

现代编程思想

月兔开发与月兔中的表达式

月兔公开课课程组

月兔MoonBit

- 现代静态类型**多范式**编程语言
- 语法轻量，易上手
- 参考资料: moonbitlang.cn

开发环境准备

月兔开发环境

- 开发工具
 - VS Code插件：语言服务器、包级别构建等功能
 - 命令行工具：项目级别构建、项目测试、依赖管理等功能
 - [语言导览](#)：快速学习月兔的互动教程
- 开发环境
 - 浏览器环境（无命令行工具）
 - 云原生开发环境（如[Github Codespaces](#)）
 - 本地开发环境

浏览器环境

- 访问try.moonbitlang.cn，或从[官网](#)点击“试用”
- 试用环境可以快速创建文件并运行
- 试用环境提供代码样例，方便熟悉月兔语法
- 试用环境提供分享功能

本地开发环境

- 安装[VS Code](#)或[VS Codium](#)，并安装"MoonBit Language"插件
- 安装[命令行工具](#)（支持Windows、MacOS与Ubuntu等环境），并参考[月兔构建系统教程](#)

月兔中的表达式

一个典型的月兔程序

```
1. //顶层函数定义
2. fn num_water_bottles(num_bottles : Int, num_exchange : Int) -> Int {
3.     for num_full = num_bottles, num_empty = 0, num_total = 0 {
4.         if num_full > 0 {
5.             continue 0, num_empty + num_full, num_total + num_full
6.         } else if num_empty >= num_exchange {
7.             continue num_empty / num_exchange, num_empty % num_exchange, num_total
8.         } else {
9.             break num_total
10.        }
11.    }
12. }
13.
14. // 程序测试
15. test {
16.     assert_eq(num_water_bottles(9, 3), 13)
17.     assert_eq(num_water_bottles(15, 4), 19)
18. }
```


基于表达式编程

- 为了写出正确的程序，我们需要知道程序是如何被运算的：建立**计算模型**来理解程序的运算过程
- 月兔程序可以通过**面向值编程**来描述
 - 面向值编程：定义程序**是什么**
 - 我们写的月兔代码都是表达一个值的**表达式**
 - 命令式编程风格：定义程序**做什么**
 - 程序由修改程序状态的**命令**组成
 - 创建名为x的变量
 - 令x为5
 - 令y指向x
 -

类型、值与表达式

类型	值	运算	表达式
Int	-1 0 1 2	+ - * /	5 (3 + y * x)
Double	0.12 3.1415	+ - * /	3.0 * (4.0 * a)
String	"hello" "Moonbit"	+	"Hello, " + "MoonBit"
Bool	true false	&& !	!b1 b2

- 每一个**类型**对应一个**值**的集合
- 每一个**表达式**由基于值的**运算**构成，并且可以简化为一个值（或已经是一个值）
- 可以使用括号来嵌套表达式

静态 vs 动态

- “静态”指在程序运行之前的事物
- “动态”指在程序运行之时的事物

月兔拥有静态类型系统：在程序运行之前，编译器即会检查程序的类型是否定义良好

静态类型

- 每一个标识符都关联着唯一的一个类型
- “冒号”用于关联一个标识符和它的类型
 - `x : Int`
 - `a : Double`
 - `s : String`
- 每一个月兔表达式都有一个唯一的类型，这个类型由组成它的子表达式决定

`x + (a + 0.2).to_int()`
`: Int` `: Double`
 └──────────┘
 `: Double`
 └──────────────────────────┘
 `: Int`
 └──────────────────────────────────┘
 `: Int`

静态类型检查

- 月兔的编译器会在运行前使用**类型推导**来确认程序是否正确使用类型
- 月兔的开发工具可以在开发时实时提示检查到的类型错误

`s + (a + 0.2).to_int()`

`s` : String

`(a + 0.2)` : Double

`.to_int()` : Int

Error: type mismatch

图中的错误源于字符串与数字不能直接相加

月兔的基本数据类型

- 逻辑值（布尔值）
- 整数（整型、长整形）
- 浮点数（单精度浮点数、双精度浮点数）
- 字符与字符串
- 多元组
-

逻辑值（布尔值）

- 月兔中逻辑值的类型为 `Bool`
- 逻辑值只有两个可能值：
 - `true`
 - `false`
- 常见运算
 - 非：非真为假，非假为真 `!true == false`
 - 与：两者皆真才为真 `true && false == false`
 - 或：两者皆假才为假 `true || false == true`
- 小练习：如何用或、与、非定义**异或**（一者为真才为真）

