

现代编程思想

函数, 列表与递归

Hongbo Zhang

基本数据类型：函数

函数

- 在数学上，描述对应关系的一种特殊集合
 - 对于特定的输入，总是有特定的输出
- 在计算机中，对相同运算的抽象，避免大量重复定义
 - 计算半径为1的圆的面积：`3.1415 * 1 * 1`
 - 计算半径为2的圆的面积：`3.1415 * 2 * 2`
 - 计算半径为3的圆的面积：`3.1415 * 3 * 3`
 -
 - `fn 面积(半径: Double) -> Double { 3.1415 * 半径 * 半径 }`

函数

- 计算半径为1、2、3的圆的面积：

```
1. let surface_r_1: Double = { let r = 1.0; pi * r * r }  
2. let surface_r_2: Double = { let r = 2.0; pi * r * r }  
3. let surface_r_3: Double = { let r = 3.0; pi * r * r }  
4. let result = (surface_r_1, surface_r_2, surface_r_3)
```

- 使用函数后

```
1. fn area(radius: Double) -> Double { pi * radius * radius }  
2. let result = (area(1.0), area(2.0), area(3.0))
```

顶层函数的定义

```
fn <函数名> (<参数名>: <类型>, <参数名>: <类型>, ...) -> <类型> <表达式块>
```

定义的函数接口让其他使用者无需关注内部实现

```
1. fn one () -> Int {  
2.     1  
3. }  
4.  
5. fn add_char(ch: Char, str: String) -> String {  
6.     ch.to_string() + str  
7. }
```

函数的应用与计算

- 当函数定义后，可以通过 `<函数名>(<表达式>, <表达式>.....)` 的方式应用函数
 - `one()`
 - `add_char('m', "oonbit")`
 - 应用函数时，表达式与函数定义时的参数数量应当相同，且类型一一对应
这是错误的：`add_char("oonbit", 'm')`
- 计算应用函数的表达式时
 - 从左到右计算定义了参数的表达式的值
 - 替换函数内部参数
 - 简化表达式

函数的应用与计算

```

1. fn add_char(ch: Char, str: String) -> String {
2.   ch.to_string() + str
3. }
4.
5. let moonbit: String = add_char(Char::from_int(109), "oonbit")

```

`add_char(Char::from_int(109), "oonbit")`

\mapsto `add_char('m', "oonbit")`

因为 `Char::from_int(109)` \mapsto `'m'`

\mapsto `'m'.to_string() + "oonbit"`

替换表达式块中的标识符

\mapsto `"m" + "oonbit"`

因为 `m.to_string()` \mapsto `"m"`

\mapsto `"moonbit"`

因为 `"m" + "oonbit"` \mapsto `"moonbit"`

部分函数

函数定义域有的时候是输入类型的子集，因此可能会有对于输入未定义输出的情况

1. `let ch: Char = Char::from_int(-1) // 不合理输入: -1在统一码中不对应任何字符`
2. `let nan: Int = 1 / 0 // 不被允许的操作: 运行时出错并终止`

对于这种函数，我们称为**部分函数**（Partial Function）；相对的，函数对类型的每个值定义了输出的，我们称为**完全函数**（Total Function）

为了避免程序运行时因不被允许的操作中止，也为了区分对应合理与不合理输入的输出，我们使用 `Option[T]` 这一数据结构

Option的定义

`Option[T]` 分为两种情况:

- 无值: `None`
- 有值: `Some(value: T)`

例如, 我们可以用 `Option` 定义一个整数除法的完全函数

```
1. fn div(a: Int, b: Int) -> Option[Int] {  
2.   if b == 0 { None } else { Some(a / b) }  
3. }
```

