21 天学通 Erlang

December 12, 2015

目录

Part I

Erlang 基础

1 Erlang 使用指南

1.1 软件安装

Erlang/OTP 建议使用 17 或更高版本。编译器建议使用 Emacs。

1.1.1 Erlang

1.1.1.1 Fedora 21

运行 `yum install erlang`

安装完成后运行 `erl` 命令即可打开 Erlang Shell。

1.1.1.2 Debian Wheezy

运行 `apt-get -t wheezy-backports install erlang` [1]

安装完成后运行 `erl` 命令即可打开 Erlang Shell。

1.1.1.3 在 Windows 上安装

运行 `choco install erlang` [2]

安装后运行 `werl` 命令即可打开 Erlang Shell。

1.1.1.4 Erlang Shell

打开 Erlang Shell 后，第一行会显示 Erlang 的版本信息，第二行是空行，第三行会显示 Shell 的版本信息。因为每次打开总是会显示这三行，所以以后都

[1] 只有在 wheezy-backports 才有 Erlang/OTP 17

[2] 这需要先安装好 Chocolatey(`https://chocolatey.org/`)。也可以自行从 Erlang 官方网站 (`http://www.erlang.org/download.html`) 下载安装程序。假如下载很慢,不妨从 erlang-users.jp 的镜像 (`http://erlang-users.jp/`) 下载。

会忽略这三行。而 1> 表示这后面是 Shell 里的第 1 次输入。

```
Erlang/OTP 17 (erts-6.3) [async-threads:10]

Eshell V6.3  (abort with ^G)
1>
```

输入 io:format("Hello, world!~n"). ，按回车，会显示 Hello, world! 。

现在会看到 2> ，表示这后面是 Shell 里的第 2 次输入。

Erlang Shell

```
1> io:format("Hello, world!~n").
Hello, world!
ok
2>
```

输入 halt(). ，按回车后，就退出 Erlang Shell 了。

```
1> io:format("Hello, world!~n").
Hello, world!
ok
2> halt().
```

1.1.1.5 编译运行 **Erlang** 代码

编辑文件 hello.erl 。

Listing 1.1: install/hello.erl

```
1 -module(hello).
2
3 -export([world/0]).
4
5 world() -> "Hello, world!".
```

打开 Erlang Shell。

Erlang Shell

```
1> c(hello).
{ok,hello}
2> hello:world().
```

```
"Hello, world!"
3>
```

1.1.2 GNU Emacs

1.1.2.1 Fedora 21

运行 `yum install emacs`

1.1.2.2 Debian Wheezy

运行 `apt-get install emacs`

1.1.2.3 Windows

运行 `choco install emacs` [3]

用 Windows 须自行配置 Emacs。按 The Erlang mode for Emacs(`http://www.erlang.org/doc/apps/tools/erlang_mode_chapter.html`) 里 Setup on Windows 一节的说明设置 `.emacs` 文件。若不知道当前设置的 HOME 环境变量，可以进入 Erlang Shell 查看，运行 `os:getenv("HOME").`，假如结果是 `false`，那就是没有设置 HOME 环境变量。

1.2 初识 Erlang

1.2.1 表达式

Erlang 可以进行简单的算术运算。打开 Erlang Shell，输入算术表达式和 . 后按回车，就会显示运算结果。这个运算结果叫做表达式的值。

Erlang Shell

```
1> 1+1.
```

[3] 这需要先安装好 Chocolatey(`https://chocolatey.org/`)。也可以自行下载 Emacs。在`http://www.gnu.org/software/emacs/#Obtaining`上，找到 “nearby GNU mirror”，并点击进入。进入后，找到 windows 目录，以 24.4 版为例，选择 emacs-24.4-bin-i686-pc-mingw32.zip 。若下载很慢，不妨从中国科学技术大学开源软件镜像 `http://mirrors.ustc.edu.cn/gnu/emacs/windows/`下载。

```
2
2> 5-2.
3
3> 2*2.
4
4> 1+2*2.
5
5> (1+2)*2.
6
6>
```

