

21 天学通 Erlang

April 12, 2015

目录

I Erlang 基础	2
1 Erlang 使用指南	3
1.1 软件安装	3
1.2 初识 Erlang	5
1.3 参考	18
2 解释器原理	19
2.1 函数	19
2.2 模式	31
2.3 参考	31
II 附录	32
A 习题解答	33

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。

1. ERLANG 使用指南

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。

1. ERLANG 使用指南

```
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

把一系列数 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.
```

1. ERLANG 使用指南

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,...，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 $1, 2, 3, \dots$ ，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. ERLANG 使用指南

```
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

1. ERLANG 使用指南

```
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	3	4	5	6	7	8
1								
2								
3								
4								
5								
6								
7								
8								

```

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},

```

1. ERLANG 使用指南

	1	2	3	4	5	6	7	8
1								
2								
3			A	B				
4			C					
5								
6								
7								
8								

```
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。

1. ERLANG 使用指南

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 解释器原理

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 函数

函数是 Erlang 里最基本的概念之一。其实，算术运算也会先转换成函数调用再计算。比如，初识 Erlang 中最开始的例子可以写成下面这样。

Erlang Shell

```
1> erlang:'+'(1,1).  
2  
2> erlang:'-'(5,2).  
3  
3> erlang:'*'(2,2).  
4  
4> erlang:'+'(1,erlang:'*'(2,2)).  
5  
5> erlang:'*'(erlang:'+'(1,2),2).  
6  
6>
```

现在就来看一下怎么解释函数。

2.1.1 Environment

函数中定义的变量，在调用的过程中，会有对应的值。比如，

2. 解释器原理

把这种对应关系称为 Environment。要解释函数，就要能表示 Environment。方便起见，这里先用小写字母开头的 `atom()` 来代表变量名。

练习 2.1 ** environ: 在 `environ` 模块里定义四个函数，`empty/0`, `is_bound/2`, `bind/3`, `get_value/2`。调用 `empty/0` 可以得到一个空的 Environment。`is_bound/2` 用来检查变量名是否在 Environment 里有定义。`bind/3` 用来把变量名和值关联起来。`get_value/2` 用来获取变量的值。

Listing 2.1: `function/exercise.erl`

```
6 check(environ) ->
7     Env = environ:empty(),
8     false = environ:is_bound(a, Env),
9     Env1 = environ:bind(a, b, Env),
10    true = environ:is_bound(a, Env1),
11    b = environ:get_value(a, Env1),
12    ok;
```

现在用一个特殊的 `list()` 来表示 Environment。这个 `list()` 的每个元素都是一个特殊的 `tuple()`。这个 `tuple()` 的第一个元素用来表示变量名，第二个元素用来表示变量的值。定义 `subst/2`，用来从这样的 Environment 里取出变量的值。

Listing 2.3: `function/func1.erl`

```
7 subst(_, []) ->
8     none;
9 subst(K, [{K, V}|_]) ->
10    {ok, V};
11 subst(K, [_|T]) ->
12    subst(K, T).
```

2.1.2 S 表达式

定义 S 表达式 [2]

- 若 `Expr` 是一个 `atom()`，`Expr` 是一个 S 表达式。
- 若 `Expr` 是一个 `list()`，且 `Expr` 里的所有元素都是 S 表达式，那么 `Expr` 是一个 S 表达式。

练习 2.2 * `sexpr`: 在 `sexpr` 模块里定义 `is_sexpr/1` 函数, 判断是否是 S 表达式。

Listing 2.4: function/exercise.erl

```

13 check(sexpr) ->
14     true = sexpr:is_sexpr(a),
15     false = sexpr:is_sexpr(1),
16     true = sexpr:is_sexpr([]),
17     true = sexpr:is_sexpr([a]),
18     true = sexpr:is_sexpr([a, b, c]),
19     false = sexpr:is_sexpr([1]),
20     false = sexpr:is_sexpr([a, a]),
21     ok;

```

若 S 表达式是一个 `atom()`, 将其视为变量名, 其值可以在 `Environment` 中找到。若一个 S 表达式是一个 `list()`, 我们把它当作是一个函数调用, 其中, 这个 `list()` 第一个元素的值, 是被调用的函数, 剩余的元素就是传入的参数。这种求值的规则可以用 `apply/2` 来表示。`apply/2` 定义如下

Listing 2.6: function/func1.erl