`[T]` 代表 `Option` 是一个泛型类型, 包含的数值类型为类型参数 `T`, 如

- `Option[Int]`: 可能有的值类型为整数

我们将在稍后看到如何获得 `Some` 中的值

局部函数的定义

局部函数定义大多数时候可以省略参数类型和返回类型，亦可以省略名称（匿名函数）

```
1. let answer: () -> Int = fn () {  
2.   fn real_answer(i) {  
3.     42  
4.   }  
5.   real_answer("Ultimate Question")  
6. }  
7.  
8. let x: Int = answer() // 42
```

```
42  
(String) -> Int
```

```
real_answer("Ultimate Question")
```

函数在月兔中是“一等公民”：可以将函数作为参数、返回值，亦可以绑定或存储函数
我们将在后续课程中深入学习

函数的类型

`(<参数类型>, <参数类型>, <参数类型>, ...)` `->` `<返回值类型>`

- `() -> Int`
- `(Int, String, Char) -> Int`
- `((Int, Int, Int)) -> (Int, Int, Int)` 接受一个元组并返回一个元组

数据类型：列表

列表：一个数据的序列

- 我们有时会收到一些数据，具备以下特征：
 - 数据是有序的
 - 数据是可以重复的
 - 数据的数量是不定的
- 举例来说
 - 一句话中的文字：['一' '句' '话' '中' '的' '文' '字']
 - DNA序列：[G A T T A C A]
 -

列表的接口

我们定义一个单向不可变列表

以整数的列表为例（暂名之 `IntList`），它应当定义如下操作：

- 构造

- `nil : () -> IntList` 返回一个空列表
- `cons : (Int, IntList) -> IntList` 向列表添加一项

- 解构

- `head_opt : IntList -> Option[Int]` 获得第一项
- `tail : IntList -> IntList` 获得除第一项以外的项

列表的接口

测试案例

```
1. let empty_list: IntList = nil()
2. assert(head_opt(empty_list) == None)
3. assert(tail(empty_list) == empty_list)
4.
5. let list1: IntList = cons(1, empty_list)
6. assert(head_opt(list1) == Some(1))
7. assert(tail(list1) == empty_list)
8.
9. let list2: IntList = cons(2, list1)
10. assert(head_opt(list2) == Some(2))
11. assert(tail(list2) == list1)
```

月兔中的列表

- 在月兔标准库中，列表的定义为

```

1. enum List[T] {
2.     Nil // 一个空列表
3.     // 或
4.     Cons(T, List[T]) // 一个类型为T的值以及元素类型为T的子列表
5. }

```

- 列表的定义是归纳的（数学归纳法的归纳）
 - 定义了最简单的情况： Nil
 - 定义归纳的情况： Cons



列表样例

- 以下是列表
 - `let int_list: List[Int] = Cons(1, Cons(2, Cons(3, Nil)))`
 - `let char_list: List[Char] = Cons('一', Cons('句', Cons('话', Nil)))`
- 以下不是列表
 - `Cons(1, Cons(true, Cons(3, Nil)))`
因为混杂不同类型的数据
 - `Cons(1, 2)`
因为 2 不是列表
 - `Cons(1, Cons(Nil, Nil))`
因为混杂不同类型的数据

列表类型

列表亦是泛型类型: `List[<类型>]`

- 整型的列表类型为 `List[Int]`
- 字符串的列表类型为 `List[String]`
- 浮点数的列表类型为 `List[Double]`

模式匹配

我们可以通过模式匹配来分情况查看列表的内部结构

```
match <表达式> {  
  <模式1> => <表达式>  
  <模式2> => <表达式>  
}
```

模式可以用数据的构造方式定义。模式中定义了标识符，其作用域为对应表达式

```
1. fn head_opt(list: List[Int]) -> Option[Int] {  
2.   match list {  
3.     Nil => None  
4.     Cons(head, tail) => Some(head)  
5.   }  
6. }
```