整数类型

- 作为基础类型的整数分为两个类型，分别有不同的范围：
 - 整型 `Int`：从 -2^{31} 到 $2^{31} - 1$ （即从 -2147483648 到 2147483647）
 - 长整型 `Int64`：从 -2^{63} 到 $2^{63} - 1$
- 在月兔中，整数相除依然获得整数，其结果为商
 - 被除数 ÷ 除数 = 商 余数
- 对整数进行超出范围的运算后会溢出
 - $2147483647 + 1$ 结果为 -2147483648
- 整型只能与整型进行四则运算，长整型只能与长整型进行四则运算
 - 长整型数值后需加 `L` 进行标记：如 `2147483648L` `-2147483649L`
 - `Int` 与 `Int64` 互相转换：`(100).to_int64()` `100L.to_int()`
- 小练习：如何计算两个正的 `Int` 的平均数？小心溢出！

浮点数类型

- 作为基础类型的浮点数只能表示有限小数，且只能近似表达
 - 浮点数在计算机内部表现形式为尾数 `b` 与指数 `e`（均为整数）： $b \times 2^e$
 - 例如：`0.1 + 0.2 != 0.3`
- 月兔中浮点数类型分为单精度浮点数 `Float` 和双精度浮点数 `Double`
 - 整数和浮点数不能混合运算：`1 + 2.0` 报错
 - `Int` 可以转换为 `Double`：`(1).toDouble() == 1.0`
 - `Double` 转化为 `Int`：`(-1.2).toInt() == -1`
- 小练习：如何通过整数与浮点数的相互转换，来比较 `0.1 + 0.2` 与 `0.3`？

字符与字符串

- 月兔中字符类型为: `Char`, 字符串类型为: `String`
 - 字符用英文单引号标识: `'a'`
 - 字符串用英文双引号标识: `"Hello!"`
- 字符有不同编码方式
 - ASCII (美国信息交换标准代码): 0~127, 支持英文字符及常见符号
 - 例如: A~Z对应65~90
 - Unicode (统一码): 支持中文等多国字符及表情符号 (emoji), 兼容ASCII, 有UTF-8、UTF-16等编码方式
 - 例如: “月”“兔”分别对应26376与20820
 - `Int` 转化为 `Char`: `Char::from_int(65) == 'A'`

多元组

- 多元组允许我们将表达固定长度的、不同类型的数据组合
 - `("Bob", 3) : (String, Int)`
 - `(2023, 10, 24) : (Int, Int, Int)`
- 可以通过从0开始的下标访问数据
 - `(2023, 10, 24).0 == 2023`
 - `(2023, 10, 24).1 == 10`