1.2.2 函数

把一列数 $1, 2, 3, 4$ 称作一个数列。这个数列共有四项，其中，第 1 项为 1 ，第 2 项为 2 ，第 3 项为 3 ，第 4 项为 4 。
可以用 Erlang 的函数来表示这种对应关系。

Listing 1.2: erlang/seq1.erl

```
5 a(1) -> 1;
6 a(2) -> 2;
7 a(3) -> 3;
8 a(4) -> 4.
```

把这段代码称为函数 `a/1` 的定义，其中 `a` 是函数 `a/1` 的函数名，`1` 表示函数 `a/1` 只接受 `1` 个参数，这里这个参数对应数列的下标。一个函数的定义，可以由一个或者多个分句组成。多个分句之间需要用 `;` 隔开。函数定义需要以 `.` 结尾。`->` 左边是分句的头，`->` 右边是分句的正文。分句的正文可以由一个或者多个表达式组成，多个表达式之间需要用 `,` 隔开。把指定参数求函数的值称为调用。

1.2.3 模块

在调用这个函数之前，需要把这几行代码保存到一个文件里。Erlang 代码是按模块组织的，每个模块会有一个对应的文件。在文件最开头，还需要额外的几行代码来声明模块信息。

Listing 1.3: erlang/seq1.erl

```
1 -module(seq1).
2
```

```
3 -export([a/1]).
```

`-module(seq1).` 表示模块名是 `seq1` , 通常, 其对应源代码的文件是 `seq1.erl` 。而 `-export([a/1]).` 表示，函数 `a/1` 是公开的，也就是可以在其所在模块之外被调用。

打开 Erlang Shell，输入 `c(seq1).` ，按回车之后就会根据`seq1` 模块的源代码 (也就是文件 `seq1.erl` 的内容), 生成文件 `seq1.beam` 。这个过程叫做编译。有了 `seq1.beam` 这个文件，就可以直接在 Erlang Shell 里调用 `seq1` 模块里公开的函数了。需要指明模块名，即在函数名前加上模块名 `seq1` 和 `:` 。

Erlang Shell

```
1> c(seq1).
{ok,seq1}
2> seq1:a(1).
1
3> seq1:a(2).
2
4> seq1:a(3).
3
5> seq1:a(4).
4
6>
```

以后没有特殊说明，都假设用到的模块已经实现编译好了。

1.2.4 模式

回到函数 `a/1` 的定义

Listing 1.4: erlang/seq1.erl

```
5 a(1) -> 1;
6 a(2) -> 2;
7 a(3) -> 3;
8 a(4) -> 4.
```

Erlang 在调用函数时，会按顺序逐个尝试，直到找到第一个能和传入参数匹配的分句，对分句正文的表达式逐个求值，并以最后一个表达式的值，作为函数调用表达式的值。假如没找到，程序就会出错。

Erlang Shell

```
1> seq1:a(5).
** exception error: no function clause matching seq1:a
(5) (seq1.erl, line 5)
2>
```

因为是按顺序的，所以即便有两个能匹配的，Erlang 也只会用第一个。不妨在下面增加一个分句。

Listing 1.5: erlang/seq2.erl

```
5 a(1) -> 1;
6 a(2) -> 2;
7 a(3) -> 3;
8 a(4) -> 4;
9 a(4) -> 5.
```

可以看到结果仍然是 4。甚至在编译时 Erlang 都警告说，第 9 行没有机会匹配。

Erlang Shell

```
1> c(seq2).
seq2.erl:9: Warning: this clause cannot match because a
previous clause at line 8 always matches
{ok,seq2}
2> seq2:a(4).
4
3>
```

有一个数列 $1, 1, 1, \ldots$，这个数列有无穷项，且从第 1 项开始每一项都是 1，可以用以下 Erlang 代码来表示这个数列。

Listing 1.6: erlang/seq3.erl

```
5 b(_) -> 1.
```

_ 表示无论传入的这个参数是什么，到这里都会匹配。

Erlang Shell

```
1> seq3:b(1).
1
2> seq3:b(2).
1
3> seq3:b(3).
1
4> seq3:b(4).
1
5>
```