模式匹配结果的化简

- 简化待匹配的表达式
- 从上到下依次匹配模式
- 匹配成功后，根据模式定义替换表达式中的标识符
- 简化表达式

```
1. fn head_opt(list: List[Int]) -> Option[Int] {  
2.     match list {  
3.         Nil => None  
4.         Cons(head, tail) => Some(head)  
5.     }  
6. }  
7.  
8. let first_elem: Option[Int] = head_opt(Cons(1, Cons(2, Nil)))
```

模式匹配结果的化简

```
1. head_opt(Cons(1, Cons(2, Nil)))
```

↳ (替换函数内的标识符)

```
1. match Cons(1, Cons(2, Nil)) {  
2.   Nil => None  
3.   Cons(head, tail) => Some(head)  
4. }
```

↳ `Some(1)` (匹配并根据模式定义替换表达式中的标识符)

上面一步可以理解为:

```
1. {  
2.   let head = 1  
3.   let tail = Cons(2, Nil)  
4.   Some(head)  
5. }
```

模式匹配Option

同样地，我们也可以用模式匹配查看 `Option` 的结构来获得值

```
1. fn get_or_else(option_int: Option[Int], default: Int) -> Int {
2.     match option_int {
3.         None => default
4.         Some(value) => value
5.     }
6. }
```

模式匹配中，亦可以省略部分情况（如确认存在值），来构造部分函数

```
1. fn get(option_int: Option[Int]) -> Int {
2.     match option_int { // 编辑器会警告我们有模式尚未被匹配
3.         Some(value) => value
4.         // 若option_int为None则会程序出错中止
5.     }
6. }
```

算法：递归

递归的例子

- GNU is Not Unix
- Wine Is Not an Emulator
- 斐波那契数列的计算（第一项为1，第二项为1，之后第n项为前两项之和）
- 山里有个庙，庙里有个老和尚和小和尚，一天，老和尚给小和尚讲故事：
 - “山里有个庙，庙里有个老和尚和小和尚，一天，老和尚给小和尚讲故事：
 - ‘山里有个庙，庙里有个老和尚和小和尚，一天，老和尚给小和尚讲故事...’”

递归

- 递归是将问题分解为与原问题相似的、规模更小的问题来求解
 - 递归应当有边界条件
 - 在函数的定义中，直接或间接地使用函数自身

```
1. fn fib(n: Int) -> Int {  
2.   if n == 1 || n == 2 { 1 } else { fib (n-1) + fib (n-2) }  
3. }
```

```
1. fn even(n: Int) -> Bool {  
2.   n != 1 && (n == 0 || odd(n - 1))  
3. }  
4. fn odd(n: Int) -> Bool {  
5.   n != 0 && (n == 1 || even(n - 1))  
6. }
```

在列表上的递归

列表是递归定义的，因此适合用递归函数与模式匹配一起定义列表的操作函数

一个列表可以为

- `List::Nil`：一个空列表
或
- `List::Cons(head, tail)`：一个值 `head` 以及一个列表 `tail`

```
1. fn length(list: List[Int]) -> Int {  
2.   match list {  
3.     Nil => 0  
4.     Cons(_, tl) => 1 + length(tl)  
5.   }  
6. }
```

递归的计算

```
1. let n = length(Cons(1, Cons(2, Nil)))
2.
3. fn length(list: List[Int]) -> Int {
4.     match list {
5.         Nil => 0
6.         Cons(_, tl) => 1 + length(tl)
7.     }
8. }
```

递归的计算

```
1. length(Cons(1, Cons(2, Nil)))
```

↳ 替换为函数定义

```
1. match Cons(1, Cons(2, Nil)) {  
2.   Nil => 0  
3.   Cons(_, tl) => 1 + length(tl) // tl = Cons(2, Nil)  
4. }
```

↳ 模式匹配并替换标识符

```
1. 1 + length(Cons(2, Nil))
```

↳ 再次调用函数

```
1. 1 + match Cons(2, Nil) { ... }
```

递归的计算

```
1. 1 + match Cons(2, Nil) {  
2.   Nil => 0  
3.   Cons(_, tl) => 1 + length(tl) // tl = Nil  
4. }
```

↳ 模式匹配并替换标识符

```
1. 1 + 1 + length(Nil)
```

...