```

15 apply([H|T], Env) ->
16     {Fun, Env1} = apply(H, Env),
17     call(Fun, T, Env1);
18 apply(Expr, Env) ->
19     true = is_atom(Expr),
20     {ok, Value} = subst(Expr, Env),
21     {Value, Env}.

```

注意, Erlang 编译时默认导入的函数中, 也有叫 `apply/2` 的。可以在文件开头取消。

Listing 2.7: function/func1.erl

```

2 -compile({no_auto_import, [apply/2]}).

```

因为现在普通的数据和函数都是从同一个 `Environment` 里取出来的, 为了区分, 用 `{data, X}` 表示普通数据, 用 `{fn, X}` 表示函数。光有一个空的 `Environment` 和求值规则, 啥也做不了。接下来就来定义一些基本函数。

2. 解释器原理

2.1.3 quote

假如，我们想让一个表达式的求值结果是一个 `atom()`，比如 `a`。因为在求值过程中，一个 `atom()` 会当作变量名，直接对 `a` 求值得到的会是变量 `a` 的值，而不是 `a`。为了能得到 `a`，需要定义 `quote` 函数。这个函数只接受一个参数，调用这个函数的结果就是传入的参数。

Listing 2.8: function/func1.erl

```
36 test(quote) ->
37     {{data, a}, _} =
38         apply([quote, a], new_env()),
39     {{data, [a,b,c]}, _} =
40         apply([quote, [a,b,c]], new_env()).
```

先定义 `new_env/0`，用来得到一开始的 Environment。

Listing 2.9: function/func1.erl

```
28 new_env() ->
29     [{quote, {fn, quote}}].
```

`quote` 的定义很简单。

Listing 2.10: function/func1.erl

```
24 call({fn, quote}, [X], Env) ->
25     {{data, X}, Env}.
```

S 表达式的 `[quote, a]` 作用相当于 Erlang 里的 `a`。这已经不是一般的函数了，不过这里仍然把它称作函数。在 Erlang 里调用一个函数前，会先对各个参数的表达式求值。定义 `id/1` 函数

Listing 2.11: function/equiv.erl

```
6 id(X) ->
7     X.
```

不难看出

Listing 2.12: function/equiv.erl

```
15 test(id) ->
```

```

16     a = id(a),
17     a = id(id(a));

```

而在 S 表达式里,在调用函数前,没有先对表达式求值。所以, `[quote, [quote, a]]` 的求值结果会是 `[quote, a]`。

Listing 2.13: function/func2.erl

```

41     {{data, [quote, a]}, _} =
42     apply([quote, [quote, a]], new_env()).

```

练习 2.3 * id: 定义 S 表达式函数 `id`, 使其和 Erlang 里 `id/1` 有相同的作用

Listing 2.14: function/exercise.erl

```

22 check(id) ->
23     {{data, a}, _} =
24     id:apply([id, [quote, a]], id:new_env()),
25     {{data, a}, _} =
26     id:apply([id, [id, [quote, a]]], id:new_env()),
27     ok;

```

2.1.4 label

定义一个 S 表达式函数 `label`, 用来指定变量的值。现在不考虑模式, 只要做到和 Erlang 里 `X = a` 类似就行了。因为要检查后面的表达式里变量的值, 在定义 `label` 之前, 先定义一个辅助函数 `eval_list/2`, 用来对一连串 S 表达式分别求值。

Listing 2.17: function/func3.erl

```

36 eval_list([], Env) ->
37     {[], Env};
38 eval_list([H|T], Env) ->
39     {VH, Env1} = apply(H, Env),
40     {VT, Env2} = eval_list(T, Env1),
41     {[VH|VT], Env2}.

```

接着来定义 `label`

2. 解释器原理

Listing 2.18: function/func3.erl

```
26 call({fn, label}, [X,Y], Env) ->
27     {Y1, Env1} = apply(Y, Env),
28     {Y1, [{X, Y1}|Env1]}.
```

这样定义的问题是，前一次指定的值会被后一次指定的值覆盖

Listing 2.19: function/func3.erl