有一个数列 $1, 2, 3, \ldots$ ，这个数列有无穷项，且从第 1 项开始每一项的值都等于该项下标，可以用以下 Erlang 代码来表示这个数列。

Listing 1.7: erlang/seq4.erl

```
5 c(N) -> N.
```

N 是一个变量。变量的首字母都是大写的。在这里，N 和 _ 的作用类似，无论传入的这个参数是什么，到这里都会匹配。不同的是，匹配后，在分句的正文里，N 对应的值就是当前传入参数的值。

Erlang Shell

```
1> seq4:c(1).
1
2> seq4:c(2).
2
3> seq4:c(3).
3
4> seq4:c(4).
4
5>
```

回到数列 b 。虽然，对于数列里所有下标，函数都能给出正确的结果，但是，对于其他整数，函数却没有出错，这不是期望的结果。

Erlang Shell

```
1> seq3:b(0).
1
2> seq3:b(-1).
1
3>
```

函数一个分句的头部，还可以有一个或多个用, 隔开的 guard，用来限制匹配的参数的取值范围。

Listing 1.8: erlang/seq5.erl

```
5 b(N) when N >= 1 -> 1.
```

加上了 guard 之后就会出错了。

Erlang Shell

```
1> seq5:b(0).
** exception error: no function clause matching seq5:b
(0) (seq5.erl, line 5)
2>
```

数列 c，也需要限制取值范围。

Listing 1.9: erlang/seq6.erl

```
5 c(N) when N >= 1 -> N.
```

同样会出错了。

Erlang Shell

```
1> seq6:c(0).
** exception error: no function clause matching seq6:c
(0) (seq6.erl, line 5)
2>
```

Erlang 还可以在表达式里匹配，其中一种是 `case` 表达式。数列 a 也可以用下面这样的 Erlang 代码表示

Listing 1.10: erlang/seq7.erl

```
5  a(N) ->
6      case N of
7          1 -> 1;
8          2 -> 2;
9          3 -> 3;
10         4 -> 4
11     end.
```

结果和之前的写法是一样的

Erlang Shell

```
1> seq7:a(1).
1
2> seq7:a(2).
2
3> seq7:a(3).
3
4> seq7:a(4).
4
5>
```

另一种表达式，= 右边是一个表达式，相当于是传入的参数，= 左边相当于函数定义里某个分句头。假如不匹配，那么就会出错。匹配的话，这个表达式的值，就是 = 右边表达式的值。

Erlang Shell

```
1> 1 = 1.
1
2> 1 = 1+1.
** exception error: no match of right hand side value 2
3>
```

当 case 表达式里只有一个分句，且这个分句没有使用 guard。那么就可以由这种表达式代替。我们可以用这种表达式来写测试 [1]。= 左边写期望的值，= 右边写要测试的表达式，因为不匹配就会出错，这样就知道哪里不符合期望了。

Listing 1.11: erlang/seq7.erl

```
13 test() ->
14     1 = a(1),
15     2 = a(2),
16     3 = a(3),
17     4 = a(4),
18     ok.
```

可以在一个函数里，列出所有的测试，最后一个表达式写 `ok` ，这样没有出错就会看到 ok。而不需要在 Erlang Shell 里重复输入这些表达式了。

Erlang Shell

```
1> seq7:test().
ok
2>
```

练习 **1.1 * fac:** 阶乘数列 f_n 的定义如下

$$f_n = \begin{cases} 1 & , n = 1 \\ n \cdot f_{n-1} & , n > 1 \end{cases}$$

其前 5 项分别为 $1, 2, 6, 24, 120$

在 `fac` 模块里，定义一个公开函数 `f/1` 来表示这个数列。

每个练习都会有一个名字，紧跟在练习编号以及难度后面的就是练习的名字了。比如现在这个练习的名字是 fac。

在每个章节的目录下，都有一个 `exercise` 模块。里面定义了 `check/1` 函数，可以对练习的代码进行一些基本的检查。请把练习的名字作为参数去调用这个函数。