↳ 1 + 1 + 0 ↳ 2

结构化递归

对基于递归定义的数据结构

- 定义对基础数据结构的计算
- 定义对递归数据结构的计算

```
1. fn length(list: List[Int]) -> Int {  
2.   match list {  
3.     Nil => 0 // 终结情形  
4.     Cons(_, tl) => 1 + length(tl) // 递归情形  
5.   }  
6. }
```

每一次递归，我们都对原数据的子结构进行递归，且我们定义了终结情形，因此我们可以保证程序终结

通常我们可以用数学归纳法证明结构化递归定义的函数是正确的

数学归纳法：以子列表长度为例

- 命题：对于任意列表 `a`，令列表 `a` 长度为 l_1 ，子列表 `tail(a)` 长度为 l_2 ，则总有 $l_1 \geq l_2$

```
1. fn tail(list: List[Int]) -> List[Int] {  
2.   match list {  
3.     Nil => Nil  
4.     Cons(_, tail) => tail  
5.   }  
6. }
```

- 证明：对 `a` 分类讨论
 - 若 `a` 为空 (`Nil`)，则子列表 `tail(a) == a`，两者长度均为0，命题成立
 - 若 `a` 为非空 (`Cons(head, tail)`)，则子列表 `tail(Cons(head, tail)) == tail`，可知 $l_1 = l_2 + 1 > l_2$ ，命题成立
 - 由数学归纳法，原命题成立

