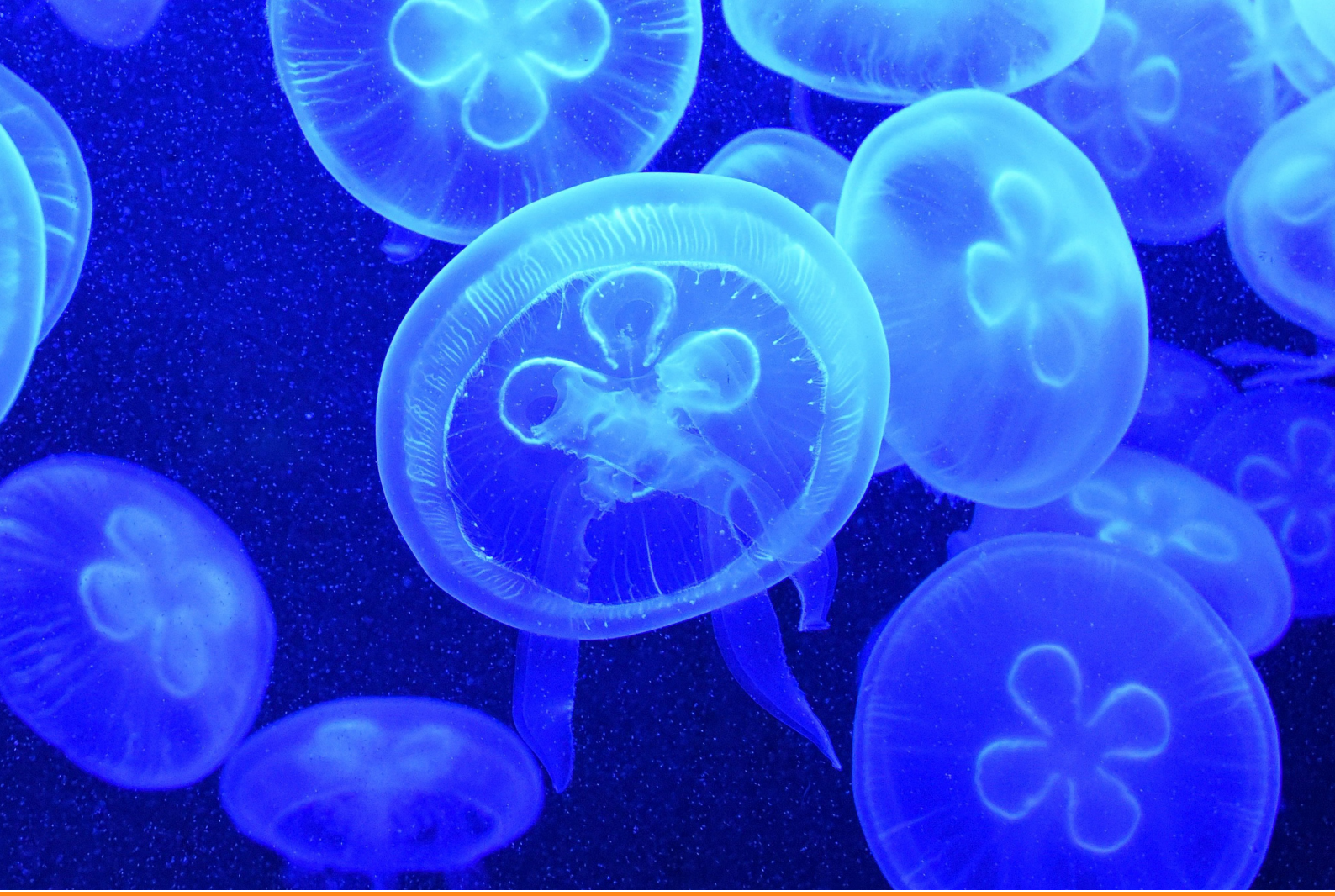

AutoScript 编程语言

作者：萧茗葛

时间：April 2023

版本：0.1

目录

第一部分

基本知识

第一章 开始使用 AutoScript

任何编程语言的学习都从一个简单的程序开始，本章我们将尝试实现一个词频统计程序。

内容提要

- 变量的定义
- 主函数
- 类型
- 注释
- 标准输入和输出
- if 语句
- while 语句
- for 语句
- 函数
- 运算符
- 元组
- 结构

1.1 Hello World 程序

让我们从最简单的 Hello World 程序来开始 AutoScript 的学习。任何 AutoScript 程序都需要至少有一个*语句*。通常 AutoScript 要求代码中有一个 `main` *函数*，在经过编译后系统会尝试调用这个函数，并执行其中的代码。

```
1 const main = funct () -> do {
2     print "Hello, World!";
3 };
```

虽然只有短短三行程序，但其中包含的知识点需要我们反复巩固才能深入理解，这里让我们逐个部分简单地说明它们的含义和作用。

- `const`：是一个*存储类型限定符*，用来定义一个变量。特别地，`const` 表示*常量存储期*，常用于函数和类型的定义，我们这里先不细究其含义。类似这样拥有特殊含义的“单词”被称为*关键字*。
- `main`：也就是我们的**主函数（Main Function）**，它是一个*变量*。在 AutoScript 中，几乎所有的概念都可以绑定到一个名字上，此时它就成为了一个变量；不过，关键字的名字不能用于变量绑定。主函数是程序的入口，系统会以*调用*主函数作为程序运行的开始。
- `=`：是一个**分隔符（Delimiter）**，用来控制代码的结构。这里的 `=` 用来定义变量，也被称为*初始化器提示符*。
- `funct`：是一个*实体类别提示符*，其提示后续实体拥有的*类别*。`funct` 也是一个*阐述关键字*，我们可以用符号 `\` 来代替这个关键字。
- `()`：一个空的*函数参数列表*，这里用来声明函数的参数。对于我们上面的例子，不需要接受任何参数，因此圆括号中是空的。
- `->`：是一个**运算符（Operator）**。在 AutoScript 中有许多这样的连接符号，其将程序中的一些结构串联起来形成特殊的含义。对于函数来说，参数列表和函数体之间通过 `->` 串联就得到一个**函数表达式（Function Expression）**。
- `do`：是一个关键字，用来声明一个 `do` *表达式*，后者是函数体常见的形式。在函数被调用时，`do` 表达式的*语句块*，即大括号包围的部分中所有语句会被依次执行。
- `{...}`：是一个语句块。写在 `do` 关键字后可以构成一个 `do` 表达式。
- `print`：是一个关键字，会将后面的内容打印在指定的输出窗口上。
- `"..."`：是一个*字符串*，在 AutoScript 代码中，有必要区分名字和字符串：前者是一个关键字或变量，后者则是一份数据。AutoScript 中使用双引号来表示字符串。
- `;`：是一个分隔符，用来分开两个语句。

上面的程序会经过 AutoScript 编译器 `asc` 编译后生成一个可执行文件，执行后就会在命令行中输出“Hello, World!”。不过，如果直接在命令行中运行 `asci`，则可以通过下面的代码来达到相同的效果：

```
print "Hello, World!";
```

这是因为 asci 是一个 REPL 程序，我们输入的所有指令都会被当场解释为某个中间代码，并通过 asci 解释运行。由于 REPL 中一些代码的行为和常规程序不同，因此我们在需要使用 REPL 运行代码的例子会特别指出；没有特别说明时，都应该默认时常规的需要经过编译的代码。

1.2 输入和输出

输入输出（IO）并不是一个简单的概念，但是它们是从开始学习编程就一定会接触的编程语言知识。AutoScript 将标准输入输出纳入了*语言特性*的范畴中，即我们不需要导入标准库来使用常规的输入输出操作；具体来说，我们可以通过 print 和 scan 关键字来进行 IO 操作。

```
const main = funct () -> do {
    auto x : Int = undefined;
    scan x;
    print x;
};
```

这里出现了一个新的*存储限定符* auto，用来表示*自动存储期*；它常用于普通数据类型变量的声明（比如一个整数或字符串等）。和 main 函数不同的是，我们没有在变量名字后直接使用 = 分隔符来直接定义它，而是使用 : 分隔符来声明它的**类型（Type）**。

AutoScript 中所有变量都拥有类型，它描述了一个对象的内存布局和行为特点，从而有助于在编译时对代码的行为进行检测和优化。上面的例子中，我们让 x 的类型为 Int，并通过 undefined 来作为它的初始值。程序运行到这里时，会为 x 分配空间但不对其初始化；所有读取 undefined 的行为都是未定义的[1]。好在随后我们将其交给了 scan 语句，后者会根据命令行中的输入来为 x 赋值。

AutoScript 提供了一个更加方便的语法来从标准输入读取内容。

```
const main = funct () -> do {
    scan x : Int;
    print x;
};
```

如上面所示，我们可以在 scan 语句中直接声明一个变量，它会通过输入的内容来初始化。这样就不用担心在变量声明到使用 scan 语句之间误用变量了。这个技巧不仅限于 scan 语句中，不过碍于篇幅让我们先记住它在 scan 语句中的便捷使用。

1.3 空格和注释

读者或许已经注意到，前面的示例代码中，我们采用了特定的格式，比如在 do 表达式的语句块中会换行并缩进、函数体中的语句每个都占用一行、关键字和符号之间用空格分开等。实际上这只是一个编写习惯。AutoScript 的名字和符号之间不会受空格和换行的影响。因此，下面的代码也是合法的：

[1] **未定义行为（Undefined Behavior）**是 C/C++ 中臭名昭著的特性之一，不过有的时候它也非常合理。AutoScript 中我们既然都字面意义上暗示了 undefined 相关行为是未定义的，因此（或许）它能警示到更多人。

```
const
    main = funct
() -> do { print "Hello, World"
;}
```

当然，这绝不是我们推荐的写法。通常来讲，代码应该通过换行和缩进来提示其结构，建议使用本文中使用的风格。

除了空格之外，AutoScript 也会忽略**注释（Comment）**的存在。一共有两种注释，其一是行内注释，可以出现在每一行代码的末尾；其二是行间注释，可以出现在任何空格允许出现的地方。

```
// 主函数
const main = funct /* 参数列表 */ () -> {
    /*
        这里什么都没有写
     */
};
```

注释在代码中比较常见，它用来解释抽象的程序行为。

1.4 控制流

程序通常是从前到后依次执行的。不过在一些情况下，我们需要选择地执行部分代码，或重复执行部分代码，此时需要一些*控制流语句*来描述代码行为。本节中介绍其中最常用的三种。

1.4.1 if 语句

程序中，我们时常会根据某个条件来判断是否要执行一些语句，比如我们根据工资的多少来决定税收的比重。

```
const main = funct () -> do {
    scan income : Int;
    if (income < 2000) {
        print "不需要缴税";
    }
    if (income >= 2000 and income < 8000) {
        print "需要缴 5% 税";
    }
    if (income >= 8000 and income < 20000) {
        print "需要缴 10% 税";
    }
    if (income >= 20000) {
        print "需要缴 20% 税";
    }
};
```

可以看到，if 语句包含了 if 关键字、一个由圆括号包围的判断表达式，还有一个语句块。当判断表达式求值为真时，就会执行语句块中的代码。此外，上面用到了 < 和 >= 运算符，它们的含义和数学中的小于和大于等于相同，而逻辑运算符 and 当两边同时成立时才判断为真。

1.4.2 while 语句

while 语句会按照某个条件重复执行一串代码，参考下面的“猜数游戏”：

```
const main = funct () -> do {
    auto x = Math.randint(0, 100);
    print "Guess a number between 0 and 100 (excluded)";
    scan guessed : mut Int;

    while (guessed <> x) {
        if (guessed < x) {
            print "Try a bigger one!";
        }
        else {
            print "Try a smaller one!";
        }
        scan guessed;
    }
    print "Congrats! You've nailed it!";
};
```

和 if 语句类似地，while 语句由 while 关键字、一个圆括号包围的表达式还有一个语句块组成。执行到这个语句时，会首先判断表达式是否为真，若是则执行一次语句块，随后再次进行判断。直到判断条件为假前，循环会被不断执行。此外，上面用到了 <> 运算符，它判断两个数是否不相等。我们也用到了 else 子句，它紧跟在一个 if 语句后，如果后者的判断条件为假，则会进入 else 子句的语句块中。

上面的例子中值得特别说明的是，我们将 guessed 声明为 mut Int 类型，这里的 mut 是一个*类型修饰符*，其声明这个类型的对象是可变的。AutoScript 中默认所有变量都是不可变的，除非其被 mut 修饰。由于我们在每次循环中都通过 scan 语句重新对其赋值，因此需要将器声明为可变的[2]。

1.4.3 for 语句

对于一个*范围*，我们可以通过 for 语句遍历其中的元素[3]。下面让我们以数组类型为例。

```
const main = funct () -> do {
    auto arr : Int[3] = (1, 2, 3);
    for (e : Int from arr) {
        print e;
    }
};
```

[2]此前我们使用 scan 初始化的变量并不需要声明为 mut，这是因为可变性只在初始化之后才有意义。

[3]for 语句的实际功能要更加复杂，但这里我们先只介绍其用作范围遍历的功能。

这里，`Int[3]` 表示拥有三个 `Int` 对象的数组类型，其初始化为三个连续的元素 `1`，`2`，`3`。`e : Int from arr` 是一个比较复杂的结构，这里可以理解成声明了变量 `e : Int` 后依次从 `arr` 中取出元素。

1.5 函数和运算符

本节让我们简单介绍一下**函数（Function）**的定义和功能。函数可以看作是一个*子程序*，它拟接受一系列*参数*，并对一个表达式求值。当我们**调用（Invoke）**一个函数时，会将调用处的**参数（Argument）**用于初始化函数参数列表中的变量，并执行函数体。

```
const add = funct (x : Int, y : Int) -> x + y;  // 函数返回 x + y 的值

const main = funct () -> do {
    scan a : Int;
    scan b : Int;
    print add(a, b);
};
```

上面的程序中，`add` 和 `main` 都是函数。在 `main` 函数体中我们调用了 `add` 函数，其语法是函数名紧跟一个数对 `(a, b)`。调用函数时需要保证和参数列表中参数个数和对应的类型相同[4]。

运算符则是一类特殊的函数，它以中缀的形式调用。但在定义运算符时，和定义函数时没有太大区别。

```
const <+> = funct (x : Int) -> funct (y : Int) -> x^2 + y^2;

const main = funct () -> do {
    print 1 <+> 2;
};
```

注意到这里我们让函数返回了一个函数。这没有任何问题，因为函数本身也是一个对象。所有类似于 `* -> * -> *` 的函数都称为*运算符*，它需要连续进行两次调用才能得到一个确切的值。从这个角度来看，`a @ b`（其中 `@` 是一个运算符）实际上是 `@ a b` 的**语法糖（Syntax Sugar）**，编译器会在编译时将其转化为实际的形式。

无论是函数还是运算符，它们在 AutoScript 中的地位都和普通的对象无异：我们可以将其作为函数参数传递。

```
const transform_twice = funct (f : Int -> Int, x : Int) -> f(f(x));
const operate_reverse = funct (op : Int -> Int -> Int, x : Int, y : Int) -> op(y, x);
const <-> = funct (x : Int, y : Int) -> x - 2*y;

const main = funct () -> do {
    print transform_twice(funct (x : Int) -> x^2, 2);       // 输出 16
    print operate_reverse(<->, 24, 42);                     // 输出 -6
};
```

上面的类型 `Int -> Int` 表示了接受一个整数并返回一个整数的函数类型，可以看到 `->` 在类型中的位置对应了函数声明时 `->` 的位置。

[4]后面我们会学习特殊的情况，但这里先认为参数必须严格遵守函数定义

1.6 元组和结构类型

AutoScript 中可以利用**元组（Tuple）**类型将多个类型按顺序排列在一起。元组可以通过逗号运算符 `,` 构建。

```
const main = funct () -> do {
    auto tup = (42, "abc");
    print tup[0];               // 输出 42
    print tup[1];               // 输出 abc
};
```

逗号运算符会将两个对象并列存储在一起，并返回这个元组对象。出于良好的习惯，此后所有的元组都会用圆括号包围[5]。从上面的代码中可以看到，我们可以通过 `t[N]` 的形式来取得元组的某个元素，这是一个特殊的调用语法，让我们在后续章节再详细介绍[6]。

AutoScript 中，可以使用**结构（Structure）**来定义元组的模版；在结构中，我们可以为元组的每个元素命名。

```
const Point = struct (x : Real, y : Real);

const main = funct () -> do {
    auto p : Point = (1.0, 0.0);
    print p.x;                  // 访问第一个元素
    print p.y;                  // 访问第二个元素
};
```

可以看到，结构也可以声明为一个变量。在定义特定结构类型的变量时，需要显式写出他的类型。结构类型的变量可以通过特殊的调用语法来访问它的元素，比如上面的 `p.x` 就是访问了 `p` 中被命名为 `x` 的元素。在结构章节中，我们会深入介绍 `p.x` 的本质。

一些结构上适合定义**方法（Method）**，所谓方法是绑定在特定对象上的函数。

```
const Point = struct (x : Real, y : Real);
const reset = funct (this p : ref mut Point) -> do {
    p.x <- 0.0;
    p.y <- 0.0;
};

const main = funct () -> do {
    auto p : mut Point = (1.0, 0.0);
    p.reset();          // 相当于 reset(p)
    print p.x;          // 输出 0.0
};
```

上面通过用 `this` 修饰参数来让函数能通过符号 `.` 的方法调用，这也是一个语法糖。在看到 `p.reset` 时，编译器会生成一个表达式，其可以通过空对象 `()` 得到我们想要的结果。一个函数中至多只能有一个参数被 `this` 修饰。下面的代码展示了编译器对拥有 `this` 参数的函数的处理：

[5] 这是因为逗号运算符的优先级比较低，容易被其它运算符影响

[6] 实际上，`t(N)` 也完全可以得到相同的结果，但为了符合主流语言的习惯，我们会使用方括号来取元素。

```
const Point = struct (x : Real, y : Real);
const reset = funct (this p : ref mut Point) -> do {
    p.x <- 0.0;
    p.y <- 0.0;
};

// 编译器会生成类似下面的代码
const compiler_generated_reset = funct (p : ref mut Point) -> funct () -> do {
    p.x <- 0.0;
    p.y <- 0.0;
};

const main = funct () -> do {
    auto p : mut Point = (1.0, 0.0);
    p.reset();

    // 编译器会生成类似下面的代码
    compiler_generated_reset(p)();
};
```

1.7 标准库

编写 AutoScript 程序时，标准库是非常有用的帮手。我们不需要从头手动编写许多工具，因为它们已经在标准库中存在了。

下面我们罗列一些常用的标准库部分和它们的作用。

- `Prelude` 库：其中包含了许多 AutoScript 必须的组件，包括 `Prelude.Bool` 类型、`Prelude.Proxy` 类型等。这个库会默认被 `import` 到每个 AutoScript 文件中并内联，即我们可以直接使用 `Bool` 来表示 `Prelude.Bool`。
- `Math` 库：其中包含了常用的数学函数，如 `Math.max`、`Math.sin` 等。
- `System` 库：其中包含了和系统底层交互的函数，包括一些和 C/C++ 接口交互的函数。
- `Containers` 库，其中包含了许多常用的容器和数据结构。
- `Meta` 库，其中包含了许多和反射相关的组件。

如果想要引入某个标准库，需要使用 `import` 关键字。它会将给定模块导入到当前作用域中。

```
import Math;                                  // 导入 Math 模块，现在 Math 在这个文件中可见了

const main = funct () -> do {
    print Math.sin(42)^2 + Math.sin(42)^2;  // 输出 1
};
```