```
55 test(label) ->
56     [{[data, a], [data, a]}, _] =
57         eval_list([label, x, [quote, a]],
58                 x
59                 ], new_env()),
60     [{[data, b], [data, b]}, _] =
61         eval_list([label, x, [quote, b]],
62                 x
63                 ], new_env()),
64     [{[data, a], [data, b], [data, b]}, _] =
65         eval_list([label, x, [quote, a]],
66                 [label, x, [quote, b]],
67                 x
68                 ], new_env()).
```

而在 Erlang 里，这样会导致出错

Erlang Shell

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

不过，这实际上是模式匹配的作用。这里就先保留当前的定义，小心一点不重复指定同一个变量的值就好了。

2.1.5 car, cdr, cons

因为 S 表达式由 `list()` 和 `atom()` 组成，所以肯定要定义一些 `list()` 相关的函数。`car`, `cdr`, `cons` 分别对应 Erlang 里的 `car/1`, `cdr/1` 和 `cons/2`。

Listing 2.20: function/equiv.erl

```

10 car([H|_]) -> H.
11 cdr([_|T]) -> T.
12 cons(H, T) -> [H|T].

```

car/1 用来取出一个 list() 的第一个元素

Listing 2.21: function/equiv.erl

```

18 test(car) ->
19     a = car([a]),
20     a = car([a,b]);

```

cdr/1 用来取出一个 list() 的除第一个元素以外的所有元素

Listing 2.22: function/equiv.erl

```

21 test(cdr) ->
22     [] = cdr([a]),
23     [b] = cdr([a,b]);

```

cons/2 用来在一个 list() 之前加上一个元素。

Listing 2.23: function/equiv.erl

```

24 test(cons) ->
25     [a] = cons(a, []),
26     [a,b] = cons(a, [b]).

```

定义 car, cdr, cons 。

Listing 2.24: function/func4.erl

```

29 call({fn, car}, [X], Env) ->
30     [{data, [H|_]}, Env1] = apply(X, Env),
31     [{data, H}, Env1];
32 call({fn, cdr}, [X], Env) ->
33     [{data, [_|T]}, Env1] = apply(X, Env),
34     [{data, T}, Env1];
35 call({fn, cons}, [X,Y], Env) ->
36     [{data,X1}, {data,Y1}], Env1} =
37     eval_list([X,Y], Env),
38     [{data, [X1|Y1]}, Env1].

```

2. 解释器原理

可以看到和 Erlang 里对应的函数作用是一样的。

Listing 2.25: function/func4.erl

```
82 test(list) ->
83     {{data, a}, _} =
84         apply([car, [quote, [a]]], new_env()),
85     {{data, a}, _} =
86         apply([car, [quote, [a,b]]], new_env()),
87     {{data, []}, _} =
88         apply([cdr, [quote, [a]]], new_env()),
89     {{data, [b]}, _} =
90         apply([cdr, [quote, [a,b]]], new_env()),
91     {{data, [a]}, _} =
92         apply([cons, [quote, a], [quote, []]],
93             new_env()),
94     {{data, [a,b]}, _} =
95         apply([cons, [quote, a], [quote, [b]]],
96             new_env()).
```

2.1.6 cond

Erlang 里，一个函数可以有多个分句。这也可以用 case 表达式来表示。因为不考虑模式，所以这里定义的 cond 就相当于只剩下 guard 的 case 表达式。函数 cond 只接受一个 list() 类型的参数，其中每个元素都是一个有两个元素的 list()。逐个对其第一个元素求值，直到找到一个 list()，对其第一个元素求值的结果是 true，那么调用 cond 函数的结果就是对其第二个元素求值的结果。

Listing 2.26: function/func5.erl

```
106 test('cond') ->
107     {{data, a}, _} =
108         apply(['cond',
109             [[quote, true], [quote, a]],
110             [[quote, false], [quote, b]],
111             [[quote, false], [quote, c]]
112             ], new_env()),
113     {{data, b}, _} =
114         apply(['cond',
115             [[quote, false], [quote, a]],
116             [[quote, true], [quote, b]],
```