假如你的代码没问题，结果就会是 `ok` 。

Erlang Shell

```
1> exercise:check(fac).
```

```
ok
2>
```

`check/1` 函数里对应的定义如下

Listing 1.12: erlang/exercise.erl

```
5  check(fac) ->
6      1 = fac:f(1),
7      2 = fac:f(2),
8      6 = fac:f(3),
9      24 = fac:f(4),
10     120 = fac:f(5),
11     ok;
```

以后没有特殊说明，就表示可以用 `check/1` 来检查。

练习 **1.2 * fib:** 在 `fac` 模块里，定义一个公开函数 `f/1` 来表示 Fibonacci 数列。

Fibonacci 数列 f_n 的定义如下

$$f_n = \begin{cases} 1 & ,n = 1 \\ 1 & ,n = 2 \\ f_{n-2} + f_{n-1} & ,n > 2 \end{cases}$$

其前 5 项分别为 $1, 1, 2, 3, 5$

Listing 1.14: erlang/exercise.erl

```
12 check(fib) ->
13     1 = fib:f(1),
14     1 = fib:f(2),
15     2 = fib:f(3),
16     3 = fib:f(4),
17     5 = fib:f(5),
18     ok.
```

1.2.5 数据类型

ok 是什么？Erlang 不仅能进行整数运算，还有别的数据类型。ok 就是一个类型为 atom() 的数据。可以简单认为一个 atom() 就是一连串小写字母。当且仅当每个位置上的字母都相同的时候，才认为两个 atom() 相同。

Erlang Shell

```
1> a = a.
a
2> a = b.
** exception error: no match of right hand side value b
3> a = aa.
** exception error: no match of right hand side value aa
4>
```

那 case 和 end 呢？这两个是用来表示这中间是一个 case 表达式，都是 Erlang 的保留字。实际上，atom() 必须在两个 ' 之间输入。只不过，假如一个 atom() 是小写字母开头的，不包含特殊字符，而且不是保留字，输入时就可以省略前后两个 ' 。

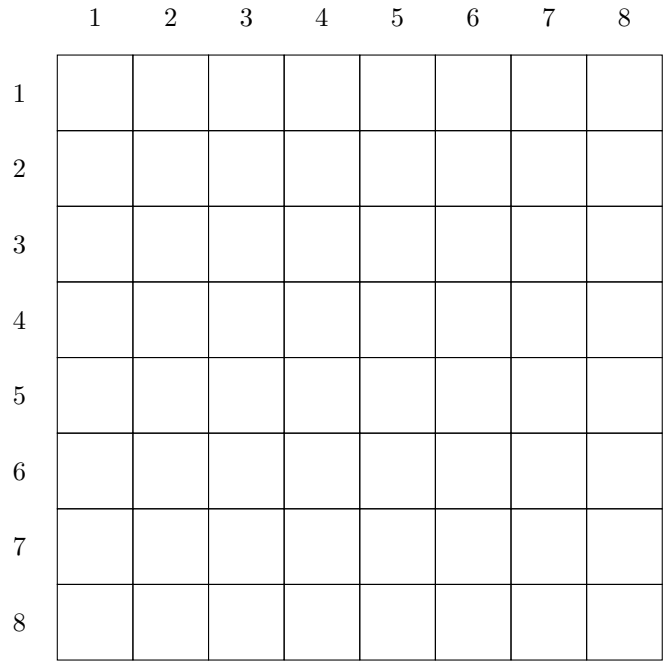
Erlang Shell

```
1> case.
* 1: syntax error before: '.'
1> 'case'.
'case'
2> a = 'a'.
a
3>
```

假设存在一个 8×8 的棋盘，横坐标是 1 到 8，纵坐标也是 1 到 8。

可以用一个 tuple() 表示这个棋盘上一个格子的坐标。比如，{1,2} 表示横坐标为 1，纵坐标为 2 的格子。tuple() 可以有零个、一个或者多个元素。只有当每个元素都相同的时候，才认为是相同的 tuple() 。