本章中我们暂时不对上面提到的标准库详细介绍；后续章节在用到其中的组件时再进行解释。

1.8 其它特性

1.9 一个词频统计程序

首先，我们需要指定文件中读取文本。

```
import System.IO;

const read_from = funct (filename : String) -> do {
    auto file = IO.open(filename, "r");
    if (file == undefined) {
        throw "Cannot open file.";
    }
    return file.text();
};
```

这个返回的文本拥有 String 类型，因此我们可以利用 split 方法来根据空格或标点符号将其分开。

```
const get_words = funct (this text : String) -> do {
    const delim = funct (ch : String) -> is_space(ch) or is_punct(ch);
    return text.split(delim);
};
```

随后，利用 HashTable 容器来统计每个单词出现的次数。

```
const count_words = funct (this words : Vector(String)) -> do {
    auto table : mut HashTable(String, Int) = default;
    for (w : String from words) {
        if (not table.contains(w)) {
            table[w] <- 0;
        }
        table[w] <- table[w] + 1;
    }
    return table;
};
```

随后，我们需要关于 HashTable 键值对中的值进行排序。为此可以使用 PriorityQueue 容器。

```
const comparator = funct (x : (String, Int), y : (String, Int))
    -> x[1] < y[1] or (x[1] == y[1] and x[2] < y[2]);

const sort_counts = funct (this table : HashTable(String, Int)) -> do {
    auto queue : mut PriorityQueue((String, Int), comparator) = default;
    for (e : (String, Int) from table) {
        queue.push(e);
    }
    return queue;
};
```

最后，可以根据输入的数量来输出前几位词频的单词。

```
const most_frequent = funct (queue : PriorityQueue((String, Int), comparator), k : Int) ->
    do {
    auto words : mut Vector(String) = default;
    auto i : mut Int = 0;
    while (i < k and not queue.is_empty()) {
        words.append(queue.top()[0]);
        i <- i + 1;
    }
    return words;
};
```

现在利用上面所有的函数，我们可以写出下面的程序：

```
const main = funct () -> do {
    print "Input the text file to read: ";
    scan filename : String;
    print "Input the k for the most frequent k words: ";
    scan k : Int;

    auto words = read_from(filename)
                .get_words()
                .count_words()
                .sort_counts()
                .most_frequent(k);
    for (w : String from words) {
        print w;
        print "\n";
    }
};
```

第二章 变量和类型

内容提要

- 基本类型
- sizeof 函数
- 字面量
- 变量
- 名字
- 关键字
- 阐述关键字
- 作用域
- 存储期
- 可变性
- 别名
- 复合类型

2.1 基本类型

AutoScript 中内置了下面几种**基本类型（Fundamental Type）**：

- `Int`：整数类型，表示特定范围内的整数。
- `Real`：实数类型，表示特定范围内的实数，是双精度浮点数类型。
- `String`：字符串类型，表示一段文本内容。
- `Byte`：字节类型，表示一个字节。
- `Void`：空类型，没有任何信息。也可以写成 `()`。
- `Type`：类型的类型，表示一个类型。

大多数其它类型都是建立在这些基本类型上的。基本类型的特征是它们有比较清晰和原子的内存结构，下面是这些类型占用的内存大小：

`Int`	8 字节
`Real`	8 字节
`String`	16 字节
`Byte`	1 字节
`Void`	0 字节
`Type`	8 字节

表 2.1: 基本类型的大小

我们可以用内置的 `sizeof` 函数来得到类型的内存占用 [1]。

```
const main = func () -> do {
    print sizeof(Int);      // 输出 8
    print sizeof(Void);     // 输出 0
};
```

由于 `Int` 和 `Real` 类型的大小有限，因此它们只能表示特定范围的数，当试图让其持有超过界限的数时，会产生错误。

```
const main = func () -> do {
    auto err1 : Int = 10^100;       // 编译期错误，10^100 的结果无法放入一个 Int 变量中
    scan err2 : Int;                // 输入 10'000'000'000，运行期错误
};
```

[1] 和 C++ 不同的是，不能将 `sizeof` 用在普通的对象上以获得其类型的大小：毕竟类型本身也是一个对象，如果 `sizeof(42)` 等同于 `sizeof(typeof(42))`，那么 `sizeof(Int)` 会出现歧义。

可能令你疑惑的是，字符串的长度明明多不相同，但 `String` 却能有确定的大小。这是因为 `String` 和其它基本类型不同，它拥有一层 **间接性（Indirection）**，我们直接访问到的是其信息实际存储地址的句柄[2]。具体的内容我们会在引用类型的小节中说明。

对其它语言有一些了解的读者可能发觉上面这些基本类型并不够用，比如没有表示真假的 `Bool` 类型，以及没有空间高效的短整数和单精度浮点数。这些类型都放在 `Prelude` 库中，它们的定义如下（我省去了它们名字中的 `Prelude`）：

- `Bool`：1 字节的布尔类型，只有两个可能的值：`True` 和 `False`。
- `BigInt`：是 8 字节的无限整数类型，（理论上）可以存储任意大的整数。
- `Int64`：是 `Int` 的*别名*，即所有 `Int64` 都和 `Int` 完全等同。
- `Int32`：4 字节的短整数类型。
- `Int16`：2 字节的短整数类型。
- `Int8`：1 字节的短整数类型。
- `BigReal`：是 8 字节的无限精度实数类型，（理论上）可以存储任意的有理数。
- `Real64`：是 `Real` 的*别名*。
- `Real32`：4 字节的单精度浮点数类型。

值得特别说明的是不同类型之间的*兼容性*。从直觉来看，一个整数总能无损失地转换成一个实数，但考虑到精度，一些过大的整数显然不能顺利地变成实数；不过，低精度的整数或实数显然可以安全地转换成对应的高精度类型。

AutoScript 中用两个名词来描述这两种情况的类型转换，对于总是安全的类型转换，其名称为**隐式类型转换（Implicit Type Conversion）**。正如其字面暗示的，编译器会自动为你做合适的转换；剩余的则是**显式类型转换（Explicit Type Conversion）**，我们必须用某种方式告知编译器这个转换的确是我们的意图。

```
cons main = funct () -> do {
    auto x : mut Int = 42;
    auto y : mut BigInt = 0;    // 隐式类型转换
    y <- x;                     // 隐式类型转换
    x <- y as Int;              // 显式类型转换，如果省略 as BigInt 会出现编译错误
};
```

上面出现的 `as` 运算符会调用特定类型的*构造函数*来构建目标类型的对象（`as` 是一个运算符）。值得一提的是，初始化变量时 `=` 右侧的表达式只能经过隐式转换变为声明的类型。

```
const main = funct () -> do {
    auto x : Int = 42 as BigInt;    // 编译错误，BigInt 无法隐式转换为声明的 Int 类型
};
```

一些类型之间不存在类型转换，比如 `Int` 和 `String`。

```
const main = funct () -> do {
    auto x = 42;
    auto y = "abc";
    x <- y as Int;      // 编译错误
    y <- x as String;   // 编译错误
};
```

[2] 也就是 C++ 中的指针。

这是因为两者间不存在显然的转换关系："123abc" 如何转换成整数？反过来 1e8 应该转换成什么格式的字符串？因此更好的方法是借用函数丰富的参数列表来自定义转换时的行为。Prelude 中提供了 as_string 和 as_real 等函数用于这些类型间的转换。

```
const main = funct () -> do {
    print 42.as_string();    // 输出 42
    print "42".as_int();     // 输出 42
};
```

2.2 字面量

字面量（Literal）是指类似于 42 这样从字面上就能理解其值的结构。在 AutoScript 中，字面量的类型主要有下面几种：

- 整数字面量：表示一个整数，主要有下面四种形式：
  - 十进制整数：以数字组成的字面量，如 42。
  - 十六进制整数：以 0x 开头的，包含所有数字和 A-F 的字母的字面量，比如 0xCAFE。
  - 二进制整数：以 0b 开头的，包含数字 0 或 1 的字面量，比如 0x110。
  - 任意进制整数：以 0#[N] 开头的（其中 N 是一个不超过 36 的正整数），比如 0#[7]66。
- 实数字面量：表示一个实数，主要有下面几种形式：
  - 包含一个小数点，且小数点两边都是整数字面量，如 1.23。
  - 包含一个指数标识符 e 或 E，标识符左侧是一个整数或实数字面量，右侧是一个整数字面量，如 1e10。
- 字符串字面量：表示一个字符串，主要有下面几种形式：
  - 转义字符串字面量：表示一个字符串，由一对双引号 " 作为边界的字面量，比如 "\nHello"!。这种字符串中定义了一系列转义字符，详细介绍看下文的表格。
  - 原始字符串字面量：表示一个字符串，由三对双引号作为边界的的字面量，比如 """\ is a backslash."""
- 字节字面量：表示一个字节，由一对单引号 ' 作为边界的字面量，比如 'a'。
- 空字面量：即 ()，表示一个 Void 类型的对象。注意到 () 同时也是它的类型。
- 特殊字面量：包括 default、missing、otherwise 和 undefined。它们没有类型，只是一个占位符；编译器会适宜地处理它们。
- 自定义字面量：本质上还是整数、实数或字符串字面量；不过在字面量的最后会加上一个*后缀*，实际效果等同于一个函数调用，比如 123m。

上面提到的 **转义字符（Escape Character）**，这是一些因为特殊原因不能在字符串中直接写出来的字符：

新一行	\n	水平制表符	\t	单引号	\'
垂直制表符	\v	退格	\b	美元符号	\$
反斜杠	\\	响铃	\a	Unicode 符号	\u
回车	\r	双引号	\"	Unicode 符号	\x

表 2.2: 转义字符

其中，Unicode 符号 \u 后需要接一个八进制整数字面量，而 \x 后需要接一个十六进制整数字面量。如果两者的位数过长（\u 最多接受三位，\x 最多接受四位），则只会取前面几位。

字符串字面量支持**插值（Interpolation）**，即在字符串中使用一个表达式。可以通过符号 $ 和圆括号来引入一个表达式。表达式中不能再次出现字符串字面量。

```
const main = funct () -> do {
    print "Enter your name: ";
    scan name : String;
    print "Hello, $(name)";
};
```

所有的基本类型都可以进行插值。后面结构章节中我们再介绍如何让自定义的类型也能插值到字符串中。

有时候我们会使用**原始字符串（Raw String）**。在原始字符串中，所有常规的转义字符都不再生效：字符串中出现了什么就代表什么。字符串插值依然有效，不过这里 `$` 的转义字符变成了 `$$`。即我们要通过 `$$` 来表示一个美元符号。

```
const main = funct () -> do {
    auto x = 42;
    auto raw =
    """
    A new line indented once.
        Tabbed twice.
    Interpolation is supported: x = $(x).
        $$ is for a single dollar symbol.
    """;
};
```

特殊字面量是一些可以灵活使用在各种场景的字面量，这里详细介绍其中的两种。

`default` 表示一个类型的默认值。对于内置的类型来说，其默认值如下：

- `Int`：默认值为 0。
- `Real`：默认值为 0.0。
- `String`：默认值为空字符串 `""`。

如果直接打印 `default`，我们不会得到任何内置类型的值；不过在定义变量时，可以通过 `default` 将其初始化为上面列出的默认值。

```
const main = func () -> do {
    print default;              // 输出 default
    auto x : Int = default;
    print x;                    // 输出 0
};
```

`undefined` 则有特殊的含义：它代表了所有未定义变量的值。任何对它的求值都会产生异常。

```
const main = func () -> do {
    print undefined;            // 运行期错误，对 undefined 求值
    auto x : Int;               // x 当前的值是 undefined
    print x;                    // 运行期错误，对 undefined 求值
};
```

2.3 变量

变量（Variable）是 AutoScript 程序的基石。定义变量时，我们将一个**实体（Entity）**绑定到一个名字上面，此后这个名字在特定范围（即*作用域*）内就代表了一个变量。在不引起混淆的情况下，我们会用“变量”一词来同时指它的名字和绑定的对象。

2.3.1 变量的声明

变量的声明结构比较复杂，这里让我们先介绍其中最简单的部分。

语法 2.1 (变量声明 [simple-var-decl])

1. [storage-spec] [var-name] : [type-spec] = [expr]
2. [storage-spec] [var-name] : [type-spec]
3. [storage-spec] [var-name] = [expr]

♣

注意变量声明的结构之后需要接一个分号才能形成一个完整的语句（即声明语句）。

```
const main = funct () -> do {
    auto x : Int = 42;
    auto y : Int;
    auto z = 42;
};
```

上面的例子中，x、y、z 的声明方式分别对应了语法 2.1 中的三种格式。下面来详细说明三者的异同：

1. x 经过初始化且类型是显式给出的，此时编译器会忠实地将 [type-spec] 作为 x 的类型，并尝试将 [expr] 的值隐式类型转换为这个类型。如果转换失败，则会产生编译错误。
2. y 没有初始化，此时它只是一个**提前声明（Forward Declaration）**。程序执行到这里时不会为 y 分配内存，相当于用 `undefined` 为其初始化。
3. z 经过了初始化但没有显式给出类型。此时 AutoScript 会利用**类型推断（Type Inference）**来决定 z 的类型。这里由于初始化表达式是 `Int` 类型的字面量，因此显然 z 的类型是 `Int`。

通常我们会采用第三种变量定义方式，即依赖 AutoScript 的静态类型系统自动推导变量的类型；在 IDE 的类型显示辅助下，这是最便捷且清晰的方法。

下面让我们详细介绍有关提前声明的知识。所有不是提前声明的变量声明都称为**定义（Definition）**。一个程序中只允许出现某个变量被定义一次，但它可以出现多次提前声明。所有声明应该是*兼容*的；具体地说，无论提前声明如何，需要保证变量在定义处显式给出或推断出的类型是每个提前声明处类型的*子类型*[3]。

需要提前声明的原因其实不难理解：在没有定义变量时，如果需要利用到这个变量，则至少要让编译器知道它的类型信息，从而方便提前进行类型推断。真正使用到这个变量时，需要保证其实际功能要强于提前声明处的类型声明，因此前者应是后者的子类型。让我们用一个例子来了解提前声明的作用：

[3]有关子类型将会在结构的章节中详细讲解，这里可以先给出其定义：如果类型 A 可以出现的地方 B 也能出现，则后者是前者的子类型。在类型声明这个语境下，变量实际的初始化对象理应出现在所有声明类型可以出现在的地方，因此是前者是后者的子类型。

```
const b : Int -> Int;
// a_n = b_n + b_{n-1}
// a_0 = 0
const a = funct (n : Int) -> do {
    if (n != 0) {
        return b(n) + b(n-1);       // 通过提前声明确认这是 Int 类型
    }
    return 0;
};
// b_n = a_n - a_{n-1}
// b_0 = 0
const b = funct (n : Int) -> do {
    if (n != 0) {
        return a(n) - a(n-1);
    }
    return 0;
};
```

上面的例子中，函数 a 和 b 之间需要相互调用，因此必定需要有一方被提前声明。在 b 被提前声明后，编译器就知晓了它的类型，因此在 a 的 return 语句中可以推断出 a 的返回类型。这里值得对 AutoScript 中的类型推断机制进一步说明。

AutoScript 拥有强大的静态类型系统，因此在编译期可以得知大部分变量的类型信息。根据这些类型信息我们可以推断出表达式的类型（比如 1 + 1 的类型显然也是 Int），因此不仅对象的类型可以通过字面量来推断得知，函数的类型也可以通过函数中表达式的组合来推断出来。凭借类型推断，许多地方我们并不需要写出变量的类型[4]。

2.3.2 名字与作用域

变量的一个重要属性就是它的**名字（Name）**。不过，并非所有名字都是合法的：AutoScript 保留了一些名字用作语法标记，这些名字称为**关键字（Keyword）**。下面的几张表分类罗列了 AutoScript 中所有的关键字。

首先是用作程序流程控制的关键字：

assert	continue	for	print	throw	yield
assume	do	guard	requires	try	
break	delete	if	return	where	
case	else	match	scan	while	
catch	fall	monad	then	with	

表 2.3: 控制流关键字

其次是用作函数或运算符的关键字：

[4]一些来自于静态类型语言的读者可能不习惯广泛利用类型推断，但这并非洪水猛兽：在现代 IDE 或语言分析插件的助力下，我们可以尽管将背后交给类型推断系统。

addrof	bitor	exceptof	keyof	of	shr
and	decay	hasbase	kindof	or	tagof
as	decayof	hasclass	nameof	protoof	templof
bitand	deref	in	next	sizeof	typeof
bitnot	derefof	instof	not	shl	xor

表 2.4: 运算符/函数关键字

然后是作为“语法胶水”的阐述关键字，这些关键字都可以被特定符号代替：

alias（$）	equals（=）	object（#=）
auto（缺省）	from	static（'）
class（?=）	funct（\）	struct（#）
const（!）	is（?）	thread
data（%）	new（^）	type（:）

表 2.5: 阐述关键字

随后是一些不能被符号替代的关键字：

atom	except	internal	mut	undefined
await	export	lazy	otherwise	
comptime	extend	local	private	
default	import	missing	ref	
dyn	infer	move	this	

表 2.6: 其它关键字

除此之外还有一些只在特定上下文中有特殊含义的**上下文关键字（Context Keyword）**。

branch	iteration
choice	loop
excluding	noreturn
including	
intrinsic	

表 2.7: 上下文关键字

最后，所有包含两个连续下滑线的名字都是**保留字（Reserved Word）**，它们会用作编译器生成的代码。虽然定义这样的变量不一定发生错误，但可能会让程序有未定义行为。

和变量息息相关的一个语言特性是**作用域（Scope）**，它用于隔离变量的作用范围，并合理地分隔代码。所有出现大括号的地方都会引入一个新的作用域，比如我们已经熟悉的函数体。作用域内部可以无障碍地访问外面定义的变量，但反过来通常不行。不同的作用域中可以定义相同名字的变量；特别地，如果在内部作用域定义和外部相同名字的变量，则会发生**变量覆盖（Variable Hiding）**，此时在内部作用域中只能访问内部定义的变量。

```
const main = funct () -> do {
    auto x = 42;
    {
        print x;        // 输出 42
        auto x = 24;
        print x;        // 输出 24
    }
    print x;            // 输出 42
};
```

每个变量都处于一个作用域当中。最外面的作用域被称为**全局作用域（Global Scope）**，函数体中出现的任何作用域都被称为**局部作用域（Local Scope）**。除此之外的作用域我们暂时不会遇到，因此暂不详述了。

之所以不同作用域中可以定义同名的变量，是因为编译器对每个名字进行了特殊的处理。全局作用域的名字不会有任何改变，但以 `main` 函数中的变量为例，`x` 会被改写成 `::main::unnamed0::x`，而内部作用域中的 `x` 会被改写成 `::main::unnamed0::unnamed1::x`。这些名字之间显然没有冲突。我们会在名字空间的章节中详细介绍这个特性。

2.3.3 存储期

本节让我们介绍 AutoScript 中的一个重要语言特性，即变量的**存储期（Storage Duration）**。它决定了变量的生命周期，即什么时候被分配内存，以及什么时候被释放。有下面几种存储期：

