kỳ lỗi nào trong driver kết nối sẽ gây crash hệ thống Erlang.

Sau đây là ví dụ của 1 driver trong Erlang:

```
-module(helloworld).  
-export([start/0, stop/0]).  
-export([twice/1, sum/2]).  
  
start() ->  
    start("example1_drv" ).  
start(SharedLib) ->  
    case erl_ddll:load_driver(".", SharedLib) of  
    ok -> ok;  
    {error, already_loaded} -> ok;  
    _ -> exit({error, could_not_load_driver})  
    end,  
  
    spawn(fun() -> init(SharedLib) end).  
  
init(SharedLib) ->  
    register(example1_lid, self()),  
    Port = open_port({spawn, SharedLib}, []),  
    loop(Port).  
  
stop() ->  
    example1_lid ! stop.
```



```
twice(X) -> call_port({twice, X}).
sum(X,Y) -> call_port({sum, X, Y}). call_port(Msg) ->
    example1_lid ! {call, self(), Msg}, receive
        {example1_lid, Result} ->
            Result
    end.
```

LINKED-IN DRIVERS [223](#)

```
loop(Port) ->
receive
    {call, Caller, Msg} ->
        Port ! {self(), {command, encode(Msg)}}, receive
            {Port, {data, Data}} ->
                Caller ! {example1_lid, decode(Data)}
        end,
```

```
loop(Port);
stop -> Port !
    {self(), close},
receive
    {Port, closed} ->
        exit(normal)
end;

{'EXIT', Port, Reason} ->
    io:format("~p ~n" , [Reason]),
    exit(port_terminated)
end.
```

```
encode({twice, X}) -> [1, X];  
encode({sum, X, Y}) -> [2, X, Y]. decode([Int]) -> Int.
```

Cần chú ý rằng làm việc với driver rất phức tạp và luôn cần phải cẩn thận khi làm việc với chúng.

14. TỔNG KẾT

- **Ưu điểm**

Erlang – ngôn ngữ lập trình cấp thấp xét theo việc nó cho phép lập trình điều khiển những thứ mà thường do HĐH kiểm soát, như quản lý bộ nhớ, xử lý đồng thời, nạp những thay đổi vào chương trình khi đang chạy... rất hữu ích trong việc lập trình các thiết bị di động. Erlang được dùng trong nhiều hệ thống viễn thông lớn của Ericsson.

Chức năng chủ yếu của Erlang được sử dụng để viết các hệ thống phân tán.

Erlang là NNLT hướng concurrent và sử dụng cho việc lập trình đa lõi, và SMP (đa xử lý).

Các chương trình trên Erlang sẽ chạy nhanh hơn trên hệ thống đa lõi hoặc là SMP.

Erlang: được sử dụng nhiều trong telecom, để viết các hệ thống lớn, soft real-time, có khả năng chịu lỗi tốt, chúng có thể được chỉnh sửa mà không cần phải dừng ứng dụng lại.

Erlang tiếp cận concurrent programming theo hướng hoàn toàn khác nếu bạn đến từ Java hay C#. Đây là ngôn ngữ gây ảnh hưởng đến nhiều ngôn ngữ sau này như Scala.

Erlang hỗ trợ lập trình mạng khá tốt.

Ngôn ngữ sử dụng paradigm lập trình hàm (functional programming).

Đây là một ngôn ngữ được kiểm nghiệm thực tế trong hệ thống sản xuất công nghiệp quy mô lớn.

Các ứng dụng được viết lượng lượng dòng lệnh không quá nhiều.

- **Khuyết điểm**

Tốc độ: chậm hơn Java Server kể cả khi đã compile ra native code. Java ngày xưa chậm nhưng dần dần nhanh lên nhờ có nhiều người dùng. Erlang cũng đã cải thiện tốc độ rất nhiều nhưng chưa đủ. Một lý do chính là do vẫn quá ít người dùng.

Community: không rộng bằng các ngôn ngữ khác

Paradigm: functional programming đôi khi không phù hợp với một số vấn đề. Và người lập trình đôi khi cố chấp và nghĩ là nó không phù hợp.

Support library: không hùng hậu như C++ hay Java tuy nhiên nó lại có những library mà các ngôn ngữ khác không có, rất phù hợp khi xây dựng distributed system như gproc (quản lý process trên nhiều máy), yaws (scalable web server, vượt xa apache trong một vài benchmark liên quan đến realtime webapp), riak_core (library hỗ trợ xây dựng một distributed system mà không có node nào là chủ, tất cả đều ngang hàng và có thể thay thế nhau)...

Tóm lại, Erlang, với những điểm mạnh và yếu của nó rất đáng để quan tâm. Thiết bị kết nối internet càng ngày càng tăng cao. Processor đã đạt đến giới hạn về tốc độ và chuyển sang song song. Một ngôn ngữ và framework có thể đảm bảo horizontal scalability và khuyến khích suy nghĩ về giải pháp song song, phân tán là rất cần thiết. Erlang là một trong số đó. Nó không phải là một thứ đồ chơi do ai tạo ra cho vui (ngôn ngữ brainfuck chẳng hạn) mà là một sản phẩm đã được sử dụng thành công trong thực tế.

PHỤ LỤC A: TÀI LIỆU THAM KHẢO

Bảng sau tổng hợp tài liệu được tham khảo trong tài liệu này.

Tên tài liệu và phiên bản	Mô tả	Vị trí
Tìm hiểu về ngôn ngữ lập trình Erlang	Tiểu luận môn học của nhóm sinh viên nào đó	https://123doc.org/document/2403357-tim-hieu-ve-ngon-ngu-lap-trinh-erlang.htm
Erlang là gì?	..	https://linux4vietnam.wordpress.com/2011/08/14/erlang-la-gi/
Learn you some Erlang	..	http://learnyousomeerlang.com/content
Erlang Tutorial	..	https://www.tutorialspoint.com/erlang/index.htm
Erlang advisor	..	http://erladvisor.blogspot.com/2015/09/erlang-enum.html
		https://vi.scribd.com/doc/101504510/Erlang-Programming-Style-Guide

PHỤ LỤC B: THUẬT NGỮ

Bảng sau cho biết định nghĩa của những thuật ngữ được sử dụng trong tài liệu.

Thuật ngữ	Định nghĩa
<i>Pattern Matching</i>	Mẫu giống nhau ở hai vế dấu '='
<i>Guard Clause</i>	Câu lệnh điều kiện trên cùng của hàm trả về giá trị cho hàm ngay khi nó đúng
<i>Arity</i>	Số tham số truyền cho hàm
<i>CamelCase</i>	Kiểu viết code theo dạng lạc đà (u bước), viết hoa các chữ cái đầu từ