```

117         [[quote, false], [quote, c]]
118     ], new_env()),
119     {{data, c}, _} =
120         apply(['cond',
121             [[quote, false], [quote, a]],
122             [[quote, false], [quote, b]],
123             [[quote, true], [quote, c]]
124         ], new_env()).

```

以下是 `cond` 的定义

Listing 2.27: function/func5.erl

```

39 call({fn, 'cond'}, [[P,E]|T], Env) ->
40     {{data, R}, Env1} = apply(P, Env),
41     case R of
42         false ->
43             apply(['cond'|T], Env1);
44         true ->
45             apply(E, Env1)
46     end.

```

2.1.7 atom, eq

光有这样的 `cond` 并没有什么用，还需要定义一些函数能对值进行一些判断。先来看 `atom`。`atom` 的作用和 Erlang 里 `is_atom/1` 一样。

Listing 2.28: function/func6.erl

```

143 test(atom) ->
144     {{data, true}, _} =
145         apply([atom, [quote, a]], new_env()),
146     {{data, false}, _} =
147         apply([atom, [quote, [a,b,c]]], new_env());

```

`atom` 的定义很简单。

Listing 2.29: function/func6.erl

```

47 call({fn, atom}, [X], Env) ->
48     {{data, X1}, Env1} = apply(X, Env),
49     {{data, is_atom(X1)}, Env1};

```

2. 解释器原理

再来看 `eq`。`eq` 用来比较两个值是否相同。不过有个例外，对于两个非空的 `list()`，即便所有元素都相同，在这里也认为不同。因为能判断的话，已经具有部分模式匹配的功能了，这是现在需要尽量避免的。

Listing 2.30: function/func6.erl

```
148 test(eq) ->
149     {{data, true}, _} =
150         apply([eq, [quote, []], [quote, []]],
151             new_env()),
152     {{data, true}, _} =
153         apply([eq, [quote, a], [quote, a]],
154             new_env()),
155     {{data, false}, _} =
156         apply([eq, [quote, []], [quote, a]],
157             new_env()),
158     {{data, false}, _} =
159         apply([eq, [quote, a], [quote, []]],
160             new_env()),
161     {{data, false}, _} =
162         apply([eq, [quote, [a]], [quote, [b]]],
163             new_env()),
164     {{data, false}, _} =
165         apply([eq, [quote, [a,b,c]], [quote, [a,b,c]]],
166             new_env()).
```

在 Erlang 里，可以用 `eq/2` 表示

Listing 2.31: function/func6.erl

```
75 eq([], []) ->
76     true;
77 eq(X, Y)
78     when is_atom(X), is_atom(Y) ->
79     X == Y;
80 eq(_, _) ->
81     false.
```

直接用一下 Erlang 里的 `eq/2` 就定义出 `eq` 来了。

Listing 2.32: function/func6.erl

```
50 call({fn, eq}, [X,Y], Env) ->
51     [{data,X1}, {data,Y1}], Env1} =
52     eval_list([X,Y], Env),
53     [{data, eq(X1,Y1)}, Env1].
```

2.1.8 lambda

可以用 `lambda` 定义函数。`lambda` 的第一个参数是函数的参数列表，第二个参数是一个 S 表达式。

Listing 2.33: function/func7.erl

```

187 test(lambda) ->
188     {{data, a}, _} =
189         apply([[lambda, [x], x],
190             [quote, a]], new_env()),
191     {{data, a}, _} =
192         apply([[lambda, [x], [car, x]],
193             [quote, [a,b,c]]], new_env()),
194     {{data, a}, _} =
195         apply([[lambda, [x], [car, [car, x]]],
196             [quote, [[a]]]], new_env()),
197     {{data, true}, _} =
198         apply([[lambda, [x, y], [eq, x, y]],
199             [quote, a],
200             [quote, a]], new_env()),
201     {{data, a}, _} =
202         apply([[lambda, [x, x], x],
203             [quote, a],
204             [quote, b]], new_env()).

```

Listing 2.34: function/func7.erl