- 自动存储期：用关键字 `auto` 表示。在变量定义时分配内存[5]并初始化，并在作用域结尾处释放内存。
- 静态存储期：用关键字 `static` 表示。在程序开始时分配内存，在变量定义时初始化，并在程序结束时释放内存。
- 线程存储期：用关键字 `thread` 表示。在线程开始时分配内存，在变量定义时初始化，并在线程结束时释放内存。
- 常量存储期：用关键字 `const` 表示。在编译时分配内存并初始化，有时会被编译器优化掉。
- 动态存储期：用关键字 `new` 表示。在变量定义时分配内存并初始化，一些情况下需要手动通过 `delete` 函数释放内存。

从习惯上来讲，函数体中大多数的变量都应拥有自动存储期，这是最简单且安全的方法；静态存储期适用于存储某个全局的状态；常量存储期适用于声明函数和类型，以及元编程等在运行期不会改变的信息，是最高效但局限性最强的；动态存储期则适用于动态类型编程，是最灵活但却不够高效的。下面给出一些使用的例子。

```
const store_add = funct (x : Int) -> do {
    static res : mut Int = 0;       // 静态存储期，只在初次执行时初始化
    res <- res + x;
    return res;
};
const main = funct () -> do {
    print store_add(1);     // 输出 1
    print store_add(2);     // 输出 3
};
```

这个例子中，我们在函数 `store_add` 中定义了静态变量 `res`，它只会在第一次遇到时初始化，此后会在多次调用间保持一致。

[5]实际上变量在栈上的内存在函数开始时就已经全部分配好了，不过由于它没有初始化，因此其在堆上的内存空间需要在变量定义处分配。

```
const compile_time_add = funct (const x : Int, const y : Int) -> x + y;
const main = funct () -> do {
    print compile_time_add(1, 2);        // 输出 3，但这个值在编译期就可以算出来
};
```

这个例子中，我们了解到 const 用来修饰函数的参数类型时，可以让函数在编译期计算出结果[6]。对比此前声明的函数，这个多出来的 const 其实正是变量声明时的存储期限定符；也就是说，函数参数本质上是一个变量声明。我们习惯的参数声明，以 x : Int 为例，本质上是 auto x : Int。只不过作为阐述关键字，auto 等同于空，而我们习惯在变量声明时加上 auto，在参数声明处则省略它而已。

2.3.4 可变性

此前我们就已经提到过，AutoScript 中所有的变量都默认不可变，因此为了能够更改某个变量，我们需要用到类型修饰符 mut。

```
const main = funct () -> do {
    auto x = 42;
    auto y : mut Int = 42;
    x <- 24;                // 编译错误，x 不可变
    y <- 24;                // 没问题
};
```

不过，实际上有一个更加简单的方式定义可变的变量，只需在表达式上直接用 mut 修饰即可。

```
const main = funct () -> do {
    auto x = mut 42;
    x <- 24;
};
```

AutoScript 致力贯彻声明和结构等价的语法，因此一个类型的构建方式（表达式形式）基本上都能对应它的类型语法。上面的例子中，虽然 x 的类型是 mut Int，但我们可以用 Int 为它赋值（隐式类型转换）。这实际上展示了 AutoScript 中定义和赋值的一个主要区别：定义时类型推断会忠实地遵守初始化表达式的类型，但赋值时会拥有更灵活的行为；这样的区别也是 AutoScript 采用不同符号表示这两个操作的原因之一[7]。

不过，让表达式总是何其类型结构相同的设计有时可能会造成不便。依然考虑 mut 的使用场景：

```
const main = funct () -> do {
    auto x = 42;            // x 类型是 Int
    auto y = mut 42;        // y 类型是 mut Int
    auto z = y;             // z 类型是 mut Int
};
```

这里的 x 和 y 都很直观地体现了它们的类型，然而 z 并非如此。当然，我们依然可以通过 auto z = mut y 同样构建一个 mut Int，但如果我们没有注意 y 的类型而直接写 auto z = y，那么就会误将其定义为一个可变变

[6] 事实上即使没有将它声明为 const，开启优化后的编译器依然会尽己所能将其提前到编译器进行运算，不过如果主动标上 const 则可以在编译期执行许多常量计算的检查，以确保它一定能在编译器计算出来。

[7] 绝大多数其它语言都让定义（初始化）和赋值使用相同的符号，这在一些情况下会引起概念上的困惑。比如 C++ 中的内存分配、初始化和赋值实际上是三件事情，但都融合到一个符号 = 上，这不容易在深入探究时缕清它们的关系。此外，C++ 中泛滥的隐式类型/类别转换让这些操作变得更加不透明。

量。下面是几种解决这个问题的方法：

- 显式写出变量的类型。
- 利用模式匹配强行去除表达式最外层的 mut。
- 使用 demut 关键字。

这里第二种让我们在模式匹配章节中再介绍。第三种可以参考下面的例子：

```
const main = funct () -> do {
    auto x = mut 42;
    auto y = demut x;        // y 的类型是 Int
};
```

简单来说，demut 函数将任何 mut 修饰的对象都变成不带修饰的版本，且其中并不会引入新的对象。

2.3.5 别名

别名（Alias）是一类特殊的名字，通过关键字 alias 声明，它们不属于实体，因为它们不会得到任何内存空间。从定义完毕之后，它们在任何时候都与其引用的名字性质相同。

```
const main = funct () -> do {
    auto x = 42;
    auto y = mut 42;
    alias also_x = x;        // also_x 是 x 的别名
    alias also_y = y;        // also_y 是 y 的别名
    also_x <- 24;            // 编译错误，also_x (aka x) 是不可变类型
    also_y <- 24;
    print y;                 // 输出 24
};
```

除了为绑定在对象上的变量起别名外，我们也可以为函数和类型起别名。

```
const main = funct () -> do {
    alias MyInt = Int;
    auto x : MyInt = 42;
};
```

别名常用于模块的引入：

```
alias MyMod = import Some.Really.Long.Name; // 右侧导入了一个模块名，然后定义 MyMod 为其别名
```

除了可以用来简写一些冗长的名字，别名在宏编程中也有重要的作用，这里不详述了。

2.4 复合类型

从基本类型出发，我们可以定义复合类型。AutoScript 中提供了丰富的复合类型的定义方式，包括元组类型、选择类型、延展类型等。下面让我们一一介绍。

2.4.1 元组与数组

元组（Tuple）可以通过逗号运算符构造，它是一系列元素的连续排列。

```
const main = funct () -> do {
    auto tup = 42, "abc";
    print typeof(tup);      // Int, String
};
```

这里我们使用了函数关键字 typeof 来得到变量 tup 的类型。可以看到，元组的类型和它构造的结构是完全相同的，因此很好理解。不过，在实际使用时，通常会为元组的两侧添上一对圆括号。这是因为逗号运算符的优先级很低，因此容易受到其它运算符的干扰。

元组上定义了一些实用的方法，简单说明如下：

- size：得到元组中元素的个数。
- []：元组可以对查询结构进行调用，用来访问元组中特定位置的元素。
- types：得到元组中每个元素类型组成的数组。

下面是一个示例：

```
const main = funct () -> do {
    auto tup = (42, "abc");
    print tup.size();       // 输出 2
    print tup[1];           // 输出 abc
    print tup.types()[0];   // 输出 Int
};
```

数组（Array）是元组中的特例：当元组中所有元素的类型都相等时，我们就得到了一个数组。

```
const main = funct () -> do {
    auto arr = (1, 2, 3);   // 长度为 3 的数组
    print typeof(arr);      // 输出 Int, Int, Int
};
```

不过，这样的设计会让数组类型变得非常冗长（想象一下长度为 1024 的数组）。因此在 Prelude 中定义了类型对查询结构的调用函数。我们只需要用 T[N] 来表示长度为 N 的 T 数组即可[8]。

```
const main = funct () -> do {
    auto arr : Int[8];
    auto ct = mut 8;
    while (ct > 0) {
        ct <- ct - 1;
        print "Enter the next number, $(ct) left: ";
        scan arr[7-ct];
    }
    print arr;
};
```

和所有其它对象一样，元组在默认情况下也是不可变的。利用查询结构得到的元组中的元素也是不可变的。如果想要修改元组，可以将其声明为 mut。

[8]不同于一些其它语言中将 [] 视为一个运算符的设计，AutoScript 中的 [] 声明了一个结构，其拥有特别的语义。我们会在函数的章节中详细介绍。

```
const main = funct () -> do {
    auto tup = mut (1, 2);
    tup[0] <- 42;
    print typeof(tup);          // 输出 mut (Int, Int)
    print typeof(tup[0]);       // 输出 mut Int
};
```

对于元组类型，对其使用 mut 修饰符后，它的元素都将拥有 mut 的性质，因此上面的 tup[0] 的类型是 mut Int，这也被称为 mut 的*传递性*。一个可变元组类型中不可能包含不可变的元素，不过反过来是可以的，比如我们可以声明类型为 (Int, mut Int) 的对象[9]。

元组类型的大小理论上是所有元素大小的和，但考虑到*内存对齐*的因素，其大小实际上通常是 8 的倍数。

```
const main = funct () -> do {
    print sizeof(Int, String);      // 输出 24
    print sizeof(Byte, String);     // 输出 24
    print sizeof(Byte, Byte);       // 输出 2
};
```

有关内存对齐的内容让我们在后期章节再详述。

2.4.2 选择

选择类型是一种多选其一的类型，这样的变量在每一时刻都是多种类型中的一个。

```
const main = funct () -> do {
    auto int_or_str : Int | String = 42;
    print typeof(int_or_str);           // 输出 Int | String
    print tagof(int_or_str);            // 输出 Int
};
```

这里用到的 tagof 函数关键字是用来判断选择类型变量的实际类型的，也称为**标签（Tag）**。选择类型中的任一个标签都可以被隐式转换为前者，但反过来需要进行显式转换。

```
const main = funct () -> do {
    auto int_or_str : Int | String = 42;    // 隐式类型转换
    print int_or_str as Int;                // 显式类型转换且成功
    print int_or_str as String;             // 运行错误，标签与实际不符合。
};
```

可以看到，有关选择类型的标签错误是一个运行期错误，这是因为编译器无法准确判断选择类型究竟拥有哪个标签。比如看下面的例子：

```
import Math;

const f = funct (x : mut Int | String) -> do {
```

[9]这一点可以理解成，如果元组本身是可变的，则其中某个元素根本无法保证不可变性（可以通过覆写整个元组来修改某个元素）；但一个不可变对象的局部完全可以是可变的（只需要保证其它的部分不可变即可）。C++ 的可变性默认与 AutoScript 正好相反，但其允许在（默认）可变的类型中声明一个不可变变量，这并非不合理：我们无法在赋值函数中直接修改这个不可变成员变量。AutoScript 中元组的赋值函数是自带的且其行为是字面的含义：将元组的值设为给定的新元组，因此某个不可变的元素会让这个行为变得模糊。

```
    auto flag = Math.randflag();
    if (flag) {
        x <- "abc";
    }
    print tagof(x);
};
```

在函数 f 中，首先我们接受了一个选择类型的参数，此时理论上可以根据调用处的情况来分类为输入 Int 和 String 的两种情况 [10]。但在函数中，我们根据一个随机值来判断是否为 x 赋值为 String 标签，这就让可能的情况再次一分为二。程序中，类似这样的语句还有很多，因此不可能在编译期判断选择类型的标签。从这个结论出发，我们知道 tagof 是一个运行期执行的函数。

选择类型中标签的顺序不影响其最终的类型，这是理所当然的。实际上，编译器会为所有类型进行某种排序，使得其标注的选择类型中的标签符合特定的顺序 [11]。

如果为选择类型使用 mut 修饰符，则其中无论是任何标签都是可变的；如果只想让某个标签可变，可以单独为其使用 mut 修饰符。背后的原因和元组类似。

此前在元组小节中介绍的查询结构调用，其返回值实际上就拥有选择类型。以 (Int, Real) 为例，其返回成员的类型就是 Int | Real，不过这个函数的定义是内置的。

选择类型的大小是其标签中最大的类型大小。

```
const main = funct () -> do {
    print sizeof(Int | String);          // 输出 24
    print sizeof(Byte | Int);            // 输出 8
};
```

2.4.3 引用

AutoScript 中支持**引用（Reference）**。这种类型的变量存储的是其它变量的地址，从而能够在函数间传递时依然持有引用目标的句柄 [12]。引用通过引用运算符 ref 构造。更通俗地解释，我们可以认为每个变量都有一个独特的编号，程序中可以通过编号访问它对应的变量。引用存储的正是这个编号。

```
const main = funct () -> do {
    auto x = mut 42;
    auto r = ref x;
    print typeof(r);        // 输出 ref mut Int
    r <- 24;
    print x;                // 输出 24
};
```

[10] 类似于 C++ 中的全特化模版函数，不过这样会增加程序的二进制大小。

[11] 出于方便，我们可以假设编译器采用辞典顺序排序，因此 String | Int 会被正规化为 Int | String；但这并不是一定的：所有依赖于选择类型的正规排序方式的代码都会产生未定义行为。

[12] 也就是 C++ 中的指针。

上面的代码中出现了神奇的景象：明明 r 的类型是 ref mut Int，为什么可以通过 Int 对其赋值？实际上，编译器在处理赋值操作时会对两边的类型进行判断：如果左侧是一个引用类型，则会尝试将右侧的类型隐式转换成左侧引用的类型。如果成功则会将左侧解引用为引用的变量，然后进行赋值操作。

至此就会出现一个有趣的现象：即使一个变量不是可变的，它依然可以进行赋值操作，因为它可能是一个可变对象的引用。下面用一张表来展示对于类型 T，它和它的引用在使用可变修饰符时相互的可赋值性。

从右到下	T	mut T	ref T	mut ref T	ref mut T	mut ref mut T
T	错误	错误	错误	错误	错误	错误
mut T	OK	OK	OK	OK	OK	OK
ref T	错误	错误	错误	错误	错误	错误
mut ref T	错误	错误	OK	OK	OK	OK
ref mut T	OK	OK	错误	错误	错误	错误
mut ref mut T	OK	OK	OK	OK	OK	OK

表 2.8: 引用类型赋值

如果要简单概括这张表的信息，那就是一切都以不可变为准：如果在类型某个位置不可变，则相应位置就不能被修改；如果引用的变量可变，则它可以转换为引用不可变的（如 mut ref mut T 到 mut ref T 的转换）[13]。

```
const main = funct () -> do {
    auto i = 42;
    auto mi = mut 42;
    auto mri = mut ref i;
    auto mrmi = mut ref mi;

    mri <- mrmi;            // 没问题，mri 现在引用了 y，但不能修改 y
    mrmi <- mri;            // 编译错误
};
```

引用的重要性在于，它是唯一通过修改函数参数影响外部状态的方式。

```
const swap = funct (x : ref mut Int, y : ref mut Int) -> do {
    auto tmp = deref(x);
    x <- deref(y);
    y <- tmp;
};
```

上面用到的函数关键字 deref 用来将引用转化为值。deref 不一定能运算成功，这是因为引用可能赋值为 default，此时也就是所谓的**空引用（Null Reference）**。对于空引用，deref 函数会导致运行错误。

```
const main = funct () -> do {
    auto null : ref Int = default;
    print deref(null);          // 运行错误，尝试解引用一个空引用
};
```

引用类型的大小固定为 8。

[13] 这里并不会印象所引用变量的可变性，而只是从这个引用来看，它不能修改所引用变量的内容。可以类比 C++ 中 int* 到 int const* 的隐式类型转换。

2.4.4 函数

函数（Function）我们至此已经有不少的使用经历了，但一直没有介绍过它的类型。简单来说，函数是所有通过箭头运算符 `->` 构建的表达式模版。箭头的左侧是一个**参数列表（Parameter List）**，其表现为零到多个变量声明，而右侧是一个表达式，它可以使用左侧声明的变量。不过，为了编译器的方便，避免语法分析的歧义，AutoScript 要求函数在表达式中出现时必须使用阐述关键字 `funct` 来提示它是一个函数[14]。

```
const add = funct (x : Int, y : Int) -> x + y;
const partial_add = funct (x : Int) -> funct (y : Int) -> x + y;
```

上面两种函数的类型分别是 `(Int, Int) -> Int` 和 `Int -> Int -> Int`。在第一章中我们提到过，前面这种是狭义上的函数，而后面这种称为运算符。实际上，如果我们在构建函数时不断增加 `->` 运算符，可以理论上构建出任意长的函数链；但习惯上我们几乎只会定义上面这两种，即 `* -> *` 或 `* -> * -> *` 这种类别的函数。从构造来看，`->` 应该是一个右结合的运算符，因此 `Int -> Int -> Int` 应该理解为 `Int -> (Int -> Int)`，即返回一个函数的函数。下面展示两者使用的差异：

```
const add = funct (x : Int, y : Int) -> x + y;
const partial_add = funct (x : Int) -> funct (y : Int) -> x + y;

const main = funct () -> do {
    auto sum_1 = add(1, 2);
    auto add1 = partial_add(1);
    auto sum_2 = add1(2);
};
```

可以看到，`partial add` 可以先接入一个参数，并返回一个函数（正如它的定义暗示的），然后我们可以为这个函数再输入一个参数，并返回结果。运算符这样的性质让它们可以轻易地复用自己的定义，这在一些函数中变量初始化需要较大开销的场景中颇为有用。

```
const random_int = funct (type : EngineType) -> do {
    initiate_engine : EngineType -> Engine;
    static engine = initiate_engine(type);
    return funct (...args) -> engine(...args);
};
```

上面的例子中用到了范型参数包 `...args`，可以先简单理解为可以接受零到多个不定的参数，并传递给 `engine`，我们会在后面介绍模式匹配时详细介绍这个语法。

函数区别与此前介绍的各种复合类型在于，它没有确定的内存布局。事实上在许多语言中，函数和其它类型是区别对待的。AutoScript 消除了对函数的特殊照顾，并让它拥有确切的类型，并可以绑定为一个变量。如果从实现来看，函数的本质是一个内存地址，因此其大小和引用类型相同，固定为 8。

[14]除非它此前已经被声明拥有函数类型。

第三章 表达式

内容提要

- 表达式
- 运算符
- 优先级
- 结合性
- 运算符字面量
- 表达式修饰符
- 常量表达式
- 惰性表达式
- 表达式异常

3.1 表达式语法

表达式（Expression）是 AutoScript 中所有能被求值的结构，以变量、字面量、运算符、函数调用相互嵌套组成。所有的运算符都可以分类为下面的几种形式：

语法 3.1 (表达式)

1. [atom-expr]
2. [compound-expr]
3. [expr-1] [expr-2]
4. [expr-1] [op] [expr-2]

它们对应的含义如下：

1. 对应了一个变量或字面量。
2. 对应了一个复合表达式。
3. 对应了一个函数调用表达式，其中 [expr-1] 是可调用的表达式，而 [expr-2] 是参数。
4. 对应了一个运算符调用表达式，其中 [expr-1] 是左侧表达式，[expr-2] 是右侧表达式。

3.1.1 原子表达式

原子表达式是 AutoScript 中最小的结构。它的类型就是变量或字面量的类型，值就是变量或字面量的值。

```
const main = func () -> do {
    auto x = 42;                // 42 是一个原子表达式
    auto y = x;                 // x 是一个原子表达式
};
```

3.1.2 do 表达式和 monad 表达式

AutoScript 中的复合表达式中有两种尤其重要，它们是 do 表达式和 monad 表达式。

语法 3.2 (do 表达式)

```
do [block]
```

语法 3.3 (monad 表达式)

```
monad [class-id] [block]
```

这些表达式的值与类型由其中的 return 或 yield 等语句决定。

```
const main = funct () -> do {
    auto z = do {
        scan x : Int;
        return Math.max(x, 0);
    };                          // do { ... } 是一个 do 表达式
    auto w = monad Option {
        return 42;
    };                          // monad MonadClass { ... } 是一个 monad 表达式
};
```