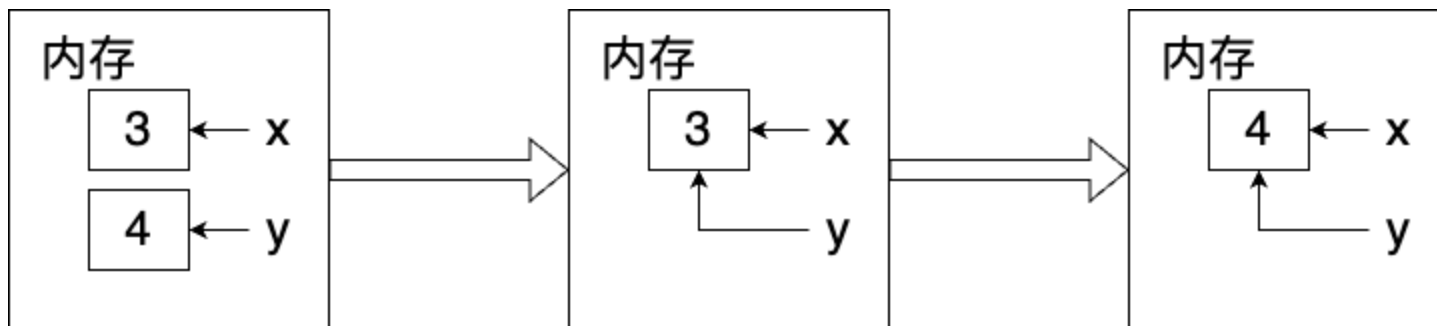
其他数据类型

- 月兔有丰富的类型结构
 - 函数类型
 - `add : (Int, Int) -> Int`
 - 单值类型
 - `() : Unit`
 - 列表类型
 - `List::Cons(1, Nil) : List[Int]`
 -
- 我们会在后续课程中见到它们，以及自定义数据结构

计算表达式的值

简化 vs 运行

- 我们可以将月兔的表达式看作定义**值**的一种方式
- 我们可以将月兔的运行过程看作一系列的**计算或简化**的求值步骤
- 相比之下，命令式编程则可以被看作执行一系列**行为或者命令**
 - 创建指针 **x** 与 **y** 并分配内存，令 **x** 值为3，令 **y** 值为4
 - 令 **y** 指向 **x**
 - 令 **x** 自增
 -



简化表达式

- $3 \Rightarrow 3$ (值即为本身)
- $3 + 4 \Rightarrow 7$
- $2 * (4 + 5) \Rightarrow 18$
- $\text{num_water_bottles}(9, 3) \Rightarrow 13$

我们将<表达式>简化为<值>记作: $\langle \text{表达式} \rangle \Rightarrow \langle \text{值} \rangle$

单步计算

我们将分解 \Rightarrow 简化的流程为**单步计算**，记作 \mapsto

举例而言：

$$\begin{aligned}
 & (2 + 3) * (5 - 2) \\
 \mapsto & 5 * (5 - 2) && \text{因为 } 2 + 3 \mapsto 5 \\
 \mapsto & 5 * 3 && \text{因为 } 5 - 2 \mapsto 3 \\
 \mapsto & 15 && \text{因为 } 5 * 3 \mapsto 15
 \end{aligned}$$

故 $(2 + 3) * (5 - 2) \Rightarrow 15$

数值绑定

```
let <标识符> : <类型> = <表达式>
```

- 数值绑定将一个**名称**（或者说**标识符**）赋予一个用表达式定义的值
- 类型声明多数时候可省略，月兔编译器会根据表达式类型进行推断
 - `let x = 10`
 - `let y = "String"`
- 对一个标识符进行多次绑定将会**遮掩**之前的值，而不会发生修改

表达式块

```
{  
  数值绑定  
  数值绑定  
  .....  
  表达式  
}
```

表达式块的类型即为最后的表达式的类型，表达式块的值即为最后表达式的值

```
1. let 顶层标识符 = 10 // 顶层（全局）指定义在一个文件中所有表达式块外部定义的函数或标识符  
2.  
3. fn 顶层函数() -> Unit {  
4.   fn 本地函数() { // 本地（局部）指某个表达式块内部的函数或标识符  
5.     let 本地标识符 = 1 // 局部数值绑定可以简化  
6.     本地标识符 // 表达式块的值  
7.   }  
8. }
```

作用域

即定义或数值绑定有效的范围

- 全局（整个文件）
- 局部（从定义到所在表达式块结束）

```
1  let value: Int = {  
2    let x = 1  
3    let tmp: Int = x * 2  
4    let another_tmp: Int = {  
5      let tmp: Int = x * 3  
6    }  
7    tmp + another_tmp + y  
8  }  
9  
10  
11  
12 let y: Int = 10
```

顶层定义的作用域为全局，而本地定义的作用域为局部；本地的定义会遮掩之前的定义

数值绑定下的表达式简化

- 简化数值绑定右侧的表达式
- 将出现的标识符替换为简化后的值
- 省略数值绑定部分
- 对剩余表达式进行化简

```
1. let y : Int = 10
2. let value = {
3.   let x = 1
4.   let tmp = x * 2
5.   let another_tmp = {
6.     let tmp = x * 3
7.     tmp
8.   }
9.   tmp + another_tmp + y
10. }
```