```

54 call({fn, lambda}, [P,E], Env) ->
55     {{lambda, {P,E}}, Env};
56 call({lambda, {P,E}}, Args, Env) ->
57     {Args1, Env1} = eval_list(Args, Env),
58     {V, _} = apply(E, append(zip(P, Args1), Env1)),
59     {V, Env1}

```

2.1.9 macro

练习 2.4 * **defun**: `[defun, N, P, E]` 表示 `[label, N, [lambda, P, E]][3]`。

定义 S 表达式函数 `defun`

Listing 2.35: function/exercise.erl

```

28 check(defun) ->

```

2. 解释器原理

```
29     {[_,{data, a}], _} =  
30     defun:eval_list(  
31         [[defun, caar, [x], [car, [car, x]]],  
32         [caar, [quote, [[a]]]], defun:new_env()).
```

macro

Listing 2.37: function/func8.erl

```
218 test(macro) ->  
219     {[_,{data, a}], _} =  
220     eval_list(  
221         [[label, defun,  
222             [macro, [n,p,e],  
223                 [cons, [quote, label],  
224                     [cons, n,  
225                         [cons, [cons, [quote, lambda],  
226                             [cons, p,  
227                                 [cons, e,  
228                                     [quote, []]  
229                                 ]]],  
230                                 [quote, []]  
231                             ]]]  
232                     ],  
233                     [defun, caar, [x], [car, [car, x]]],  
234                     [caar, [quote, [[a]]]  
235                     ], new_env()).
```

Listing 2.38: function/func8.erl

```
61 call({fn, macro}, [P,E], Env) ->  
62     {{macro, {P,E}}, Env};  
63 call({macro, {P,E}}, Args, Env) ->  
64     Env1 = append(zip(P, quote_list(Args)), Env),  
65     {{data, E1}, _} = apply(E, Env1),  
66     apply(E1, Env).
```

2.2 模式

2.3 参考

- [1] Alan J. Perlis. Epigrams on programming. *ACM SIGPLAN Notices*, pages 7–13, 1982.
- [2] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part I. *Communications of the ACM*, 3:184–195, 1960.
- [3] Paul Graham. The roots of lisp.
<http://www.paulgraham.com/rootsoflisp.html>, May 2001.

Part II

附录

A 习题解答

练习 1.1:

Listing 1.13: erlang/fac.erl

```
1 -module(fac).  
2  
3 -export([f/1]).  
4  
5 f(1) ->  
6     1;  
7 f(N)  
8     when N > 1->  
9         N * f(N-1).
```

练习 1.2:

Listing 1.15: erlang/fib.erl

```
1 -module(fib).  
2  
3 -export([f/1]).  
4  
5 f(1) ->  
6     1;  
7 f(2) ->  
8     1;  
9 f(N)  
10    when N > 2->  
11        f(N-2) + f(N-1).
```

A. 习题解答

练习 2.1:

Listing 2.2: function/environ.erl

```
5 empty() ->
6     [].
7
8 bind(Name, Value, Env) ->
9     [{Name, Value}|Env].
10
11 is_bound(_, []) ->
12     false;
13 is_bound(Name, [{Name, _}|_]) ->
14     true;
15 is_bound(Name, [_|Env]) ->
16     is_bound(Name, Env).
17
18 get_value(Name, [{Name, Value}|_]) ->
19     Value;
20 get_value(Name, [_|Env]) ->
21     get_value(Name, Env).
```

练习 2.2:

Listing 2.5: function/sexpr.erl

```
6 is_sexpr([]) ->
7     true;
8 is_sexpr([Head|Tail]) ->
9     case is_sexpr(Head) of
10         false ->
11             false;
12         true ->
13             is_sexpr(Tail)
14     end;
15 is_sexpr(Expr) ->
16     is_atom(Expr).
```

练习 2.3:

Listing 2.15: function/id.erl

```
26 call({fn, id}, [X], Env) ->
27     apply(X, Env).
```

Listing 2.16: function/id.erl

```
32     {id,     {fn, id}}].
```

练习 2.4:

Listing 2.36: function/defun.erl

```
61 call({fn, defun}, [N,P,E], Env) ->
62     apply([label, N, [lambda, P, E]], Env).
```