其中 do 表达式和 monad 表达式默认要求所有 return 语句后的类型完全相同[1]。不过，如果想要让 do 表达式同时接纳所有可能的返回类型（即使用选择类型），则需要在 do 的表达式中使用 choice 属性[2]：

```
const f = funct (x : Int) -> do choice {
    if (x < 0) {
        return 42;
    }
    if (x < 100) {
        return "abc";
    }
    return True;
};
const main = funct () -> do {
    print typeof(f(0));     // 输出 Int | Prelude.Bool | String
};
```

我们会尽量避免使用 choice 属性。

另一边，monad 表达式会对 return 语句的返回值进行一个包装。以 Prelude.Option 为例，在 monad Option 表达式中，所有的返回值都会尝试构造成 Option。

```
const main = funct () -> monad Option {
    scan x : Int;
    if (x > 0) {
        return x;
    }
    return Null;
};
```

上面的 Option 是一个范型抽象数据类型，我们可以通过特定类型的实体或 Null 来构建。我们会在单子的章节详细介绍这个表达式。

[1] 如果没有提前声明返回类型，则会将第一个 return 或 yield 语句作为推断依据。随后出现的这些返回值需要能够隐式转换为推断出来的类型。

[2] 如果编译器发现所有 return 语句后的表达式类型相同，则会推断返回类型为一个普通的类型。不要滥用 choice 的便利性，它可能会让函数类型推断变得更加复杂。

3.1.3 调用表达式

AutoScript 中的调用表达式指的是函数调用，其中特别定义了一些关键字用于函数调用，如我们已经见过的 `sizeof`、`typeof` 等。运算符的调用也算在调用表达式中，因此内置的 `as` 表达式也属于调用表达式。此后我们会遇到更多这样的关键字。

调用表达式是右结合的，因此 `f g x` 会被理解为 `f (g x)` 而非 `(f g) x`，这也是为了让内置的这些函数调用起来更加简便（比如可以用 `sizeof typeof(Int)` 而非 `sizeof(typeof(Int))`[3]。

在调用表达式中出现的括号并不是其必须的成分，其作用只在于表达式的分组，和其它类型表达式中出现的括号一致；只不过，在一些情况下我们不得不使用调用表达式来让程序行为和预期一致：比如 `f(1, 2)` 用于调用一个接受一个两元组的函数 `f`，但表达式 `f 1, 2` 则会尝试调用 `f 1`，随后和 `2` 形成一个两元组，含义迥然不同。

3.1.4 match 表达式

AutoScript 中可以用 `match` 关键字来引入一个模式匹配表达式。不过目前我们还没有足够的知识来介绍这一类表达式，因此暂用一些简单的例子来体验 `match` 表达式。

```
const main = funct () -> do {
    scan x : Int;
    print match (x)
        ? 0  -> "Zero",          // 如果 x 的值是 0
        ? 42 -> "Good",          // 如果 x 的值是 42
        otherwise -> "Bad";      // otherwise 容纳了所有其它情况
};
```

`match` 表达式依次检查的返回值是通过下面这些“类似于函数表达式”的右侧类型决定的。它要求每个表达式拥有相同的类型。

3.1.5 算术表达式

算术表达式是包含加法运算符 `+`、乘法运算符 `*`、减法运算符 `-` 或除法运算符 `/` 的运算符表达式。它拥有下面的预设[4]：

- 加法运算符 `+` 一般要求满足交换律和结合律，且不会抛出异常。
- 乘法运算符 `*` 一般要求满足交换律和结合律，且不会抛出异常。
- 减法运算符 `-` 一般要求不会抛出异常。
- 除法运算符 `/` 没有任何要求。

内置支持算术运算符的类型包括整数和实数，还有 `Prelude` 中定义的一系列算术类型。

```
const main = funct () -> do {
    auto x = 42 / 24;        // Int 之间的是整数除法，因此结果是 1
    auto y = 1.0 * 2.0;      // 结果是 2.0
};
```

[3]这和 Haskell 的设定不同，需要注意。

[4]所谓预设就是语言和标准库保证满足，且建议程序员也遵守的规则。不过，即使没有遵守也不会造成编译错误，但可能会造成运行时功能偏离预期。这是面向契约编程的特点。

3.1.6 位运算表达式

位运算表达式是包含按位与运算符 `bitand`、按位或运算符 `bitor`、左移运算符 `shl`、右移运算符 `shr` 或按位异或运算符 `xor` 的运算符表达式，它们没有任何预设。

```
const main = funct () -> do {
    print 1 bitand 0;       // 输出 0
    print 1 xor 0;          // 输出 1
};
```

3.1.7 比较表达式

比较表达式是包含小于运算符 `<`、小于等于运算符 `<=`、大于运算符 `>`、大于等于运算符 `>=`、等于运算符 `==`、不等于运算符 `<>` 或三路比较运算符 `<=>` 的运算符表达式。它们拥有下面的预设：

- 除了三路比较运算符 `<=>` 外所有运算符都返回 `Prelude.Bool` 类型。
- 三路比较运算符 `<=>` 应该返回一个可以和 `0` 比较的对象，且其等于零当且仅当 `==` 返回真、大于零当且仅当 `>` 返回真、小于零当且仅当 `<` 返回真[5]。
- 小于运算符 `<`、大于运算符 `>` 一般要求满足反对称性，且不会抛出异常。
- 小于等于运算符 `<=` 返回值为真当且仅当 `<` 或 `==` 返回真。大于等于运算符同理。
- 不等于运算符 `<>` 返回值为真当且仅当 `==` 返回假。

```
const main = funct () -> do {
    print 1 < 2;            // 输出 True
    print 1 <=> 2;          // 输出 -1
};
```

3.1.8 逻辑表达式与条件表达式

逻辑表达式是包含与运算符 `and` 或或运算符 `or` 的表达式。它们拥有下面的预设：

- `and` 运算符的两个参数和返回类型都应该能隐式转换为 `Prelude.Bool` 类型。
- 两个运算符都应该拥有**短路**性质。对 `and` 来说，如果第一个参数可以转换为 `False`，则表达式直接得到一个结果；对 `or` 来说，如果第一个参数不是 `undefined`，则表达式直接得到一个结果。短路发生时第二个参数不会被求值。

```
const main = funct () -> do {
    auto danger = funct () -> do { print undefined; return False; };
    print False and danger();       // 输出 False，因为 danger() 并没有被求值
    print True or danger();         // 输出 True，因为 danger() 并没有被求值
};
```

特别地，如果一个类型上没有定义转换到 `Prelude.Bool` 的函数，则会默认判断它是否为 `undefined`，且仅在此时返回一个 `False`，否则返回 `True`。

```
const main = funct () -> do {
    print "Hello," or " world!";    // 输出 Hello,
    print undefined and 42;         // 运行期错误，对 undefined 求值
};
```

[5] 显然，AutoScript 在三路比较运算符上相当程度地借鉴了 C++ 的设计。只不过 AutoScript 并没有强有序、弱有序和部分有序的分类。

这个性质加上逻辑表达式短路的特性，可以用来实现条件表达式。

语法 3.4 (条件表达式 [cond-expr])

```
[expr-cond] and [expr-1] or [expr-2]
```

♣

上面的表达式中，首先会对 [bool-expr] and [expr-1] 进行求值，若 [bool-expr] 为真，则会返回 [expr-1] 的值，否则返回 [expr-2] 的值。这个特性的一个致命缺陷在于当 [expr-1] 本身可以显式转换为 False 时，条件表达式依然会返回 [expr-2]。在内置类型中，这样的表达式只有 False 和 undefined [6]，因此这并不是一个显著的问题。

```
const main = funct () -> do {
    scan x : Int;
    print x < 10 and "Small"    // 若 x 小于 10 则输出 Small
       or x > 20 and "Big"      // 若 x 大于 20 则输出 Big
       or "Good";               // 其余情形输出 Good
};
```

条件表达式中 and 右侧的类型应该完全一致，否则编译器就无法成功推导表达式的类型。但我们可以使用显式类型转换。

```
const main = funct () -> do {
    print (x < 10 and "Good"
        or x > 20 and undefined
        or 42) as Bool;         // 声明为 Bool 类型的表达式
};
```

3.1.9 类型转换表达式

类型转换表达式是包含类型转换运算符 as 或 of 的表达式。前者的行为总是固定的（即在元组类型和相同内存布局的其它复合类型之间的转换），但后者允许通过属性 @ctor 来覆写其行为，我们会在结构章节中详细说明。

```
const Point = struct (x : Real, y : Real);

const main = funct () -> do {
    auto p1 = (1.0, 0.0) as Point;
    auto p2 = Point of (1.0, 0.0);
};
```

3.1.10 其它运算符表达式

AutoScript 提供了 in 运算符来判断某个元素在一个范围中是否出现。

```
const main = funct () -> do {
    auto arr = (1, 2, 3);
    print 1 in arr;         // 输出 True
    print 0 in arr;         // 输出 False
};
```

[6] 如果你有 C++ 等语言的背景，这里要特别注意：AutoScript 中的 Bool 不是内置类型，0 并不能被强制类型转换为 False。

在 Prelude 中定义了字符串的拼接运算符 ++。

```
const main = funct () -> do {
    auto str = "abc";
    print str ++ "def";      // 输出 abcdef
};
```

下一节中我们将介绍如何自定义运算符。

3.1.11 运算符

运算符的优先级有高低之分，这是为了照顾一些管用的表达式形式。比如 1 + 2 * 3 中为了迎合数学的习惯，会优先执行 2 * 3 得到 6，随后执行 1 + 6。此外，对于同样优先级的运算符，它们不通过圆括号分组时，会按照特定的顺序运算（从左到右或从右到左），这被称为*左结合*和*右结合*。下面用一张表展示所有 AutoScript 中运算符的优先级和结合性。

运算符	优先级	结合性
,	0	左
->	1	右
<-	2	右
逻辑运算符	3	左
比较运算符（<=> 以外）	4	左
<=>	5	左
位运算符	6	左
+、-	7	左
*、/	8	左
^	9	左
as、of、hasbase、hasclass、in	10	左
调用运算符	11	右

表 3.1: 运算符的优先级和结合性

AutoScript 支持自定义运算符，我们可以通过 @operator 属性来给出这个运算符的优先级和结合性。默认的情况下是最高优先级和左结合性。

```
@operator(9, left)
const <+> = funct (x : Int) -> funct (y : Int) -> x^2 + y^2;  // 这个运算符和 + 一样的性质

const main = funct () -> do {
    print 1 + 2 <+> 3;       // 输出 18
};
```

虽然运算符直接支持中缀形式，但我们总是可以将其当作普通函数使用。不过由于 AutoScript 对符号组成的名字特殊对待（默认将其视作中缀形式的运算符），因此为了让它能够作为前缀使用，需要在外面套上一个括号。类似地，由字母组成的名字如果要以中缀形式使用，需要写成*运算符字面量*的形式，即将其用反引号包围。

```
@operator(9, left)
const add = funct (x : Int) -> funct (y : Int) -> x + y;

const main = funct () -> do {
    print (+)(1)(2);         // 输出 3
    print 1 `add` 2;         // 输出 3
```

```
};
```

3.1.12 自定义字面量

AutoScript 中可以自定义字面量，其本质是一个函数调用。字面量函数要求函数接收一个数字或字符串（内置的类型），并在声明处使用 `@literal` 属性。这个属性修饰的函数能且仅能接受一个参数。

```
@literal
const dozen = funct (n : Int) -> 12 * n;

const main = funct () -> do {
    print 2dozen;           // 输出 24
};
```

当然，字面量函数可以当作函数使用：

```
@literal
const k = funct (n : Int) -> 1000 * n;

const main = funct () -> do {
    print k(1);             // 输出 1000
    print 1k;               // 输出 1000
};
```

在使用自定义字面量时，可能会出现隐式类型转换。

```
@literal
const dozen = funct (n : Real) -> 12 * n;

const main = funct () -> do {
    print 1dozen;           // 输出 12.0
};
```

3.2 表达式修饰符

AutoScript 中允许在表达式的最外面添加修饰符，用来表示其求值性质或求值得到的对象性质。下表列出了所有的表达式修饰符：

可以看到，我们可以将所有表达式修饰符分为三个类型：

1. 修饰表达式求值后结果的修饰符，也称为**实体修饰符（Entity Qualifier）**。
2. 提示表达式求值后结果范畴的修饰符，也称为**范畴提示符（Category Indicator）**。
3. 提示表达式求值方式的修饰符，也称为**求值修饰符（Evaluation Qualifier）**

下面让我们逐个介绍这些修饰符的作用。

3.2.1 实体修饰符

实体修饰符包含了 `atom`、`dyn`、`mut` 和 `ref`。其特点在于其修饰表达式后会让表达式求值的类型加上这些修饰符。比如 `mut 42` 的求值类型是 `mut Int`。此外，除了 `atom` 和 `mut` 的相互顺序不影响表达式类型，如 `atom mut 42` 和 `mut atom 42` 没有区别。

关键字	简介	类型
atom	原子修饰符	1
await	异步表达式修饰符	3
class	类族提示符	2
comptime	常量表达式修饰符	3
dyn	动态修饰符	1
fixed	恒值表达式修饰符	3
funct	函数提示符	2
import	导入修饰符	3
lazy	惰性表达式修饰符	3
mut	可变修饰符	1
object	具名元组提示符	2
ref	引用修饰符	1
struct	结构提示符	2

表 3.2: 表达式修饰符

```
const main = funct () -> do {
    auto x = atom dyn mut 42;    // 类型为 atom dyn mut Int
};
```

如果要同时使用 ref 和其它修饰符，则需要注意顺序：ref 本身可以被修饰，因此在其左侧出现的修饰符修饰的是引用类型，右侧出现的则修饰被引用类型。

```
const main = funct () -> do {
    auto x = atom ref dyn ref mut ref 42;    // 类型为 atom ref dyn ref mut ref Int
};
```

dyn 和 ref 的性质类似，我们需要注意其它修饰符和 dyn 的相对顺序。下面是对这些修饰符的展开介绍：

- atom 声明了一个原子类型，对这个类型对象的读写都是原子操作，保证线程安全。
- dyn 声明了一个动态类型，对这个类型上可以进行运行期的动态删改。
- mut 声明了一个可变类型，我们此前已经介绍过了。
- ref 声明了一个引用类型，我们此前已经介绍过了。

我们会在高级特性部分中深入讲解 atom 和 dyn 的用法。

3.2.2 范畴修饰符

类别修饰符包括了 class、funct、object 和 struct。它们分别对应了 AutoScript 中的类族、函数、对象和结构，常用于声明变量时的初始化表达式。它们存在的意义是为了让编译器能提前确定表达式的含义，从而简化编译过程。这其中 class 和 object 关键字我们还没有遇到过，详细内容请见后文。

这里值得一提的是，范畴修饰符可以在合适的时机被省略。以函数为例，如果我们提前声明了函数的类型，那么就可以在定义时省略 funct 关键字。以完整的变量定义为例：

```
const f : Int -> Int = x -> x;        // 这里不仅省略了 funct 关键字，还省略了参数外侧的括号
```

结构声明中，struct 也可以在一定条件下被省略，我们会在结构的章节中介绍。

3.2.3 await 表达式

AutoScript 中用 await 表达式来并发执行某个表达式。

```
import Time;

const sleep = funct (sec : Int) -> do {
    print "Ready to sleep for $(sec) seconds";
    await suspend_for(Time.from_second(sec));
    print "Wait done";
};

const main = funct () -> do {
    sleep(1);
};
```

await 表达式的机制比较复杂，因此这里不详细介绍了。我们会在并发的章节中详细介绍这个机制。

3.2.4 常量表达式

AutoScript 中存在常量存储期来表示编译期的常数。除此之外，我们也可以用 comptime 关键字来引入一个常量表达式。这会促使编译器在编译期对 [expr] 进行检查并求值。

```
const main = funct () -> do {
    auto ans = comptime do {
        auto x = 42;
        auto y = 24;
        if (x > y) {
            return "abc";
        }
        return "def";
    };
};
```

因此，我们可以轻易地定义编译期函数，即强制在编译期进行所有函数调用的函数。

```
const const_add = funct (x : Int, y : Int) -> comptime (x + y);

const const_sum = funct (x : Int, y : Int) -> comptime do {
    auto res = const_add(x, y);
    print res;
};
```

值得说明的是，comptime 并不要求其后表达式中出现的函数被声明为 comptime，但如果在编译时发现无法把值在编译期算出，会报编译错误。

```
const const_error = funct () -> comptime do {
    scan x : Int;       // 编译错误，scan 语句无法做到编译期
    print x;
};
```

此外，comptime 让编译器强制将某个表达式进行编译期求值，但编译器有权力将其它表达式也放在编译期求值。只要这是可能的。

```
const f = funct () -> 42;        // 并没有标记为 comptime

const main = funct () -> do {
    auto x = comptime f();       // 没问题，因为 f() 可以在编译期求值
};
```

这是因为“能在编译期求值”、“强制在编译期求值”和“确实在编译器求值”是三个不同的事件（后两者有较强的相关性）[7]。一般来讲我们不需要使用 comptime 关键字；后面介绍模版元编程的章节中我们会看到它的用处。

3.2.5 fixed 表达式

在 monad 表达式中，所有变量定义、赋值运算和返回表达式都会被包装为指定的单子类型；如果想要保持某个表达式的类型不被修改，可以使用 fixed 关键字。

```
const main = funct () -> monad Option {
    auto x = 42;
    return fixed x;      // 返回 42 而不是 Option[Int] 类型
};
```

详见单子章节。

3.2.6 惰性表达式

惰性表达式是通过 lazy 关键字构建的，它会推迟某个表达式的求值。

```
const main = funct () -> do {
    auto x = lazy 42;        // 不对 x 进行求值
    auto y = lazy x + 20;    // 不对 y 进行求值
    print y;                 // 对 x 和 y 都进行求值，输出 62
};
```

由于 lazy 并没有传染性，在任何常规的求值环境中都会迫使其中的惰性表达式被求值，因此只有反复使用 lazy 关键字才能让一个惰性表达式持续延后求值。

注意，lazy 并不是类型的一部分（因为实际上它没有影响对象的性质）。

3.2.7 导入修饰符

导入修饰符是和模块系统相关的关键字。不过它的一些用法不限于狭义的模块。我们可以在 import 后的表达式中使用不在当前名字空间中定义的标识符。

```
// main.asc
const main = funct () -> do {
    auto p : import Utility.Pair = (1, 0);  // 导入了 Utility 模块中的 Pair 类型
};
```

[7] 对 C++ 熟悉的读者可以将“能在编译器求值”对应 constexpr 函数，“强制在编译期求值”是 constexpr 变量或 consteval 函数，“确实在编译器求值”则是所有常量表达式，比如枚举量、非类型模版参数、constexpr 变量等。