算法：动态规划

斐波那契数列的计算方式

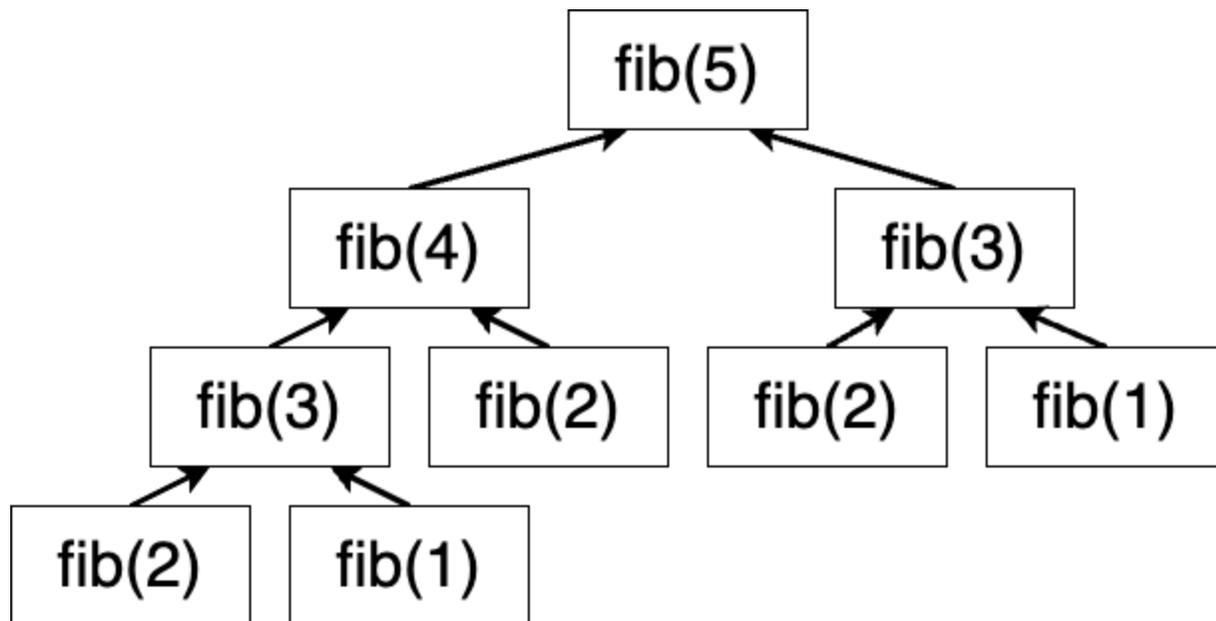
1, 1, 2, 3, 5, 8, 13, 21,

不同的斐波那契数列的计算方式带来的不同性能差别 (num > 40)

```
1. // 002_fib.mbt, try.moonbitlang.cn
2. fn fib(num: Int) -> Int {
3.   if num == 1 || num == 2 { 1 } else { fib(num - 1) + fib(num - 2) }
4. }
5.
6. fn fib2(num : Int) -> Int {
7.   fn aux(n, acc1, acc2) {
8.     match n {
9.       0 => acc1
10.      1 => acc2
11.      _ => aux(n - 1, acc2, acc1 + acc2)
12.    }
13.  }
14.  aux(num, 0, 1)
15. }
```

简单的斐波那契数列的计算方式

```
1. fn fib(num: Int) -> Int64 {  
2.   if num == 1 || num == 2 { 1L } else { fib(num - 1) + fib(num - 2) }  
3. }
```



我们观察到了大量的重复计算

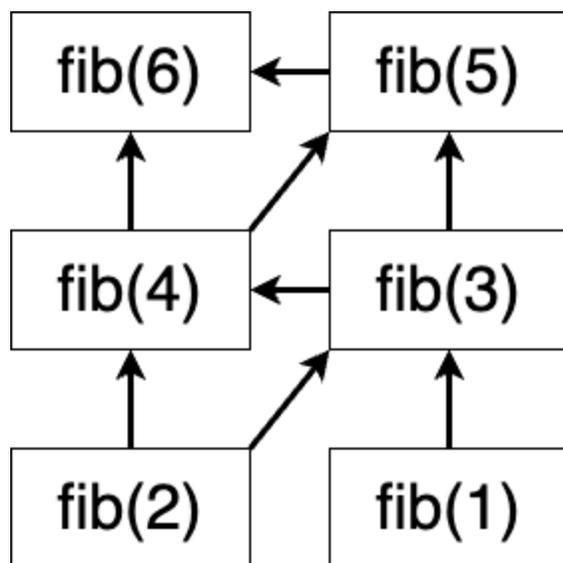
动态规划

- 将问题分解为与原问题相似的、规模更小的问题来求解
- 适用于子问题
 - 有重叠子问题：动态规划对每个子问题求解一次，将其保存，避免重复运算
 - 有最优子结构：局部最优解可以决定全局最优解
- 动态规划分为自顶向下和自底向上
 - 自顶向下：针对每个子问题，如果已求解，直接使用缓存结果；否则求解并缓存
 - 自底向上：先解决子问题，再从子问题的解构建更大的子问题的解

动态规划：以斐波那契数列为例

求解斐波那契数列符合使用动态规划的条件

- 有最优子结构： $\text{fib}(n)$ 的值可以被用来计算 $\text{fib}(n + 1)$ 和 $\text{fib}(n + 2)$ 的值
- 有重叠子结构： $\text{fib}(n + 1)$ 与 $\text{fib}(n + 2)$ 的求解均需要子问题 $\text{fib}(n)$ 的值



自顶向下：以斐波那契数列为例

- 我们需要一个数据结构，平均存取速度应当与当前存储数据量大小无关
- 以求解斐波那契数列为例，我们假设的 `IntMap` 应有如下接口：

```
1. fn make(num: Int) -> IntMap // 创建数据结构
2. fn put(map: IntMap, num: Int, value: Int64) -> IntMap // 存储数据，只执行一次
3. fn get(map: IntMap, num: Int) -> Option[Int64] // 提取数据
```

- 符合条件的数据结构有很多，我们的样例代码以 `AVLMap[Int, Int64]` 为例
 - 我们无需关注它的具体实现。我们可以用 `Array[Option[Int64]]` 等替代