Erlang Shell



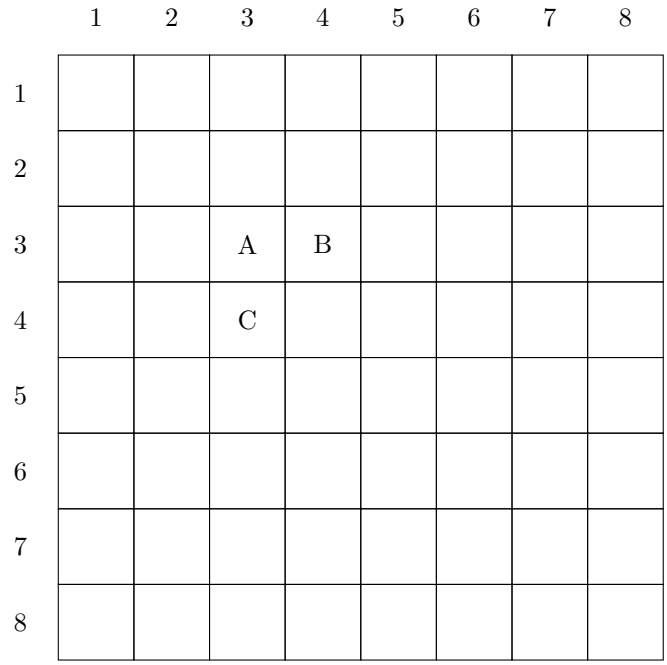

```
1> {} = {}.
{}
2> {1} = {}.
** exception error: no match of right hand side value {}
3> {1} = {1}.
{1}
4> {1} = {1,1}.
** exception error: no match of right hand side value
{1,1}
5> {1,1} = {1,{1}}.
** exception error: no match of right hand side value
{1,{1}}
6>
```

可以用 `up`, `down`, `left`, `right` 这四个 `atom()` 来表示上下左右四个方向。定义 `move/2` 函数来根据当前坐标计算棋子移动一格之后的坐标。

如上图所示，A 往右移一格就到 B，B 往左边移一格就到 A，A 往下移一格就到 C，C 往上移一格就到 A。

Listing 1.16: erlang/board1.erl

```
18 test(move) ->
19     A = {3,3},
20     B = {4,3},
```




```
21      C = {3,4},
22      B = move(right, A),
23      A = move(left, B),
24      C = move(down, A),
25      A = move(up, C).
```

分别用 X, Y 表示横坐标和纵坐标，往左就是把横坐标减一，往右就是把横坐标加一，往上就是把纵坐标减一，往下就是把纵坐标加一。

Listing 1.17: erlang/board1.erl

```
5  move(left, {X,Y})
6    when X > 1, X =< 8 ->
7      {X-1, Y};
8  move(right, {X,Y})
9    when X >= 1, X < 8 ->
10     {X+1, Y};
11 move(up, {X,Y})
12   when Y > 1, Y =< 8 ->
13     {X, Y-1};
14 move(down, {X,Y})
15   when Y >= 1, Y < 8 ->
16     {X, Y+1}.
```

练习 **1.3 * board3:** 上面的例子并没有定义比如在最左边往左移动的情况。

现在规定，在最左边往左移动的结果是待在原地。

练习 **1.4 * board4:** 现在把规定改成，在最左边往左移动的结果是出现在最右边。

练习 **1.5 * board5:** move_n

用一个 `list()` 来表示多步移动的方向。`list()` 和 `tuple()` 相似，都是多个元素按顺序组合起来的。`tuple()` 适合表示和同一个事物相关的多个数据，比如一个格子的横坐标和纵坐标。而 `list()` 适合表示很多个同类的数据，比如多个方向，多个坐标。

Listing 1.18: erlang/board2.erl

```
31 test(moves) ->
32     {4,1} = moves([right,right,right], {1,1}),
33     {1,4} = moves([down,down,down], {1,1}).
```

用 `list()` 是因为，`list()` 可以方便地取出前几个元素，比如，`[A1,A2,A3|B]` 就表示一个 `list()` 的前 3 个元素，分别为 `A1`, `A2`, `A3`。把剩余的元素按顺序组合起来的 `list()` 和 `B` 相同。