3.3 表达式异常

本节虽然不打算详细介绍异常，但有必要提及表达式的异常类型。一个表达式除了拥有类型以外，还有一个可选的异常类型。异常可以通过 throw 语句产生。

```
const main = funct () -> do {
    auto x = do {
        scan b : Bool;
        if (b) {
            throw 42;
        }
    };              // x 的表达式类型为 Void，但异常类型是 Int，因为它可能抛出 Int 对象
    auto y = do {
        return 42;
    };              // y 的表达式类型为 Int，而异常类型是 ()。这也被称为无异常表达式
};
```

throw 语句抛出的异常会通过一层层作用域传递，除非在某个作用域中被捕获。AutoScript 中表达式的异常类型可以通过 exceptof 函数获得。

```
const main = funct () -> do {
    auto x = lazy do { throw 42 };          // 惰性求值，否则会立刻抛出异常
    auto y = funct () -> do { throw 42; };
    print exceptof(x);                      // 输出 Int
    print exceptof(y());                    // 输出 Int
};
```

除了 throw 语句可以产生异常外，任何 monad 表达式都拥有非空的异常类型。

```
const main = funct () -> do {
    auto x = monad Option { return 42; };
    auto y = monad Expected[Or = String] { return 42; };
    print exceptof(x);                      // 输出 Null
    print exceptof(y);                      // 输出 String
};
```

我们将在单子章节中详细介绍两者的联系之处。

第四章 语句

内容提要

- 简单语句
- 语句块
- break 语句
- continue 语句
- 声明语句
- 模式声明
- else 子句
- then 子句
- return 语句
- throw 语句
- assert 语句
- assume 语句
- try 语句

4.1 语句

语句（Statement）代表了 AutoScript 中的一个单独操作[1]。语句之间大多通过；进行分隔，少部分（以块结构结尾的）语句不需要加分号。

4.1.1 简单语句

简单语句顾名思义是结构非常简单的语句，它包括两种情况。

语法 4.1 (简单语句)

1. ;
2. [expr];

可以看到，简单语句包含了空语句和求值语句，前者不做任何事，后者会对表达式进行求值（除非它是一个 lazy 表达式，此时也不会做任何事）。

4.1.2 块语句和跳转语句

块语句（Block Statement），也被称为**语句块（Block）**是由一对大括号组成的结构，其中包含了零到多个语句。语句块定义了一个作用域。回忆第二章中我们曾经介绍过，不同作用域中允许声明相同名字的变量。我们可以用 @name 属性为作用域命名，它会出现在语句块中变量的名字中。

```
const main = func () -> do @name(body) {
    @name(inner) {
        auto x = 42;
        print nameof(x);        // 输出 ::main::body::inner::x
    }
};
```

块语句中的最后一个语句允许不添加分号；这和表达式中在最后可以添加逗号同理[2]。

[1]注意区分单独操作和原子操作：单独操作是指在语义上是一个操作，而原子操作是指其可以被转译为一个单独的汇编指令。AutoScript 中只有 atom 修饰的类型，其读取操作是原子操作。

[2]即，分隔符可以或可以不出现在最后一个元素之后。不过，表达式中最后一个逗号习惯上时常被省略，一系列语句中最后一个分号习惯上时常被加上（C++ 等语言中不允许省略这个分号）。

```
const main = funct () -> do { return 42 };  // 这里没有在 return 语句中使用分号。
```

我们可以用 break 语句来跳出某个语句块。

语法 4.2 (break 语句 [break-stmt])

1. break [scope-id] ;
2. break;

第一种情况跳出给定的作用域，而第二种默认跳出当前作用域[3]。

```
const main = func () -> do {
    @name(outer) {
        scan x : Int;
        if (x > 0) {
            break outer;
        }
    }
};
```

只有局部作用域中允许使用 break 语句。要注意，do 和 monad 表达式后的语句块不能通过 break 语句跳出。

此外，可以使用 continue 语句跳回作用域的开始。

语法 4.3 (continue 语句 [continue-stmt])

1. continue [scope-id] ;
2. continue;

第一种情况跳到给定的作用域开始，而第二种默认跳到最内层的作用域开始。

```
const main = funct () -> do @name(body) {
    scan x : Int;
    @name(inner) {
        scan y : Int;
        if (x * y > 0) {
            continue inner;
        }
        if (x * y < 0) {
            continue body;
        }
    }
};
```

可以看到，continue 语句允许用在 do 表达式的语句块上。

除了使用主动设置的作用域名称外，我们也可以使用下面的上下文关键字来指定跳转的作用域类型：

[3]注意这和 C++ 等语言的设计是不同的：在这些语言中，break 会默认跳出最内侧的*循环语句*，包括 while、for 语句等。AutoScript 设计之初几经考虑，还是选择了默认跳出当前最内侧语句块的设计（参见附录中的讲解）。如果想要做到 C++ 等语言中的效果，可以看马上介绍的几个上下文关键字。

- branch：目标为 if 语句。
- iteration：目标为 for 语句。
- loop：目标为 while 语句。

```
const main = funct () -> do {
    scan x : mut Int;
    while (x > 0) {
        if (x == 42) {
            break loop;     // 跳出 while 语句
        }
        scan x;
    }
};
```

4.1.3 声明语句

本节让我们给出声明语句更完整的语法[4]。

语法 4.4 (声明语句)

1. [storage-spec] [id] : [type-spec] ;
2. [pattern] = [expr] ;

♣

可以将上面的语法和语法 2.1 比较。第一种情况是一个*提前声明*，提示编译器某个名字拥有的类型。第二种情况下，变量声明包含一个**模式 (Pattern)** 和初始化表达式。这里，[pattern] 是一个**模式声明 (Pattern Declaration)**，其中包含了一个特定结构，且可能会引入一些名字（在声明语句中，几乎总会引入一个名字）。[expr] 是一个表达式，它的值这里被称为待匹配实体。有关模式声明，请参考下面的一些例子：

```
const main = funct () -> do {
    auto tup = (1, 2);
    auto (m, n) = tup;      // 这里使用元组模式声明 (m, n)
    print m;                // 输出 1
    print n;                // 输出 2
};
```

我们可以在模式声明中再次使用模式声明，这也就是模式的嵌套：

```
const main = funct () -> do {
    auto tup = (1, 2);
    auto (m : mut, n) = tup;// 这里的 m : mut 是一个原子模式，它嵌在元组模式 (m : mut, n) 中
    print typeof(m);        // 输出 mut Int
    print typeof(n);        // 输出 Int
};
```

注意到 m : mut 中并没有将 m 的类型完全写出来，这是因为我们可以通过 tup 来推断它的类型（并在最外层加上 mut 修饰）。如果要得到某个实体的引用，则可以通过下面的方式：

[4]这里的语法已经包含了 AutoScript 最重要的知识点之一，模式匹配了。后面的章节中我们还会扩充变量声明的语法，毕竟这是 AutoScript 最复杂的概念之一。

```
const main = funct () -> do {
    auto tup = mut (1, 2);
    auto (ref m, n) = ref tup;  // 这里的 ref m 是一个引用模式声明，此处拿到的 m 是 tup[0] 的
    别名
    print typeof(m);            // 输出 Int
    print typeof(n);            // 输出 ref Int
};
```

这里我们看到，元组的引用类型可以被匹配为元组模式，其中的每个元素类型都是原来元组中元素的引用类型。

这里有必要说明出现在模式声明中的 `ref` 和作为类型限定符的 `mut` 的区别。模式声明会匹配对象进行强匹配：被匹配的值需要完全满足模式声明中的结构，但类型限定符允许初始化值被隐式类型转换为声明的类型。

```
const main = funct () -> do {
    auto x = 42;
    auto y = mut 42;
    auto mut z = x;          // 编译错误，mut y 无法匹配 x，后者不是可变类型
    auto mut z = y;          // 没问题
    auto w : mut = x;        // 没问题，模式匹配本身并不要求 mut 模式，而只是将 w 声明为 mut
    print typeof(z);         // 输出 Int，因为 mut y 作为模式声明整体才是 mut Int
    print typeof(w);         // 输出 mut Int
};
```

注意到 `mut` 模式可以将初始化表达式中的 `mut` 去掉，因此这是一种规避声明可变类型的方式：

```
const main = funct () -> do {
    auto x = 42;
    auto y = mut 42;
    auto mut z = mut x;      // 类型是 Int
    auto mut w = mut y;      // 类型是 Int
};
```

若非必要，并不推荐将 `ref` 以外的修饰符用于模式声明中，因为它会造成迷惑[5]。此后我们会在模式匹配章节中更深入地介绍这个系统。

4.1.4 条件语句

我们在第一章中已经介绍过 `if` 语句的用法。不过，光有 `if` 语句本身似乎表达力不够充足：我们已经判断过的表达式，此后应该不需要再反过来判断一遍才对。此时可以使用 `else 子句`。

语法 4.5 (else 子句)

```
[stmt] else [stmt]
```

♣

`else` 子句和 `if` 语句一起使用时，当且仅当 `if` 语句的表达式判断为假时会执行 `else` 后的语句块。

[5]确实，明明声明为 `mut x`，但 `x` 本身一定不是可变类型。初见时可能非常反直觉，但对 AutoScript 中的模式熟悉之后，这会变得自然。至于规避声明可变类型的方式，此后我们会介绍一个更加简洁的方法。

```
const main = funct () -> do {
    scan x : Int;
    if (x < 0) {
        print "Negative";
    }
    else {                          // 以 x < 0 为假为前提
        print "Non negative";
    }
};
```

注意到 else 子句中不仅可以放一个语句块，也可以放一个 if 语句，因此我们可以连续写多个互斥的 if 语句来筛选情况。

```
const main = funct () -> do {
    scan x : Int;
    if (x < 0) {
        print "Too small";
    }
    else if (x < 20) {
        print "A bit small";
    }
    else if (x < 80) {
        print "Good";
        return;
    }
    else if (x < 100) {
        print "A bit large";
    }
    else {
        print "Too large";
    }
    continue;
};
```

4.1.5 循环语句

从本章新介绍的 continue 出发，我们可以实现一个最原始的循环语句：

```
const main = funct () -> do {
    // 做一些事情
    continue;
};
```

实际上，while 语句可以看作是带有 continue 语句的语句块的语法糖：

```
const main = funct () -> do {
    scan b : Bool;
```

```
    while (b) {
        // 做一些事情
    }

    // 上面的代码和下面的等价
    {
        // 做一些事情
        continue if (b);
    }
};
```

显然 while 语句有更好的可读性。以上面的等价代码为启示，我们也可以猜到 break 和 continue 语句在 while 代码块中的作用。值得特别一提的是 else 子句在 while 语句中的作用。如果 while 语句由于没有满足表达式条件退出，则会进入后面的 else 子句。

```
const main = funct () -> do {
    scan x : Int;
    while (x > 0) {
        print "Good";
        if (x == 42) {
            print "Pretty good";
            break;
        }
        print "Try again";
        scan x;
    }
    else {
        print "Bad";
    }
};
```

上面的程序中，如果某次输入了负数，则会输出 Bad 然后程序结束。否则则会反复要求输入整数，直到输入 42 后通过 break 语句退出循环。注意通过 break 语句退出时不会进入 else 子句。

如果想要让 break 出来的控制流进行额外操作，可以使用 then 子句。

语法 4.6 (then 子句)

```
[stmt] then [block]
```

♣

then 子句和 else 子句可以一起使用。在同一个语句中，它们的顺序并不重要，但分别只能出现一次。即使可以在这两个子句中再次使用 break 语句，但不能通过再添加一个 then 语句来接管控制流。

```
const main = funct () -> do {
    scan x : Int;
    while (x >= 0) {
        if (x == 42) {
            print "Secret";
            break;
```

```
        }
        scan x;
    }
    then {
        print "You've passed!";
    }
    else {
        print "You've failed";
    }
};
```

4.1.6 遍历语句

语法 4.7 (for 语句)

```
for ([decl]) [block]
```

♣

AutoScript 中的遍历语句 `for` 和其它语言（如 C++）的含义有一定差别。它的本质和循环语句不同，而是一个临时的绑定操作，通常绑定的是一个单子。`for (x from range) {...}` 基本上相当于 `range.bind(func () -> {...})` [6]。

```
const main = funct () -> do {
    for (i from range(0, 10)) {          // 这里 i 的类型是 Int
        print i;                         // 从 0 输出到 9
    }
    auto maybe_int = Some 42;            // maybe_int 的类型是 Option[Int]
    for (i from maybe_int) {
        print i;                         // 输出 42
    }
};
```

上面用到了 `Prelude` 中定义的 `range` 函数，其返回一个左闭右开的区间。`Some 42` 构建了一个 `Option` 类型。可以看到，`for` 语句可以将一些单子类型中的元素取出，这些类型并不是一个范围。

`for` 语句中也可以使用 `else` 和 `then` 子句。当 `for` 遍历的对象为空时（或遍历到最后），会进入 `else` 子句。否则（通过 `break` 语句跳出时）进入 `then` 子句。这个逻辑和此前的 `if`、`while` 语句是一致的。

```
const main = funct () -> do {
    scan arr : Array(Int, 3);
    print "( ";
    for (e from arr) {
        print e;
    }
    then {
        print ")";
    }
```

[6] 熟悉 Haskell 等语言的读者应该知道，列表本身也是一个单子，其绑定操作相当于遍历列表逐个调用函数。

```
    else {
        print "The array is empty";
    }
};
```

4.1.7 返回语句

AutoScript 中，返回语句用来跳出 do 或 monad 表达式（相当于 break 语句之于普通的语句块，不过返回语句需要返回一个值）。这个过程可以通过常规的值返回（即我们熟悉的函数返回值）或异常抛出完成。

语法 4.8 (return 语句)

(1) `return [expr];`

(2) `return;`

return 语句将一个表达式的值作为返回值。如果没有给出这个表达式，则返回空对象 ()。我们已经使用它很多次了。在 do 表达式的末尾会默认插入一个 return 语句。

```
const foo = funct () -> do {
    scan x : Int;
    if (x == 42) {
        return 0;
    }
    // 这里会默认插入 `return;`，但这和此前推导的 do 表达式类型不同，因此会产生编译错误。
};
```

语法 4.9 (throw 语句)

(1) `throw [expr];`

(2) `throw;`

throw 语句将构成返回表达式的异常类型。也可以省略表达式，用来返回一个空对象，这点和 return 语句是一致的 [7]。

语法 4.10 (yield 语句)

(1) `yield [expr];`

(2) `yield;`

yield 语句用在协程中，其暂停当前的协程并返回一个值。如果不给定表达式，则会默认返回空对象。

```
const ints = funct () -> do {
    auto i = mut 0;
    yield i;
    i <- i + 1;
    continue;
};

```

[7] 这里的语法和 C++ 中不带表达式的 throw 语句含义（即抛出未处理的异常）不同，这是为了保持相似功能语句在语义上的一致性（return、throw，包括马上要介绍的 yield 语句）。如果想要抛出函数中待处理的异常，可以在捕获异常（通过 catch 子句，我们很快就会介绍）后将其显式抛出。

```
const main = funct () -> do {
    for (i from ints()) {
        print i;                    // 无限地打印从自然数（0, 1, 2, ...）。
    }
};
```

我们会在并发章节中详细介绍 yield 语句。

4.1.8 输入输出语句

AutoScript 利用 print 语句进行输出，我们已经多次使用了。

语法 4.11 (print 语句)

```
print [expr];
```

♣

默认情况下，print 会打印在命令行上。如果想要手动选择输出的目标，可以使用属性 @stream 属性，其中可以放入一个已经打开的文件流。对于 print 语句，会让其内容在给定的文件流中输出。

```
import System;

const main = funct () -> do {
    auto file = System.open("output.txt", "w");
    @stream(file)
    print "Hello, world!";
};
```

此外，也可以用 @endwith 来指定 print 结尾的字符（默认为 \n）。

打印字符串时的格式化操作并不是 print 的任务，我们需要利用字符串插值等方式调整打印结果。

```
const main = funct () -> do {
    auto x = 42.0;
    print "$(x;f.3)";       // 小数点后打印 3 位，输出 42.000
};
```

scan 语句则用来输入。

语法 4.12 (scan 语句)

(1) scan [expr];

(2) scan [forward-decl]

♣

和 print 语句类似地，我们可以通过 @stream 属性来指定一个输入流。

```
import System;

const main = funct () -> do {
    auto file = System.open("input.txt", "r");
    @stream(file)
    scan text : String;
};
```

4.1.9 优化提示语句

AutoScript 提供了一系列语句来为编译器进行优化提示。其中最常见的是用于断言的 assert 语句。

语法 4.13 (assert 语句)

```
assert [expr];
```

♣

若 assert 后的表达式在编译期判断为假，则编译器可以中止编译并输出断言错误。如果这个表达式无法编译期求值，则会生成一段代码，在运行期判断为假时抛出断言异常。

```
const main = funct () -> do {
    assert 1 == 0;          // 编译错误，断言失败
    scan x : Int;
    assert x == 42;         // 编译期无法判断

    // 编译器会生成下面的代码
    if (not (x == 42)) {
        throw AssertionError("Expected x == 42");
    }
};
```

可以使用 @alert 属性让 assert 失败时调用自定义的函数来输出错误信息。这常用于编译器的 Debug 提示。

```
const message = funct () -> print "Use common sense!";

const main = funct () -> do {
    @alert(message)
    assert 1 == 0;
};
```

注意 @alert 中传入的函数必须不接入任何参数。

比 assert 语句“温和”一些的是 assume 语句。

语法 4.14 (assume 语句)

```
assume [expr];
```

♣

编译期会将表达式 [expr] 当作优化提示。如果实际情况并非如此可能会出现未定义行为。

```
const main = funct () -> do {
    scan x : Int;
    assume x > 0;
    if (x == 0) {
        print "Impossible...?";
    }
};
```

上面的例子中，编译期通过 assume 的表达式知晓了 x 理应大于零，因此在条件语句中会跳过判断。如果在运行期我们输入了一个非正数，那么就是一个未定义行为。assume 语句可以说是面向契约编程的核心功能之一。

一些编译期可能无法通过的语句可以用预演语句 try 来尝试。

语法 4.15 (try 语句)

```
try [stmt]
```

如果编译期就发现这个语句不可能执行，则编译器会将其消除。如果无法编译期判断（比如使用了动态类型），则会在运行期检查并选择性跳过。

```
const main = funct () -> do {
    auto x = 42;
    try x <- 20;            // 预演失败，因为 Int 不是可变类型
    auto y = dyn 42;
    try y.val <- 24;        // 运行时预演失败，因为 y 没有 data 这个动态成员
};
```

可以使用 @alert 属性让 try 语句预演失败时调用一个函数。

```
const message = funct () -> print "Check mutability!";

const main = funct () -> do {
    auto x = 42;
    @alert(message)
    try x <- 20;
};
```

最后还有用于类型约束缓存的 implement 语句，让我们在模版的章节中再详细介绍。

4.2 子句

此前我们已经介绍过 if、else、then 等子句了。本节中让我们介绍一些其它常用的子句。

4.2.1 where 子句

在任何语句中都可以使用 where 子句。

语法 4.16 (where 子句)

```
where [block]
```

在 where 子句中的语句块中定义的任何变量都会引入到当前语句中。

```
const main = funct () -> do {
    auto fib_10 = fib(10) where {
        auto fib : Int -> Int;
        auto fib = funct (x : Int) -> do {
            if (x <= 1) {
                return x;
            }
            return fib(x-1) + fib(x-2);
        };
```

```
    };
};
```

同一个语句中只允许出现一个 where 子句。

4.2.2 with 子句

可以通过 with 子句将一些自动存储期的变量引入到内部作用域，从而提前释放这个变量。

语法 4.17 (with 子句)

```
with [id-tuple]
```

♣

```
const main = funct () -> do {
    auto x = 42;
    auto str = "abc";
    {
        auto x = 24;    // 错误，因为 x 被引入了当前作用域
    } with (x, str);
};
```