数值绑定下的表达式简化

- 简化数值绑定右侧的表达式
- 将出现的标识符**替换**为简化后的值
- 省略数值绑定部分
- 对剩余表达式进行化简

```
1. let y : Int = 10
2. let value = {
3.   let x = 1
4.   let tmp = 1 * 2 // 替换x
5.   let another_tmp = {
6.     let tmp = 1 * 3 // 替换x
7.     tmp
8.   }
9.   tmp + another_tmp + 10 // 替换y
10. }
```

数值绑定下的表达式简化

- 简化数值绑定右侧的表达式
- 将出现的标识符替换为简化后的值
- 省略数值绑定部分
- 对剩余表达式进行化简

```
1. // 省略y的定义
2. let value = {
3.   // 省略x的定义
4.   let tmp = 2 // 简化右侧表达式
5.   let another_tmp = {
6.     let tmp = 3 // 简化右侧表达式
7.     tmp
8.   }
9.   tmp + another_tmp + 10
10. }
```

数值绑定下的表达式简化

- 简化数值绑定右侧的表达式
- 将出现的标识符替换为简化后的值
- 省略数值绑定部分
- 对剩余表达式进行化简

```
1. let value = {  
2.   let tmp = 2  
3.   let another_tmp = {  
4.     let tmp = 3  
5.     3 // 替换表达式  
6.   }  
7.   tmp + another_tmp + 10  
8. }
```

数值绑定下的表达式简化

- 简化数值绑定右侧的表达式
- 将出现的标识符**替换**为简化后的值
- 省略数值绑定部分
- 对剩余表达式进行化简

```
1. let value = {  
2.   let tmp = 2  
3.   let another_tmp = 3 // 简化右侧表达式  
4.   tmp + another_tmp + 10  
5. }
```


数值绑定下的表达式简化

- 简化数值绑定右侧的表达式
- 将出现的标识符**替换**为简化后的值
- 省略数值绑定部分
- 对剩余表达式进行化简

```
1. let value = {  
2.   let tmp = 2  
3.   let another_tmp = 3 // 简化右侧表达式  
4.   2 + 3 + 10 // 替换表达式  
5. }
```

数值绑定下的表达式简化

- 简化数值绑定右侧的表达式
- 将出现的标识符**替换**为简化后的值
- 省略数值绑定部分
- 对剩余表达式进行化简

```
1. let value = 15
```

条件表达式

`if 条件 条件为真的表达式块 else 条件为假的表达式块`

月兔的条件表达式也是**表达式**，因此可以被用在其他表达式内

- `(if 1 < 100 { 1 } else { 0 }) * 10`
- `(if x > y { "x" } else { "y" }) + " is bigger"`
- `if 0.1 + 0.2 == 0.3 { "Great!" } else { "C'est la vie :-)" }`

条件表达式类型

if 条件 表达式块 else 表达式块

- 分支的表达式块的类型需相同
- 整个条件表达式的类型取决于分支的表达式块的类型
- 条件的类型需为逻辑值

```
if 0.1 + 0.2 == 0.3 { "Great!" } else { "C'est la vie :-)" }
```

Diagram illustrating the type inference for the conditional expression:

- The condition `0.1 + 0.2 == 0.3` is of type `Bool`.
- The first branch `{ "Great!" }` is of type `String`.
- The second branch `{ "C'est la vie :-)" }` is of type `String`.
- The entire `if-else` expression is of type `String`.

简化条件表达式

条件式表达式的值为哪个分支的值取决于条件的简化结果为真或假

例如：

```
if 1 < 100 { 1 } else { 0 } * 10
```

⇒

```
if true { 1 } else { 0 } * 10
```

⇒

```
1 * 10
```

⇒

```
10
```

条件表达式必须有两个分支（否则如果条件为假，表达式的值该是什么呢？）

总结

我们本章节学习了

- 如何配置月兔开发环境
 - 浏览器开发环境
 - 云原生开发环境
 - 本地开发环境
- 月兔基本数据类型
 - 逻辑值
 - 整数和浮点数
 - 字符和字符串
 - 多元组
- 如何以表达式和值来看待月兔程序，以简化求值来理解月兔程序的运行