Erlang Shell

```
1> [a, b] = [a|[b]].
[a,b]
2> [a, b, c] = [a,b|[c]].
[a,b,c]
3> [a] = [a|[]].
[a]
4> [a,b|_] = [a,b,c].
[a,b,c]
5> [a,b|_] = [a,b,c,d].
[a,b,c,d]
6>
```

若 `list()` 为空，说明不需要再移动了，那么起始坐标就是最终坐标。否则，取出第一个方向，把按此移动一步后的坐标作为新的起始坐标，再来计算。

Listing 1.19: erlang/board2.erl

```
18 moves([], From) ->
19     From;
20 moves([H|T], From) ->
21     moves(T, move(H, From)).
```

练习 **1.6 * length:** 定义 `length/1` 函数，求一个 `list()` 的元素个数

练习 **1.7 * sum:**

练习 **1.8 * member:**

练习 **1.9 * append:**

练习 **1.10 * zip:**

1.3 参考

[1] Joe Armstrong. Micro lightweight unit testing.

`http://armstrongonsoftware.blogspot.com/2009/01/micro-lightweight-unit-testing.html`, January 2009.

2 解释 Erlang 程序

> You think you know when you learn, are more sure when you can write, even more when you can teach, but certain when you can program.
>
> Alan J. Perlis [1]

2.1 简单表达式求值

Erlang 解释器在解释 Erlang 表达式时，会先将表达式转换成另一种形式。这种形式值求起来更方便。现在可以用 `parse_util:expr/1` 来把一个在 Erlang Shell 中输入的表达式转换成这种形式。

整数求值

Erlang Shell

```
1> parse_util:expr("4.").
{integer,1,4}
2>
```

Listing 2.1: expression/eval_integer.erl

```
9  eval_string(S, Bindings) ->
10     eval(parse_util:expr(S), Bindings).
```

Listing 2.2: expression/eval_integer.erl

```
12 test(eval_integer) ->
```

```
13      {ok, 1, []} = eval_string("1.", []),
14      {ok, 2, []} = eval_string("2.", []),
15      {ok, 3, []} = eval_string("3.", []),
16      ok.
```

Listing 2.3: expression/eval_integer.erl

```
6   eval({integer, _, Value}, Bindings) ->
7       {ok, Value, Bindings}.
```

`atom()` 求值

Erlang Shell

```
1> parse_util:expr("a.").
{atom,1,a}
2>
```

Listing 2.4: expression/eval_atom.erl

```
19  test(eval_atom) ->
20      {ok, a, []} = eval_string("a.", []),
21      {ok, b, []} = eval_string("b.", []),
22      {ok, c, []} = eval_string("c.", []),
23      ok.
```

Listing 2.5: expression/eval_atom.erl

```
8   eval({atom, _, Value}, Bindings) ->
9       {ok, Value, Bindings}.
```

变量求值

Erlang Shell

```
1> parse_util:expr("X.").
{var,1,'X'}
2>
```

Listing 2.6: expression/eval_var.erl

```
31 test(eval_var) ->
32     {ok, 1, [{'X', 1}]} = eval_string("X.", [{'X', 1}]),
33     {error, {unbound, 'X'}} = eval_string("X.", []),
34     ok.
```

Listing 2.7: expression/eval_var.erl

```
10 eval({var, _, Var}, Bindings) ->
11     case bindings:lookup(Var, Bindings) of
12         {ok, Value} ->
13             {ok, Value, Bindings};
14         none ->
15             {error, {unbound, Var}}
16     end.
```

`list()` 求值

Erlang Shell

```
1> parse_util:expr("[].").
{nil,1}
2> parse_util:expr("[1].").
{cons,1,{integer,1,1},{nil,1}}
3> parse_util:expr("[1,2].").
{cons,1,{integer,1,1},{cons,1,{integer,1,2},{nil,1}}}
4>
```