4.2.3 requires 子句

在声明语句中都可以使用 requires 子句，它要求当前声明的对象拥有特定的性质，否则产生编译错误。

语法 4.18 (requires 子句)

```
requires [expr]
```

♣

```
const main = funct () -> do {
    auto x = 42 requires True;              // 没问题
    auto y = 24 requires sizeof(y) == 0;    // 编译错误
};
```

其编译期检查的特征也就要求表达式必须可以在编译期求值。

4.2.4 catch 子句

catch 子句用于捕获一个语句的异常，通常这个异常来源于一个表达式。

```
const main = funct () -> do {
    scan x : Int catch e -> print e;        // 输入 abc，输出一个 InvalidInputException
    auto f = funct () -> {
        scan b : Bool;
        if (b) {
            throw 42;
        }
        else {
            throw "abc";
        }
```

```
11      };
12      print f() catch e -> e;                  // 输出 42 或 abc
13  };
```

可以看到，catch 子句会捕获一个异常对象后返回一个对象，并作为所在表达式的值。如果语句并不是表达式语句，则返回的值会被忽略。

有关 catch 子句的内容，我们会在单子和异常处理的章节中详细介绍。

第五章 函数

内容提要

- 函数表达式
- 函数类型推断
- 函数重载
- 函数重复定义
- guard 表达式
- match 表达式
- 模式作用域
- this 参数
- 修饰名字
- 函数运算符
- 默认参数
- 查询结构
- 缺省提示符
- 缺省调用
- 函数中的变量存储期
- 参数的传递方式
- 捕获变量

5.1 函数表达式

5.1.1 函数表达式的声明

函数是一段代码的模版。通过函数调用，用户可以将参数交给函数并执行这段代码。在 AutoScript 中，函数并没有被特别对待，我们可以通过函数表达式来创建一个函数实体。

语法 5.1 (函数表达式 [funct-expr])

```
[pattern] -> [expr]
```

♣

这里的 [pattern] 就是我们在上一章中介绍过的模式，它用来对函数调用进行模式匹配。我们此前提到过，在函数表达式中并不要求写出 funct；但在实际使用中，如果变量尚未提前声明，需要使用关键字 funct 来提示编译器这是一个函数。

```
const main = funct () -> do {
    auto add = funct (x : Int, y : Int) -> x + y;    // (x : Int, y : Int) 是一个元组模式
    print add(1, 0);                                 // 输出 1
};
```

函数表达式的类型是由其参数类型和函数体类型决定的。本节中让我们暂时假设所有函数参数类型都是确定的（即在模式中显式给出变量类型），此时函数体的类型可以通过 do 表达式的类型推断来得到。不过在一些情况下，这个推断会遇到问题：

```
const fib = funct (n : Int) -> do {
    if (n > 1) {
        return fib(n-1) + fib(n-2);      // 编译错误，fib 类型推断出现了循环引用
    }
    return n;
};
```

上面的函数体中，我们推断 fib 的返回类型需要首先得知 fib(n-1) 和 fib(n-2) 的类型[1]，这样就构成了循环

[1]这是因为函数表达式类型的推断依赖于第一个返回表达式的类型；其后的返回表达式除了判断类型外，都会尝试隐式转换为第一次推断出来的类型。

引用。为了解决这个问题，要么将 if 语句的判断条件反过来，要么就首先显式给出 fib 的类型：

```
const fib : Int -> Int = (n : Int) -> do {
    if (n > 1) {
        return fib(n-1) + fib(n-2);
    }
    return n;
};
```

这样的声明方式也能让我们略去 funct 的使用，一举两得。

5.1.2 函数重载

AutoScript 中允许“重复定义”函数，只要它们满足提前声明的类别[2]且（如果显式给出类型限定符或可以通过初始化表达式推导时）和提前声明的类型不冲突，这也被称为**函数重载（Function Overloading）**[3]。在函数调用时，会以声明顺序依次尝试匹配。

```
const main = funct () -> do {
    auto f : (Int, Int) -> String;
    auto f = (x ? 42, y) -> "x is 42";
    auto f = (x, y ? 42) -> "y is 42";
    auto f = (x ? 42, y ? 42) -> "Both are 42";
    auto f = (x, y) -> "Other";

    print f(42, 42);                // 输出 x is 42
};
```

上面示例中用到的符号 ? 称为约束提示符，x ? 42 表示这里匹配一个正好等于 42 的对象[4]。这里，由于 (x ? 42, y) 被首先匹配，因此函数会直接输出 "x is 42"。好在，编译器会对这种情况进行警告：(x ? 42, y ? 42) 是 (x ? 42, y) 的子模式然而却出现在后者的后面，会让这种模式被完全弃用。

不过，AutoScript 并非要求函数重载时的类型完全相同。在模版的章节中我们会介绍*范型函数*，它能接受任何满足特定*约束*的函数重载。这里，让我们首先关注拥有相同解引用类型的函数参数产生的重载。比如 ref Int 和 ref mut Int。此时我们需要使用 mut 修饰符的通用形式 mut?，它可以接受 mut 或非 mut 的参数。

```
const Point = struct (x : Real, y : Real);
const print_point = funct (p : ref mut? Point) -> print "The point is $(p)";
const set_point : (ref mut? Point, x : Real, y : Real) -> ();
const set_point = (p : ref mut Point, x : Real, y : Real) -> do {
    p.x <- x;
    p.y <- y;
};
const set_point = (p : ref mut Point, x, y) -> print "The point is immutable";
```

其它拥有通用形式的修饰符包括 atom、dyn 和 ref。

[2] 回忆*类别*取决于类型中 -> 的个数

[3] 显然这和常见语言，如 C++ 中的重载定义不同。后者的函数重载灵活性更大，基本不对同名函数的参数类型作任何限制，但也带来了复杂的*重载决议*。AutoScript 极力避免在链接过程中额外的困难，因此决定摒弃任意的函数重载。

[4] 约束提示符后出现的对象要求匹配完全相同类型和相同值的对象，即 x ? y = z 要求 typeof(y) == typeof(z) 且 y == z。

5.1.3 guard 表达式和 match 表达式

重复定义的函数显得非常冗余，因为其中的大部分内容是相同的（也就是 `auto f` 的部分），因此更流行的做法是使用 guard 表达式。

语法 5.2 (guard 表达式)

```
guard [funct-expr-list]
```

♣

它的本质是 match 表达式的语法糖：`guard [funct-expr-list]` 会被编译器理解为 `funct (args) -> match (args) [funct-expr-list]`。

```
const main = funct () -> do {
    auto f : (Int, Int) -> String = guard
        (x ? 42, y ? 42) -> "Both are 42",
        (x ? 42, y) -> "x is 42",
        (x, y ? 42) -> "y is 42",
        otherwise -> "Other";
};
```

上面用到的 otherwise 是一个特殊的关键字，它可以匹配任意的实体，因此可以作为默认的匹配情况。这里值得一并介绍 match 表达式的语法：

语法 5.3 (match 表达式)

```
match ([expr]) [funct-expr] , ... , [funct-expr]
```

♣

因此，match 表达式本质是一系列函数的罗列，它们的参数拥有相同的类型，在给出实际的表达式时，编译器会尝试找到一个最好的匹配。

```
const main = funct () -> do {
    scan x : Int;
    print match (x)
        (? 42) -> "Good",
        otherwise -> "Bad";

    // 编译器会生成下面的代码
    auto match__x : Int -> String;
    auto match__x = (? 42) -> "Good";
    auto match__x = (otherwise) -> "Bad";
    print match__x(x);                  // 这一行是 match 表达式
};
```

可以看到在 match 语句中出现的函数表达式，我们省略了参数的名字，其原因我们上一章已经提到了：模式声明中不一定引入变量，因此变量名完全可以省略。

match 表达式要求其中的每个函数都有相同的返回类型（这样才能确保 match 表达式拥有确定的返回类型）。如果想让 match 表达式拥有选择类型，可以在 match ([expr]) 语句中使用 choice 关键字。

```
const main = funct () -> do {
    scan x : Int;
    auto res =  match (x) choice
        (? 42) -> "Good",
        otherwise -> 42;
    print typeof(res);      // 输出 Int | String
};
```

此时，编译器会在 match 表达式周围套上一个 do 表达式。

```
const main = funct () -> do {
    scan x : Int;
    // 编译器会将上个示例转换为下面的代码
    auto res = do choice {
        match (x)
            (? 42) -> return "Good",
            otherwise -> return 42;
    };
};
```

不过很多情况下，match 语句中不同函数表达式返回类型不同是因为特定情况下，我们希望使用跳转语句。此时比起使用 choice 语句使得整个表达式拥有选择类型，不如利用 noreturn 关键字告知编译器这个函数表达式没有返回类型。

```
const main = funct () -> do @name(body) {
    scan x : Int;
    auto x = match (x) choice
        (? 42) -> x,
        otherwise -> do {
            continue body;
        };
   print typeof(x);         // 输出 Int | Void

   scan y : Int;
   auto y = match (y)
       (? 42) -> y,
       otherwise -> do noreturn {
           continue body;
       };
   print typeof(x);         // 输出 Int
};
```

得知函数表达式没有返回类型后，编译器就不会将其作为所在作用域中的类型推断参考，因此上面 match 语句的类型不存在冲突。

有关模式匹配，我们会在 AutoScript 的高级特性中进一步介绍。

5.2 函数上的运算符

5.2.1 点调用符与修饰名字

AutoScript 中有一个特殊的函数调用语法 `a.f(...args)`。通常，这样的函数在声明时都指定了唯一的一个 `this` 参数：

```
const prepend = funct (s : String, this t : String) -> s ++ t;
const main = funct () -> do {
    print "world!".prepend("Hello, ");  // 输出 Hello, world!
};
```

参数中存在 `this` 修饰的函数（称为成员函数）会被编译器进行特殊处理。

```
const prepend = funct (s : String, this t : String) -> s ++ t;

// 编译器会生成下面的代码
const ::String::prepend = funct (t : String) -> funct (s : String) -> s ++ t;

const main = funct () -> do {
    print "world".prepend("Hello, ");

    // 编译器会生成下面的代码
    print ::String::prepend("world")("Hello, ");
};
```

上面例子中出现的 `::` 运算符称为**域连接符 (Scope Connective)**。它本质上是为了区分名字的特殊符号，将其看作名字的一部分即可[5]。实际上，所有的变量真正的名字都是类似的。以 `main` 函数为例，其真正的名字是 `::main`。像这样带有 `::` 的名字被称为**修饰名字（Qualified Name）**。

```
const main = funct () -> do @name(body) {
    auto x = 42;         // 修饰名字是 ::main::body::x
    @name(inner) {
        auto x = 24;     // 修饰名字是 ::main::body::inner::x
        print x;         // 输出 24，因为未修饰名字 x 已经被覆盖为 inner 中的 x 了
        print body::x;   // 输出 42，因为 body::x 指向的是外部作用域中的 x
    }
};
```

可以看到，修饰名字中的每个部分来源于 AutoScript 中的作用域名字[6]。如果我们没有为作用域取名，我们就没法使用修饰名字来访问变量了。如果使用未修饰名字，则需要遵守变量覆盖的规律。

5.2.2 按返回值运算符

AutoScript 相同参数的函数之间可以使用大多数运算符。此时编译器会生成一个按返回值运算的函数。

[5]如果观察 `::` 的用法，就会发现它并不符合 AutoScript 中对运算符的定义：我们可以将其用在表达式的开头。

[6]注意到函数的作用域和其函数体（`do` 表达式）的作用域并非同一个作用域。

```
const inc = funct (x : Int) -> x + 1;
const dec = funct (x : Int) -> x - 1;
const sqrm1 = inc * dec;

// 上面的函数与下面的等价
const sqrm1 = funct (x : Int) -> inc(x) * dec(x);
```

排除在外的有调用运算符，如 `f g` 还是会正常地处理为将 `g` 作为参数传到 `f` 中，以及点调用运算符，如 `f.g` 还是会尝试找到 `g` 的修饰名字，随后将 `f` 作为参数传入。

5.2.3 复合运算符

最后，我们可以用 `<<` 运算符来进行函数复合。如果函数 `f` 的返回类型可以隐式类型转换为 `g` 的参数类型，则可以用 `g << f` 来表示两个函数的复合。

```
const inc = funct (x : Int) -> x + 1;
const dec = funct (x : Int) -> x - 1;
const id = inc << dec;

// 上面的函数与下面的等价
const id = funct (x : Int) -> inc(dec(x));
```

类似地还有另一个方向的复合运算符 `>>`。

```
const id = dec >> inc;
```

5.2.4 管道运算符

除了默认的函数调用 `f x`，我们可以使用管道运算符 `|>` 将参数前置。

```
auto f = funct (x : Int) -> x;
auto g = funct (x : Int) -> x;

const main = funct () -> do {
    print 42 |> f |> g;          // 输出 42
}
```

这也是对调用运算符没有显示形式的一种补偿。

5.2.5 运算符组

AutoScript 中，运算符的部分应用比较常用：

```
const main = funct () -> do {
    auto inc = funct (x : Int) -> x + 1
    auto arr = (1, 2, 3);
    print arr.map(inc);                  // 输出 2, 3 ,4
};
```

不过无论是在行内写出这么长一个函数，还是提前声明都显得有些笨拙。因此，AutoScript 引入了**运算符组（Operator Section）**的概念，我们可以通过语法糖迅速构建一个函数：

语法 5.4 (运算符组)

1. `([operator] [atom-expr])`
2. `([atom-expr] [operator])`

当运算符和原子表达式（变量或字面量）一并使用并加上圆括号时，编译器会将其视作一次部分调用，分别绑定运算符第一个参数和第二个参数。

```
const main = funct () -> do {
    auto arr = (1, 2, 3);
    print arr.map(+ 1);                 // 输出 2, 3, 4

    // 编译器会生成下面的代码：
    print arr.map(funct x -> x + 1);
};
```

这里生成的函数没有给出参数的类型，因此是一个函数模版，可以接纳任何类型的参数。

所有运算符中，函数调用是最为特殊的（它没有对应的符号）；如果想在运算符组中使用它，可以使用管道运算符代替。

```
const main = funct () -> do {
    auto fs = ((+ 1), (* 2), (/ 3));
    print fs.map(3 |>);                 // 输出 4, 6, 1
};
```

运算符组和函数复合运算符结合使用，可以简化很多函数定义：

```
const main = funct () -> do {
    auto arr = (1, 2, 3, 4, 5);
    print arr.filter((/ 2) >> (== 0));  // 输出 2, 4
};
```

5.2.6 调用序列

对于一些使用相同模式的运算符调用链，我们可以利用**调用序列（Call Chain）**来简化形式。调用序列使用 `call` 关键字。

```
const main = funct () -> do {
    print call (+) 1 2 3 4 5;                       // 输出 15
    auto f = call (>>) (+ 1) (* 2) (/ 3) (- 4);
    print f(0);                                     // 输出 -4
};
```

甚至，我们可以用它来改写函数表达式：

```
const main = funct () -> do {
    auto curry_add : Int -> Int -> Int = call (->) x y (x + y);
};
```

从这里也可以印证 -> 本身是一个表达式。

5.3 函数参数

5.3.1 输出参数

可以为参数使用 @out 属性来表示这是一个输出参数。输出参数的类型必须是一个可变引用类型，它可以让调用处原地声明一个变量（类似于 scan 的常见用法）。作为输出参数传入的变量不必是可变类型：可以将整个函数视作这个变量的初始化过程。

```
const bind_value = funct (@out x : ref mut Int, x_val : Int) -> x <- x_val;

const main = funct () -> do {
    print bind_value(x : Int, 42);  // 输出 42
    print x                         // 输出 42
};
```

这样的语法在一些情况下可以简化语句。@out 也可以用来初始化静态存储期的变量（线程存储期同理）：

```
const foo = funct (@out x : ref mut Int) -> do {
    x <- 42;
};

const bar = funct () -> do {
    foo(static x : mut Int);        // 仅在初始化 x 时调用 foo，此后会跳过
};
```

注意到这里静态存储期变量拥有和常规变量声明相同的特征：它的初始化语句只会在第一次遇到时执行。

常量存储期的变量也可以通过这种方式初始化，但其要求函数是可以编译期求值的。

```
const good = funct () -> 42;
const bad = funct () -> do {
    scan x : Int;
    return x;
};

good(const x : Int);
bad(const y : Int);         // 编译错误，bad 无法在编译期求值
```

5.3.2 查询结构

AutoScript 通常不允许函数调用时参数和声明的类型不同，不过我们有特殊的方式来实现默认参数。首先让我们介绍**查询结构（Query Structure）**。AutoScript 中用方括号来创建一个查询结构。

```
const main = funct () -> do {
    auto q0 = [0];      // 一个查询 0 的结构
    auto q1 = [x = 2];  // 一个查询 x = 2 的结构
};
```

查询结构拥有的类型是 Prelude.Query。与它相关的操作都是编译器内置定义的。下面介绍其中的一些：

- 合并操作。可以用 + 运算符来合并两个查询结构，此时不允许两者中的键出现重复。

```
const main = funct () -> do {
    auto q0 = [x = 0];
    auto q1 = [y = 1];
    print q0 + q1;      // 输出 [x = 0, y = 1]
};
```

- 差集操作。可以用 - 运算符来将一个查询结构中，另一个查询结构中出现的键都删除。

```
const main = funct () -> do {
    auto q0 = [x = 0, y = 1];
    auto q1 = [y = 42];
    print q0 - q1;      // 输出 [x = 0]
};
```

- 成员访问操作。可以用 . 运算符来调取查询结构中某个键对应的值。

```
const main = funct () -> do {
    auto q = [x = 0, y = 1];
    print q.x;          // 输出 0
};
```

我们会在后面详细地介绍查询结构的本质。

5.3.3 缺省提示符

我们可以将查询结构传给函数，编译器会在编译期为我们将参数准备好。

```
const add = funct (x : Int, y : Int) -> x + y;

const main = funct () -> do {
    print add [y = 42, x = 24];     // 编译器会将其转变成 add(24, 42)
};
```

结构参数的名字需要完美对应函数参数中的名字才能正确调用[7]，它们的顺序不重要。当我们用查询结构传递参数时，编译器会检查是否所有参数都被初始化，对于未初始化的参数会以 undefined 作为缺省值传入函数。如果想要为参数设定缺省时的默认值此时只要在后面使用缺省提示符 ?? 再加上一个表达式即可。

语法 5.5 (缺省声明 [default-decl])

?? [expr]

[7]这其实意味着函数的参数会被认真看待，这和 C/C++ 中函数参数名称被忽略所不同。

```
const iterate = funct (n ?? 5 : Int) -> do {
    for (i from range[n]) {
        print i;
    }
};
const main = funct () -> do {
    iterate(3);          // 输出 0, 1, 2
    iterate[];           // 输出 0, 1, 2, 3, 4
};
```

表达式中允许出现已经被声明的函数参数。

```
const pair = funct (x : Int, y ?? x : Int) -> (x, y);
```

正如在变量定义语句中函数的参数类型可以通过初始化表达式的类型来推断一样，缺省提示符后的表达式可以用来推断函数参数类型，因此上面的函数或许可以写成：

```
const pair = funct (x : Int, y ?? x) -> (x, y);
```