自顶向下：以斐波那契数列为例

- 我们每次计算时先查看当前数据结构中是否存有结果
 - 若有，则直接使用
 - 若无，并将结果添加至数据结构中

```
1. fn fib1(num: Int) -> Int64 {
2.   fn aux(num: Int, map: AVLMap[Int, Int64]) -> (Int64, AVLMap[Int, Int64]) {
3.     match get(map, num) {
4.       Some(result) => (result, map)
5.       None => {
6.         let (result_1, map_1) = aux(num - 1, map)
7.         let (result_2, map_2) = aux(num - 2, map_1)
8.         (result_1 + result_2, put(map_2, num, result_1 + result_2))
9.       }
10.    }
11.  }
12.  let map = put(put(make(num), 1, 1L), 2, 1L)
13.  aux(num, map).0
14. }
```

可变变量

注意到 `map: AVLMap[Int, Int64]` 被不断传递。为了简化写法，月兔提供可变变量

```
1. fn fib1_mut(num: Int) -> Int64 {
2.   let mut map = put(put(make(num), 1, 1L), 2, 1L) // 通过let mut声明可变变量
3.   fn aux(num: Int) -> Int64 {
4.     match get(map, num) {
5.       Some(result) => result
6.       None => {
7.         let result_1 = aux(num - 1)
8.         let result_2 = aux(num - 2)
9.         // 通过 <变量> = <值> 修改绑定的值
10.        map = put(map, num, result_1 + result_2)
11.        result_1 + result_2
12.      }
13.    }
14.  }
15.  aux(num)
16. }
```

自底向上：以斐波那契数列为例

- 我们从第一项出发，逐个计算之后的值，并将当前项的计算结果存入数据结构

```
1. fn fib2(num: Int) -> Int64 {
2.   fn aux(n: Int, map: AVLMap[Int, Int64]) -> Int64 {
3.     let result = get_or_else(get(map, n - 1), 1L) + get_or_else(get(map, n - 2), 1L)
4.     if n == num { result } else { aux(n + 1, put(map, n, result)) }
5.   }
6.   let map = put(put(make(num), 0, 0L), 1, 1L)
7.   aux(1, map)
8. }
```

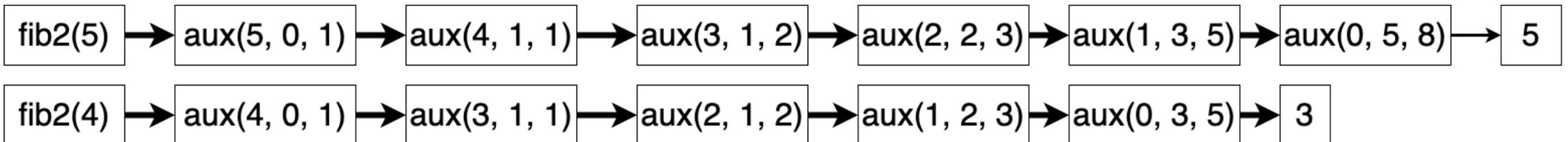
自底向上：以斐波那契数列为例

- 注意到，我们每次只需保存当前项的前两个值，因此我们可以舍弃数据结构，直接通过递归参数传递

```

1. fn fib2(num : Int) -> Int64 {
2.   fn aux(n: Int, acc1: Int64, acc2: Int64) -> Int64 {
3.     match n {
4.       0 => acc1
5.       _ => aux(n - 1, acc2, acc1 + acc2)
6.     }
7.   }
8.   aux(num, 0L, 1L)
9. }

```



总结

- 本章节我们学习了
 - 基础数据类型：函数的定义与运算
 - 数据结构：列表的定义与模式匹配
 - 算法：递归的含义与运算，以及动态规划
- 拓展阅读
 - *Software Foundations* 前三章 或
 - *Programming Language Foundations in Agda* 前三章
 - 《算法导论》第十四章