Listing 2.8: expression/eval_list.erl

```
49 test(eval_list) ->
50     {ok, [], []} = eval_string("[].", []),
51     {ok, [1,2], []} = eval_string("[1,2].", []),
52     {ok, [1,2], [{'X', 1}, {'Y', 2}]} = eval_string("[X,
Y].", [{'X', 1}, {'Y', 2}]),
53     ok.
```

Listing 2.9: expression/eval_list.erl

```
17 eval({nil, _}, Bindings) ->
18     {ok, [], Bindings};
```

```
19 eval({cons, _, H, T}, Bindings) ->
20     case eval(H, Bindings) of
21         {ok, H1, Bindings1} ->
22             case eval(T, Bindings1) of
23                 {ok, T1, Bindings2} ->
24                     {ok, [H1|T1], Bindings2};
25                 Error ->
26                     Error
27             end;
28         Error ->
29             Error
30     end.
```

tuple() 求值

Erlang Shell

```
1> parse_util:expr("{1,2}.").
{tuple,1,[{integer,1,1},{integer,1,2}]}
2>
```

Listing 2.10: expression/eval_tuple.erl

```
76 test(eval_tuple) ->
77     {ok, {tuple, [1,2]}, []} = eval_string("{1,2}.", [])
,
78     {ok, {tuple, [1,2]}, [{'X', 1}, {'Y', 2}]} =
eval_string("{X,Y}.", [{'X', 1}, {'Y', 2}]),
79     ok.
```

Listing 2.11: expression/eval_tuple.erl

```
31 eval({tuple, _, Elements}, Bindings) ->
32     case eval_elements(Elements, Bindings) of
33         {ok, Value, Bindings1} ->
34             {ok, {tuple, Value}, Bindings1};
35         Error ->
36             Error
37     end.
```

Listing 2.12: expression/eval_tuple.erl

```
39 eval_elements([], Bindings) ->
```

```
40      {ok, [], Bindings};
41  eval_elements([H|T], Bindings) ->
42      case eval(H, Bindings) of
43          {ok, H1, Bindings1} ->
44              case eval_elements(T, Bindings1) of
45                  {ok, T1, Bindings2} ->
46                      {ok, [H1|T1], Bindings2};
47                  Error ->
48                      Error
49              end;
50          Error ->
51              Error
52      end.
```

匹配整数

Listing 2.13: expression/match_integer.erl

```
38  eval({match, _, A, B}, Bindings) ->
39      case eval(B, Bindings) of
40          {ok, Value, Bindings1} ->
41              match(A, Value, Bindings1);
42          Error ->
43              Error
44      end.
```

Listing 2.14: expression/match_integer.erl

```
92  test(match_integer) ->
93      {ok, 1, []} = eval_string("1 = 1.", []),
94      {error, {mismatch, 2}} = eval_string("1 = 2.", []),
95      ok.
```

Listing 2.15: expression/match_integer.erl

```
61  match({integer, _, Value}, Value, Bindings) ->
62      {ok, Value, Bindings};
63  match({integer, _, _}, Value, _) ->
64      {error, {mismatch, Value}}.
```

匹配atom()

Listing 2.16: expression/match_atom.erl

```
100 test(match_atom) ->
101     {ok, a, []} = eval_string("a = a.", []),
102     {error, {mismatch, b}} = eval_string("a = b.", []),
103     ok.
```

Listing 2.17: expression/match_atom.erl

```
65 match({atom, _, Value}, Value, Bindings) ->
66     {ok, Value, Bindings};
67 match({atom, _, _}, Value, _) ->
68     {error, {mismatch, Value}}.
```

匹配变量

Listing 2.18: expression/match_var.erl

```
113 test(match_var) ->
114     {ok, a, [{'X', a}]} = eval_string("X = a.", []),
115     {ok, a, [{'X', a}]} = eval_string("X = a.", [{'X', a
}]),
116     {error, {mismatch, b}} = eval_string("X = a.", [{'X
', b}]),
117     ok.
```

Listing 2.19: expression/match_var.erl

```
69 match({var, _, Var}, Value, Bindings) ->
70     case bindings:lookup(Var, Bindings) of
71         {ok, Value} ->
72             {ok, Value, Bindings};
73         {ok, Value2} ->
74             {error, {mismatch, Value2}};
75         none ->
76             {ok, Value, [{Var,Value}|Bindings]}
77     end.
```