不过两个实现是不同的：仅当 y 缺省时两者才都是调用了使用两个 Int 类型参数的函数。否则前者只是一个普通函数，而后者是一个函数模版[8]。

默认参数不能是 this，这会导致编译期错误[9]。

函数中参数的名称并不是必要的，当省略参数名称时，我们可以通过*参数占位符*来为其赋值。

```
const secret = funct (: Int) -> 42;
const main = funct () -> do {
    print secret [$0 = 24];     // 输出 42
};
```

这里的 $0 是一个内置的标识符，表示第一个函数参数。依次类推可以将后面的参数记为 $1，$2，...。编译器支持的参数上限是 $255，也即 256 个参数的函数。如果参数个数超过了 256，则可以用等价的 $[k] 来访问第 $k-1$ 个参数。

对于利用重载（或 guard 语句）的函数，查询结构只能应用于它的原型；此时因为函数并不拥有参数名称，因此我们只能利用参数占位符。这并不意味着模式匹配机制和查询结构不兼容[10]，单纯只是*重载*和查询结构不兼容，编译器难以判断函数使用的是哪一个重载。

```
const f = funct (x : Int, ? 42) -> x;
const f = funct (? 42, y : Int) -> y;
const main = funct () -> do {
    f[x = 42, y = 24];      // 编译错误，经重载的函数没有参数名称。
    f[$0 = 42, $1 = 42];
};
```

[8]我们会在后面学到模版的概念，这里可以简单理解为，y 的地方原本可以填入任意类型的参数，而只在缺省时才一定会推断为 Int 类型。

[9]确实，点调用符不能接受 this 是 undefined，编译器将无法确定函数所在的名字空间。

[10]实际上，最常见的参数列表是一个*元组模式声明*，它也用到了模式匹配，我们会在后面的章节中详细介绍。

最后，我们值得说明缺省提示符 ?? 在其它地方的运用：

```
const main = funct () -> do {
    scan x ?? 42 : Int;      // 若输入了一个非法数字，则 x 初始化为 42
    scan y : Int;
    auto z = y ?? 42;        // 若 y 是 undefined，则 z 初始化为 42
    auto w ?? 42 = y;        // 若 y 是 undefined，则 w 初始化为 42
};
```

可以看到，变量声明中，?? 也可以用于声明缺省时的初始化表达式。不仅如此，所有的表达式都可以和 ?? 联动：

```
const main = funct () -> do {
    scan x : Int;
    print x ?? 42;               // 若 x 是 undefined，则输出 42
    print f ?? eat (1, 2, 3);    // 若 f 是 undefined，则不调用（但这里会输出 ()）
};
```

上面的函数 Prelude.eat 是一个接受任意多参数后返回 () 的无操作函数，上面的示例中，如果 f 是 undefined，则会默认调用 eat。

这里涉及到一个有趣的问题，对于 a ?? b，编译器对两个操作数类型的要求是什么呢？通常来说，当然应该要求两者的类型完全相同，或依照其中一个的类型来推断表达式类型。不过这里涉及到了 AutoScript 中*整表达式类型推断*的机制。在 ?? 中，编译器仅要求整个表达式的类型在不同选择下兼容即可，并选择其中的窄类型。

```
const f = funct (x : Int, y : Real ?? 42) -> x; // 类型是 (Int, Real) -> Int
const g = funct (x : Int) -> x;                 // 类型是 Int -> Int

const main = funct () -> do {
    print (f ?? g) 0;            // 没问题，这个表达式的类型一定是 Int
};
```

5.3.4 缺省属性

函数参数上可以用 @default 设置缺省属性，这样在函数传递时，可以通过 default 来构建特定的默认值。

```
const f = funct (@default(42) x : Int) -> x;

const main = func () -> do {
    print f(0);              // 输出 0
    print f(default);        // 输出 42
};
```

实际上，函数参数上 @default 属性的运用可以延伸到普通的变量声明上，只不过其意义并不大。

```
@default(42)
auto x : Int;

auto x = default;            // 初始化为 42
```

需要注意的是，变量的多次声明中 `@default` 只能出现一次，且其中的表达式必须能被立刻求值[11]。

5.4 函数中的变量

5.4.1 函数作用域和静态变量

函数中出现的变量是指被名字空间限定在当前函数的变量，包括参数、`do` 表达式中或 `where` 子句中声明的变量等。如果使用 `auto` 存储限定符（或缺省该限定符，此时会默认为 `auto`），则会在被定义的作用域结束时释放内存。特别地，参数列表中的自动存储变量，其作用域是特别的*函数作用域*，其开始于参数声明，结束于函数表达式结尾[12]。

```
const f = funct (x : Int) -> do {
    auto y = 42;
    print x + y;
} + do {                // 第一个 do 表达式的作用域到此为止
    auto y = 24;
    print x + y;
};                      // x 的作用域（函数作用域）到这里才结束
```

有时，我们需要函数中存储一些和上下文有关的状态，此时静态存储期的变量会有很大帮助：

```
const record = funct (x : Int) -> do {
    static vec : mut Vector[Int] = default;
    vec.push_back(x);
    return ref vec;
};

const main = funct () -> do {
    auto vec : ref mut Vec[Int] = default;
    for (i from range(10)) {
        vec = record(i * i);
    }
    print vec.size;             // 输出 10
};
```

静态存储期的变量只会被初始化一次，随后调用函数时遇到这个定义语句会自动跳过。

5.4.2 参数和返回值的传递方式

我们已经见到过各种类型的参数了，有作为值传递的（如 `Int`），有作为引用传递的（如 `ref String`），还有作为可变引用传递的（如 `ref mut String`），它们的意义我们已经非常了解了：由于函数参数和变量定义是等价的，因此在函数作用域中会按照参数声明构建一个特定类型的对象。因此，当参数通过值传递时，新的变量和调用处的变量不再有任何关系；反之如果用引用传递，新的变量虽然也是调用处变量的拷贝，但它们的值都是同一个对象的引用。

[11] 出现一次是为了防止其依赖于某个不纯洁的表达式导致歧义；同时立刻求值也可以防止生命周期的问题。

[12] 注意到函数作用域并没有用大括号表示，这是一个特例。

```
const f = funct (x : Int) -> do {
    x <- 42;            // 这里的 x 是值类型
};
const g = funct (x : ref mut Int) -> do {
    x <- 42;            // 这里的 x 是引用类型
};
const main = funct () -> do {
    auto x = mut 0;
    f(x);               // x 没有被修改，因为 f 中的 x 和 main 中的 x 仅仅是值相同
    g(ref x);           // x 被修改为 42，因为 g 中的 x 初始化为 x 的引用
};
```

一般来说，引用传递总是更加高效的方式（因为它的内存占用较小且稳定），且我们可以借此通过函数修改调用端的变量。不过，由于在每次解引用时都需要进行一次内存访问，在需要在函数中频繁访问引用的情形下这也会增加访问成本。内存占用不超过 8 的类型通过值传递总是更高效的（因为这正是一个引用的大小）。

类似地，返回值也可以通过值或引用传递。不过除了上一段所论述的，函数返回值会面临对象生命周期的顾虑。

```
const f = funct (x : Int) -> do {
    return ref x;               // 编译错误，返回了自动存储期变量的引用
};

const main = funct () -> do {
    print deref(f());           // （如果代码通过编译）非法的内存访问！f 中的 x 已经被释放了
};
```

因此，编译器不允许返回自动存储期变量的引用，这也是为了程序安全的设计。下一节中我们会见到，AutoScript 也禁止了任何间接返回局部引用的行为。不过，如果函数参数本身的类型就是引用类型，编译器不会阻止其作为返回值返回；这种类型的参数经常会作为多返回值的实现方式。

```
const random_max = funct (@out first : ref Int, @out second : ref Int) -> do {
    first <- Random.int(0, 100);
    second <- Random.int(0, 100);
    if (first < second) {
        return second;          // 注意这里返回的是一个引用
    }
    else {
        return first;
    }
};

const main = funct () -> do {
    auto max = random_max(x : Int, y : Int);
    print "The max of $(x) and $(y) is $(max)";
};
```

5.4.3 捕获变量

函数中可以出现外部作用域中的变量，此时它们相当于通过值传递被**捕获（Capture）**到函数体当中，此时相当于在函数体中创建了一个变量，它初始化为这个被捕获变量的值。

```
const main = funct () -> do {
    auto x = 42;
    auto f = funct (y : Int) -> x + y;  // x 是被自动捕获的变量
};
```

不过，被捕获变量总是不可变的，且无法自动捕获任何引用类型的变量。

```
const main = funct () -> do {
    auto x = 42;
    auto y = mut 42;
    auto r = ref x;
    auto f = funct () -> x <- 24;   // 编译错误，x 不可变
    auto g = funct () -> y <- 24;   // 编译错误，y 不可变
    auto g = funct () -> r;         // 编译错误，r 不能被自动捕获
};
```

不允许默认捕获引用类型变量有两个主要的原因：其一，引用目标的生命周期可能短于函数的生命周期（看下面的例子）；其二，可以通过引用改变函数外部的环境，这会损害函数的*纯洁性*，即是否在相同参数调用下有相同行为[13]。

```
const f = funct (x : Int) -> do {
    auto r = ref x;
    return funct () -> r;   // 编译错误，r 不能被自动捕获
};
const main = funct () -> do {
    auto r = f(42);         // 如果允许自动捕获引用，这里相当于得到了 f 中 x 的引用
    print r;                // 引用目标已经被释放了，这里是一个非法的内存访问
};
```

为了捕获变量的引用，可以使用 `@capture` 属性配合查询结构来实现。

```
const main = funct () -> do {
    static s = "Hello, world!";
    auto f = funct @capture[r = ref s] () -> r;
};
```

`@capture` 的原理类似于将引用装箱到一个临时的复合类型中，然后将这个复合对象值捕获。如果要手写实现这个功能会麻烦许多，但让我们作一个简单的展示：

```
const main = funct () -> do {
    static s = "Hello, world";
    auto wrapped_s : struct (r : ref String) = ref s;
    auto f = funct () -> wrapped_s.r;
```

[13] 当然，这只是一个语法盐而已，类似于默认所有变量不可变，函数也应该默认是纯的；如果想使用不纯的函数，应该显式使用 `static` 变量或下面介绍的 `@capture` 属性。

```
};
```

注意到我们在示例中将 s 声明为 static。这也是 @capture 对引用类型的一个额外要求：被引用的变量必须不是自动存储期的。不过有时，我们希望将局部作用域中的变量移动到函数作用域中，此时可以使用 move 提示符，它的作用类似于 with 子句，不过目标是一个函数作用域。

```
const main = funct () -> do {
    auto s = "Hello, world";
    auto f = funct @capture[r = ref move s] () -> r;
};
```

经过移动的变量不再有声明周期问题。

第六章 结构

内容提要

- 结构表达式
- 成员
- 成员函数
- 结构属性
- 结构作用域
- 构造函数
- 析构函数
- 垃圾回收
- 静态成员
- 常量成员
- object 表达式
- 查询结构的本质
- 部分对象
- missing 关键字
- 结构对象的格式化

6.1 结构表达式

结构（Structure）是构建元组的模版，声明结构可以用下面的语法进行。

语法 6.1 (结构表达式)

```
[var-decl-list]
```

♣

因此，结构表达式是一系列变量声明的组合。

```
const Point = struct (x : Real, y : Real);
const WrappedInt = struct val : Int;
```

可以看到，圆括号的作用依然是用于运算符分组。为了表述更加清晰，我们总是会在 struct 后使用圆括号。特别地，空的变量声明列表会定义一个内存占用为零的结构，此时的圆括号不能省略。

```
const Empty = struct ();
const (Empty1, Empty2) = struct (), struct ();
```

和函数类似地，如果提前声明结构的内存布局，就可以在定义的时候省略这个提示符。

```
@layout(Int, Int)
const Point : Type;
const Point = (x : Int, y : Int);
```

这里我们利用了 @layout 属性来提示编译器该类型的内存布局。或者，我们也可以提前声明结构的大小，此时也可以省略 struct 关键字。

```
@size(16)
const Point : Type;
const Point = (x : Int, y : Int);
```

无论是 @layout 还是 @size 属性，都要求声明的布局和大小与定义完全一致，否则会产生编译错误。

```
@size(0)
const Point : Type;
const Point = (x : Int, y : Int);       // 编译错误，实际大小和声明大小不一致
```

提前声明结构对象大小有一系列好处，比如我们可以未定义结构具体信息的情况下使用这个结构。

```
@size(16)
const Point : Type;

const Segment = struct (
    begin : Point,
    end : Point
);
```

这是因为，使用一个类型时只需要知道该类型的大小。这可以让循环引用在一些情况下可以出现：

```
@size(8)
const A : Type;
@size(8)
const B : Type;

const A = (b : B);
const B = (a : A);
```

不过它们无法保存任何有效信息。

6.2 成员

6.2.1 成员的本质

结构中声明的变量称为**成员（Member）**。它的本质是一个拥有修饰名字的函数。

```
const Point = struct (x : Real, y : Real);

// 编译器会声明下面的函数
const ::Point::x : ref mut? Point -> ref mut? Real = this -> (decay this)[0];
const ::Point::y : ref mut? Point -> ref mut? Real = this -> (decay this)[1];
```

这里出现的结构 `::Point::x` 正是我们上一章介绍的*修饰名字*，这也暗示了结构本身构成了一个作用域。上面用到的内置函数 `decay` 会将结构类型的对象转化为相同内存布局的元组引用类型，即 `(Real, Real)` 和它的*兼容类型*；在这个过程中 `Point` 对象并没有被复制出来，而是通过它的引用构建了相同内存布局的元组引用[1]。

在我们利用点调用符访问成员时，编译器也会将其变为对应函数的调用。

[1]这个过程和上一节中我们在模式匹配部分见到的例子 `ref tup` 效果相同。熟悉 C++ 的读者可以将其看作两个指针类型的 `reinterpret_cast`，由于它们的内存布局完全相同，其行为是符合预期的。

```
const Point = struct (x : Real, y : Real);

const main = funct () -> do {
    auto p : Point = (42, 0);
    print p.x;                  // 输出 42

    // 编译器会生成下面的代码
    print ::Point::x(p);
};
```

我们可以主动使用修饰名字来指定要调用的函数。

```
const Point = struct (x : Real, y : Real);

const main = funct () -> do {
    auto p : Point = (42, 0);
    const x = funct (p : Point) -> 24;
    print x(p);                         // 输出 24
    print ::Point::x(p);                // 输出 42
};
```

有关修饰名字的更多信息，我们会在高级特性中详细说明。

6.2.2 成员函数

在 C++ 等其它语言中，可以在结构上定义**成员函数（Member Function）**。不过我们在上一章已经学到过，只需为任何方法声明参数名称为 `this` 就可以像成员函数一样调用它。这里来复习一下。

```
const Vec = struct (x : Real, y : Real);
const sub = funct (this v : Vec, other : Vec) -> (v.x - other.x, v.y - other.y) as Vec;

const main = funct () -> do {
    auto v : Vec = (1.0, 2.0);
    auto u : Vec = (1.0, 1.0);
    print v.sub(u);             // 输出 0.0, 1.0
};
```

这里 sub 的修饰名字是 `::Vec::sub`，因此带有 `this` 参数的函数被声明在 `Vec` 的作用域中。

注意，我们也可以在结构中声明函数，但它对所在结构中其它的变量一无所知；其地位和所有其它成员相同。

```
const Vec = struct (
    x : Real, y : Real,
    transform : Vec -> Vec
);

const main = funct () -> do {
    auto v : Vec = (1.0, 2.0, funct (v : Vec) -> v);
    print v.transform(1.0, 2.0);                        // 输出 1.0, 2.0
};
```

注意到上面的 `v.transform` 在 AutoScript 中有两种潜在的含义：其或者是基于包含 `this` 参数的函数，或者是一个成员（这里的含义）。我们需要避免成员名称和函数名称雷同。

```
const Foo = struct (bar : Int);
const bar = funct (this f : Foo) -> 42;     // 编译错误，::Foo::bar 重复定义。
```

6.2.3 默认初始化

回忆 `default` 可以让内置类型得到一个默认的初始值，这对结构类型对象也有类似的效果：对象中每个成员都会通过 `default` 来初始化。通过这种方式我们可以简洁地构建一个结构类型变量。

```
const Point = struct (x : Real, y : Real);

const main = funct () -> do {
    auto origin : Point = default;
};
```

和函数参数类似地，我们可以用 `@default` 属性为其设置默认值：

```
const Point = struct (
    @default(1.0) x : Real,
    @default(2.0) y : Real
);

const main = funct () -> do {
    auto p : Point = default;
    print p.x;                  // 输出 1.0
    print p.y;                  // 输出 2.0
}
```

设置缺省属性后，如果我们为这个成员初始化为 `default`，它会自动用设定的值代替。不过，此后再次用 `default` 为其赋值时，会统一按照内置类型的缺省值。

和缺省属性功能类似的便是缺省声明。我们可以用查询结构来初始化一个结构类型变量。

```
const Point = struct (
    x ?? 1.0 : Real,
    y ?? 2.0 : Real
);

const main = funct () -> do {
    auto p : Point = [x = 42.0];
    print p.y;                  // 输出 2.0
};
```

我们可以省略类型标注，不过这实际上定义了一个结构模版；当缺省声明没有被使用时，它可以是任何类型：

```
const Point = struct (
    x = 1.0,
    y = 2.0
);

const main = funct () -> do {
    auto p : Point = [x = 42];
    print typeof(p.x);          // 输出 Int
};
```

6.2.4 结构属性

在结构中可以利用 `@prop` 属性来定义结构**属性（Property）**。

语法 6.2 (结构属性 [property-attr])

1. `@prop ([get-funct-name], [set-funct-name])`
2. `@prop`

♣

用 `@prop` 修饰的成员被称为一个结构属性，或（在不和 `@` 开头的属性有歧义的情况下）简称为属性[2]。

```
const Person = struct (
    name : String,
    @prop(get_greeting, undefined) greeting : String
);

const get_greeting = funct (this : ref mut? Person) -> "Hello, $(this.name)";
```

在结构属性的声明处不需要给出访问辅助函数的类型，这是因为它们的类型是确定的。对于在结构 `S` 中声明为 `T` 类型的属性，它的两个辅助函数类型一定是：

```
getter : ref mut? S -> ref T;
setter : (ref mut? S, ref? T) -> ref mut? T
```

当使用点调用符访问结构属性时，根据其所在位置会分别调用 `getter` 和 `setter`。

[2]这其实是中文翻译的缺陷。无论是用于修饰的属性（Attribute）和结构属性（Property）都被翻译成“属性”。

```
const Foo = struct (
    @prop(getter, setter) x : Int
);
// 这里省略 getter 和 setter 的定义
const main = funct () -> do {
    auto foo : mut Foo = 42;
    print foo.x;
    foo.x <- 24;
    // 编译器会类似生成下面的代码
    print getter(foo);
    setter(foo, 24);
};
```

实际上编译器对 AutoScript 中的结构属性有特定的操作方式，其中利用了辅助类型 `Prelude.Proxy`，简单来说就是让点调用返回一个特殊类型的变量，它在被赋值或赋值给其它对象时有不同的行为。我们会在 AutoScript 标准库的章节中详细介绍。

由于结构属性依赖于 `getter` 和 `setter`，并不一定需要一个内存位置，因此 AutoScript 提供了 `@skip` 属性来让编译器不将其视作内存位置。

```
const Vec2 = struct (
    x : Real,
    y : Real,
    @prop(length, undefined) length : Real
);

const main = func () -> do {
    print decayof(Vec2);        // 输出 (Real, Real)
};
```