匹配list()

Listing 2.20: expression/match_list.erl

```
140 test(match_list) ->
141     {ok, [], []} = eval_string("[] = [].", []),
142     {error, {mismatch, 1}} = eval_string("[] = 1.", []),
143     {ok, [1,2,3], []} = eval_string("[1,2,3] =
[1,2,3].", []),
```

```
144     {error, {mismatch, [1,2]}} = eval_string("[1,2,3] =
    [1,2].", []),
145     ok.
```

Listing 2.21: expression/match_list.erl

```
78  match({nil, _}, [], Bindings) ->
79      {ok, [], Bindings};
80  match({nil, _}, Value, _) ->
81      {error, {mismatch, Value}};
82  match({cons, _, H, T}, [VH|VT], Bindings) ->
83      case match(H, VH, Bindings) of
84          {ok, H1, Bindings1} ->
85              case match(T, VT, Bindings1) of
86                  {ok, T1, Bindings2} ->
87                      {ok, [H1|T1], Bindings2};
88                  {error, {mismatch, _}} ->
89                      {error, {mismatch, [VH|VT]}};
90                  Error ->
91                      Error
92              end;
93          {error, {mismatch, _}} ->
94              {error, {mismatch, [VH|VT]}};
95          Error ->
96              Error
97      end;
98  match({cons, _, _, _}, Value, _) ->
99      {error, {mismatch, Value}}.
```

匹配tuple()

Listing 2.22: expression/match_tuple.erl

```
178 test(match_tuple) ->
179     {ok, {tuple, []}, []} = eval_string("{} = {}.", []),
180     {error, {mismatch, 1}} = eval_string("{} = 1.", []),
181     {ok, {tuple, [1,2,3]}, []} = eval_string("{1,2,3} =
    {1,2,3}.", []),
182     {error, {mismatch, {tuple, [1,2]}}} = eval_string
    ("{1,2,3} = {1,2}.", []),
183     ok.
```

Listing 2.23: expression/match_tuple.erl

```
100 match({tuple, _, Elements}, {tuple, Values}, Bindings)
    ->
```

```
101     case match_elements(Elements, Values, Bindings) of
102         {ok, Value, Bindings1} ->
103             {ok, {tuple, Value}, Bindings1};
104         {error, {mismatch, _}} ->
105             {error, {mismatch, {tuple, Values}}};
106         Error ->
107             Error
108     end;
109 match({tuple, _, _}, Value, _) ->
110     {error, {mismatch, Value}}.
```

Listing 2.24: expression/match_tuple.erl

```
112 match_elements([], [], Bindings) ->
113     {ok, [], Bindings};
114 match_elements([H|T], [VH|VT], Bindings) ->
115     case match(H, VH, Bindings) of
116         {ok, H1, Bindings1} ->
117             case match_elements(T, VT, Bindings1) of
118                 {ok, T1, Bindings2} ->
119                     {ok, [H1|T1], Bindings2};
120                 {error, {mismatch, _}} ->
121                     {error, {mismatch, [VH|VT]}};
122                 Error ->
123                     Error
124             end;
125         {error, {mismatch, _}} ->
126             {error, {mismatch, [VH|VT]}};
127         Error ->
128             Error
129     end;
130 match_elements(_, Value, _) ->
131     {error, {mismatch, Value}}.
```

2.2 参考

[1] Alan J. Perlis. Epigrams on programming. *ACM SIGPLAN Notices*, pages 7–13, 1982.

Part II

附录

A 习题解答

练习 **1.1:**

Listing 1.13: erlang/fac.erl

```
-module(fac).

-export([f/1]).

f(1) ->
    1;
f(N)
  when N > 1->
    N * f(N-1).
```

练习 **1.2:**

Listing 1.15: erlang/fib.erl

```
-module(fib).

-export([f/1]).

f(1) ->
    1;
f(2) ->
    1;
f(N)
  when N > 2->
    f(N-2) + f(N-1).
```