这里使用的 `decayof` 是作用在类型上的内置函数，其得到相同内存布局的元组类型。

我们可以为结构的属性设置缺省值，或者通过默认属性，或者通过默认声明。它们能使用的前提是结构属性拥有内存位置。

```
const Foo = struct (
    @default(42) @prop(a_getter, undefined) a : Int;
    @default(24) @skip @prop(b_getter, b_setter) b : Int;
);

const main = funct () -> do {
    auto foo : Foo = default;
    print foo.a;                // 输出 42
    print foo.b;                // 输出取决于 b_getter，不一定是 24
};
```

6.3 成员的生命周期

结构作用域是一个虚拟的作用域，当我们定义结构类型的对象时，所有其中的成员都会拥有和该对象相同的生命周期，其所在的作用域便是结构对象所在的作用域。

```
const Point = struct (x : Real, y : Real);

const main = funct () -> do {
    auto p1 : Point = (1.0, 0.0);   // p.x 和 p.y 的作用域是 p1 的作用域，生命周期与 p1 相同
    static p2 : Point = (1.0, 0.0); // 同样，作用域是 do 表达式的语句块，生命周期与 p2 相同
};
```

下面让我们具体介绍成员的声明周期。

6.3.1 构造和析构

结构类型变量的初始化除了可以通过相同内存布局的元组隐式转化外，可以通过属性 `@ctor` 来指定**构造函数（Constructor）**。

```
@ctor
const Complex = struct (real : Real, imag : Real);

const main = funct () -> do {
    auto c = Complex (1.0, 2.0);    // 可以利用类似函数调用的语法创建对象
};
```

除此之外，也可以将任意的函数注明 `@ctor` 属性，此时它会称为其返回类型的构造器。

```
import Math;
@ctor
const Complex = struct (real : Real, imag : Real);
@ctor
const unit_circle = funct (theta : Real) -> (Math.cos(theta), Math.sin(theta)) as Complex;

const main = funct () -> do {
    auto c = Complex(Math.PI);      // 调用了 unit_circle
};
```

当结构类型拥有多个构造函数时，编译器会根据传入参数的类型来推断应该调用哪个具体的函数；唯一的特例是结构变量本身，我们不需要为其定义函数。某种角度来看，这里的行为是一种更强的“函数重载”[3]。

下面是编译器处理构造函数的近似行为：

[3]在 C++ 等支持函数重载的语言中，所有的函数都可以像这样重载。不过这会导致非常复杂的重载决议，以及可能导致难懂的函数底层接口。因此在 AutoScript 中，只对构造函数开放全参数类型的重载。

```
@ctor
const Foo = struct ();
@ctor
const ctor_1 = funct (x : Int) -> () as Foo;
@ctor
const ctor_2 = funct (x : Real) -> () as Foo;

// 编译器会生成下面的代码
const Foo__ctor = funct (...args) -> do {
    // 按照声明顺序依次尝试调用
    try return ctor_Foo(...args);
    try return ctor_1(...args);
    try return ctor_2(...args);
};
```

AutoScript 中得益于垃圾回收机制，我们通常不需要手动回收任何资源，不过我们依然可以定义析构函数让结构类型对象在被释放前调用它。

```
const Person = struct (name : String);
@dtor
const farewell = funct (ref mut? Person) -> print "Goodbye~";

const main = funct () -> do {
    auto p : Person = "Alice";
};                              // 自动存储期变量再此被释放，释放之前会调用 farewell 函数
```

和构造函数不同的是，析构函数只能存在一个。对于复合对象，系统会首先按顺序析构所有子对象，随后再析构复合对象本身。

6.3.2 非自动存储期的成员

默认情况下成员都拥有自动存储期（它们所在的作用域即是结构作用域），不过我们也可以声明其其指定静态或常量作用域。

```
const Static = struct (
    static value : mut Int
);

const main = funct () -> do {
    auto s1 : Static = ();
    auto s2 : Static = ();
    s1.value = 42;
    print s2.value;         // 输出 42
};
```

静态成员的本质是一个拥有静态变量的函数。因此无论从哪个变量去访问都可以得到相同的对象。静态成员不参与到结构的内存布局中。

```
const Static = struct (
    static value : mut Int
);

// 编译器会生成下面的代码
const ::Static::value = funct (@unused this : ref mut? Static) -> do {
    static value : mut Int;
    return ref value;
};
```

常量成员和静态成员有类似的原理。

```
const TrueType = struct (
    const value = 42
);

// 编译器会生成下面的代码
const ::TrueType::value = funct (@unused this : ref mut? TrueType) -> do {
    const value = 42;
    return ref value;
};
```

最后，让我们简单着重介绍动态存储期的成员。AutoScript 中用 new 限定符来声明一个动态存储期的变量。默认情况下它是受垃圾回收器托管的，因此如果在其所在作用域中我们没有将其引用传递到其它作用域，那么在当前作用域结束后由于引用计数归零，它和自动存储期变量一样会被回收。

```
const main = funct () -> do {
    new x = 42;
};                              // 作用域结束时 x 被自动释放
```

AutoScript 也提供了能够手动控制内存的方式，我们需要为变量使用 @manual 属性。此时垃圾回收器会放弃对这个变量的托管，我们需要用 delete 关键字手动将其释放。通常会在析构函数中进行。

```
const MyInt = struct (
    new value : Int;
);

@dtor
const destroy_int = funct (i : ref MyInt) -> do {
    delete i;
};

const main = funct () -> do {
    auto i : MyInt = 42;
};                              // 在这里调用了析构函数，手动释放了动态存储期的变量
```

值得说明的是，即使是受托管的动态存储期变量，我们依然可以用 delete 语句将当前的引用删除；如果此时不存在任何这个变量的引用，则会释放它的内存。

内存被释放之后的动态存储期变量，其值是 invalid。我们无法再对其进行任何操作。

```
const main = funct () -> do {
    new x = 42;
    delete x;
    print x;                    // 运行期错误，x 处于非法状态
};
```

6.4 自定义对象

6.4.1 object 表达式

AutoScript 中，除了通过 struct 关键字构建结构后声明变量，也可以直接通过 object 关键字构建自定义结构的变量。

语法 6.3 (对象表达式)

```
object [block]
```

在对象表达式的语句块中，所有的自动存储期和动态存储期变量声明都会默认看作生成对象的一个内存位置。

```
const main = funct () -> do {
    auto p = object {
        x = 42;             // 第一个内存位置
        new y = 24;         // 第二个内存位置
    };
    print p.x;              // 输出 42
};
```

可以猜到，对象表达式会让编译器生成一个匿名的结构，其对应的内存位置初始化为给定的表达式值。

```
const main = funct () -> do {
    auto p = object {
        x = 42;
        new y = 24;
    };

    // 编译器会生成下面的代码
    const p__object = struct (
        @default(42) x : Int,
        @default(24) new y : Int
    );
    auto p : p__object = default;
};
```

对象表达式可以用来方便地生成临时的复合类型对象，不过由于它的类型难以复用（需要利用 typeof 关键字），所以如果需要超过一次使用该类型的场合，还是应该显式写出结构声明。

对象表达式中的*初始化*顺序决定了编译器生成类型内存位置的顺序。因此即使提前声明一个成员，它只会在初始化时加到内存位置中。

```
const main = funct () -> do {
    auto p = object {
        auto y : Int;
        auto x = 42;
        auto y = 24;
    };

    // 编译器会生成下面的代码
    const p__object = struct (
        @default(42) x : Int,
        @default(24) y : Int
    );
    auto p : p__object = default;
};
```

在对象表达式中我们可以使用所有的语句，不过需要特别小心自动存储期变量的声明（它会被默认作为对象的一个内存位置）。

```
const main = funct () -> do {
    auto p = object {
        scan x : Int;       // 注意这里暗含了一个自动存储期变量的声明
        scan y : Int;
    };
};
```

至于对象表达式中的其它存储期变量，其行为和结构中其它存储期成员的行为类似。

6.4.2 查询结构的本质

AutoScript 中支持查询结构。从本质来讲它是对象表达式的语法糖。下面让我们罗列在查询结构中，方括号里可以出现的内容：

- 表达式（包括已经声明的变量）[4]，此时相当于声明了一个匿名的变量，它初始化为这个表达式的值。
- 变量定义，此时相当于定义了一些列变量，并按照声明顺序设立内存位置。
- 模式声明，此时相当于声明了一个约束，用来筛选满足当前模式声明的成员 [5]。

请参考下面的例子：

```
const main = funct () -> do {
    auto q1 = [0, x = 3];           // 定义了内存布局为 (Int, Int) 的查询结构，其中第一个位置是匿名成员
    auto q2 = [(x, y) = decay q1];  // 在查询结构中使用模式匹配来初始化其中的成员
    auto q3 = [? 42];               // 在查询结构中使用模式匹配来构建一个约束
};
```

[4]这里是和模式声明有歧义的地方：`x` 可以理解为一个原子模式声明，也可以理解成一个原子表达式。在查询结构中，如果编译器能找到这个名字，则将其视作表达式，否则才视作模式声明。这和其它情景下的默认行为不同。

[5]有关约束我们会在模式匹配的章节中详细说明

可以看到，前两种形式都在查询结构中创建了拥有内存位置的成员，而最后一种仅仅用来给出查询信息。下面让介绍查询结构作为查询信息的功能。

```
const main = funct () -> do {
    auto p = object { x = 42, y = 24 };
    auto q = mut (1, 2, 3);
    print p[:? Int];                    // 输出 42, 24
    print p[? (\x -> x `mod` 2 == 0)];  // 输出 2（这里需要括号因为 ? 的优先级高于 ->）
    q[0, 2] <- 0;
    print q;                            // 输出 0,  2,  0
};
```

上面的例子中，我们通过查询结构 `[0, 2]` 来“同时”访问 `q` 的第一个和最后一个元素，并一起对它们赋值。这里逗号蕴含了“或”的意义：逗号分隔的多个表达式或约束中，只需要有一个满足就能被筛选出来参与后续的操作。如果想要表示“与”的意义，可以使用连续多个查询结构（比如 `q[? m][? n]`）来同时要求多种约束。

当查询的条件比较复杂时（析取一个合取式），我们可以使用嵌套的查询结构。

```
const main = funct () -> do {
    auto x = object { x = 42, y = 0, z = 2.0 };
    print x[:? Real, [:? Int][? (\x -> x > 0)]];    // 输出 42, 2.0
};
```

合取一个析取式会简单很多：

```
const main = funct () -> do {
    auto x = object { x = 42, y = "abc", z = 0.0 };
    print x[:? Int, ?: Real][? (\x -> x > 0)];      // 输出 42
};
```

6.4.3 部分对象

在上节中我们介绍了可以用来得到对象筛选内容的查询结构，但这也就涉及到一个有趣的问题：经过查询结构后我们得到的类型是什么？如果筛选出的内容编译期可知（比如用下标筛选，或使用类型约束等），那么我们可以得到一个确切类型的对象，不然在编译期一无所知。事实上，通过查询对象得到的是一个**部分对象（Partial Object）**，其类型通过类模版 `Partial` 表示。

```
const Pair = struct (
    x : Int,
    y : Int
);

const main = funct () -> do {
    auto p : Pair = (1, 0);
    print typeof(x[0]);             // 输出 Partial[Pair]
};
```

部分对象本质上存储了其对应的普通对象每个子对象的引用。以上面的 `Pair` 为例，`Partial[Pair]` 中存储了 `ref mut? Int` 和 `ref mut? Int` 两个对象。这些引用可能取值为 `missing`，此时对应了该对象在此是空位。

```
const Pair = struct (
    x : Int,
    y : Int
);

const main = funct () -> do {
    auto p : Pair = (1, 0);
    auto px = p[? 1];
    print px.query(Pair::x);        // 输出 1
    print px.query(Pair::y);        // 输出 missing
};
```

这里我们使用的 query 函数用来得到部分对象中，原对象的成员的值。值为 missing 的各个对象，在任何操作的时候都会被忽略[6]。对部分对象使用任何运算符时，会依次访问其中元素：

```
const main = funct () -> do {
    auto p : Partial = (1, 2);      // 构建了一个部分对象（虽然它实际上是完全的）
    auto q : Partial = (3, 4);
    print p * q;                    // 输出 3, 4, 6, 8
};
```

这实际上源于 Partial 的实例类型属于 Monad 类族。我们会在类族的章节中详细介绍这个性质。

6.4.4 undefined 和 missing

至此，我们遇到了两种表示“未定义”的字面量，undefined 和 missing。前者是所有未经初始化变量的默认值，在求值时会中断程序；后者则是对象经过查询结构筛选后产生的“空洞”，在求值时会被跳过。本节让我们考察在一些场景中，将 undefined 替换为 missing 会有何效果。

```
const main = funct () -> do {
    auto x : Int = missing;
    print x;                    // 无事发生
    auto y = x + 10;            // y 也变成 missing
    print x.foo();              // 编译错误，Int 上没有定义 foo 函数
};
```

可以看到，拥有 missing 值的变量依然可以像正常那样进行操作，不过表达式的结果一律会变成 missing。

此外，前面介绍过的缺省声明和缺省属性对 missing 一样会触发。

```
const main = funct () -> do {
    auto x ?? 42 : Int = missing;
    print x;                        // 输出 42
};
```

[6]它可以理解为安全的 undefined，后者一旦求值则产生运行时错误。

第七章 抽象数据类型与模式匹配

内容提要

- ADT 表达式
- 标签
- 模式声明
- 原子模式
- 元组模式
- 修饰模式
- 选择模式
- 约束声明
- 约束的合取与析取

7.1 抽象数据类型

7.1.1 ADT 表达式

AutoScript 中用 **抽象数据类型（Abstract Data Type, ADT）** 来表示抽象的结构化数据类型。其语法如下：

语法 7.1 (ADT 表达式 [adt-expr])

```
data [tag-choices]
```

♣

其中的每个类型标签都有下面的语法：

语法 7.2 (ADT 标签声明 [adt-tag-decl])

1. [tag-id] [var-decl-list]
2. [tag-id]

♣

其中第二种是不带任何参数的标签，可以理解为带一个 Void 参数的标签。相邻的标签用 | 运算符相隔，这和选择类型的语法相似（实际上两者的本质也是相同的）。下面是几个例子：

```
const Bool = data (True | False);
const MaybeInt = data (Value Int | Null);
const Color = data (RGB (r ?? 0.0 : Real, g ?? 0.0 : Real, b ?? 0.0 : Real) | NoColor);

const main = func () -> do {
    auto b = True;
    print typeof(b);        // 输出 Bool
    print tagof(b);         // 输出 True

    auto i = Value 42;
    print typeof(i);        // 输出 MaybeInt
    print tagof(i);         // 输出 Value

    auto c = RGB [r = 0.4, b = 0.6];
    print typeof(c);        // 输出 Color
    print tagof(c);         // 输出 RGB
};
```

可以看到，ADT 是一个选择类型和结构类型的组合体：每个标签都可以声明为一个结构类型，然后 ADT 是至少两个标签组成的选择类型。和结构表达式类似地，上面的圆括号并非必须，但为了避免运算符优先级造成的混乱，我们习惯上使用圆括号。

7.1.2 标签的本质

从标签的用法我们可以看出，它类似于一个函数。但更本质来说，它是一个使用了 `@ctor` 属性的结构对象。

```
const MaybeInt = data (Value Int | Null);

// 编译器会生成下面的代码
@ctor(MaybeInt)
const Value = struct (
    int : Int
);
@ctor @eager
const Null = struct ();
```

这里的 `@eager` 属性可以让函数会自动被调用，其要求函数接收 `Void` 为参数。

因此 ADT 是一系列结构对象组成的选择类型，后者的很多性质同样适用于 ADT，比如强制类型转换。上面使用的 `@ctor` 带了一个参数，它相当于指定了构造函数生成对象的类型（不是默认的 `Value` 而是指定的 `MaybeInt`），这也是为什么我们直接通过 `Value 42` 就可以构建一个 `MaybeInt` 对象。

```
const MaybeInt = data (Value Int | Null);

const main = funct () -> do {
    auto i = Value 42;
    print i as Value;       // 输出 Value 42
};
```

标签是一个运行期的性质，编译器无法保证在编译期得到对象的标签性质。我们可以通过 `tagof` 对类型的性质进行动态指派。

```
const MaybeInt = data (Value Int | Null);

const f = funct (x : MaybeInt) -> do {
    if (tagof(x) == Value) {
        print "Value is $(x as Value)";
    }
    else {
        print "Nothing";
    }
};
```

不过这样的可读性比较差。我们通常会使用模式匹配对 ADT 类型对象进行操作。

7.1.3 选择类型的模式匹配

选择类型，包括 ADT 结构可以通过模式匹配来读取其中的值。

```
const MaybeInt = data (Value Int | Null);

const f = funct (x : MaybeInt) -> do {
    match (x)
        Value i -> print "Value is $(i)",
        Null -> print "Nothing";
};
```

上面我们用 `Value x` 来匹配 `MaybeInt` 作为 `Value` 时的结构，并声明了一个变量 `i` 绑定其中存的值。特别地，`Null` 后面不加任何变量。

这种语法可以直接用于普通的选择类型，这是因为选择类型中的标签正是其自身（以 `Int | String` 为例，其类似于一个 `data (Int Int | String String)`。

```
const f = funct (x : Int | String) -> do {
    match (x)
        Int i -> print "An integer $(i)",
        String s -> print "A string $(s)";
};
```

配合上一章介绍的 `guard` 表达式，我们可以简洁地操纵选择类型对象。

```
const f = funct (x : Int | String) -> do {
    x <| (guard
        Int i -> print "An integer $(i)",
        String s -> print "A string $(s)");
};
```

在之后的小节中，我们将正式且详细地介绍 AutoScript 的模式匹配机制。

7.2 模式声明

模式匹配（Pattern Matching）是 AutoScript 中无处不在的语言机制。其最典型用在实体声明和函数声明。模式匹配顾名思义，需要存在一个*模式*和一个待匹配的实体。在实体声明中，出现在 `=` 左侧的是模式，右侧则是一个待匹配的实体。函数声明中，出现在 `->` 左侧的是模式，右侧则是函数体；在函数被调用时，参数就成为了待匹配的实体。

语法 7.3 (模式匹配)

1. `[pattern] = [init-expr];`
2. `[pattern] -> [body]`

♣

AutoScript 中的模式中，存在三种不同作用的结构，其一是用于声明变量的部分，也被称为*声明器*；其二是用于限制匹配内容的*约束*；其三是暗示匹配结构的*模式声明*。后两者是模式匹配的重点：它判断待匹配实体是否拥有特定结构并满足给定约束，如果是则利用声明器将匹配的结构或子结构声明为变量。

本节让我们先熟悉 AutoScript 中所有模式声明的语法。

7.2.1 原子模式

原子模式中，模式声明里不存在子模式，此时最多能声明一个变量。下面是原子模式中所有可能出现的语法：

语法 7.4 (原子模式 [atomic-pattern])

1. `otherwise`
2. `()`
3. `[constr] [default-decl]`
4. `[storage-spec] [id] [default-decl]`
5. `[storage-spec] [id] [default-decl] [constr]`
6. `[storage-spec] [id] [default-decl] [type-decl]`
7. `[storage-spec] [id] [default-decl] [type-decl] [constr]`

♣

其中 `[constr]` 指的是模式中的约束，让我们在下一节中再详细介绍。下面是对这些语法的逐个说明：

1. 跳过类型推断[1]，可以匹配任何实体，但此时不声明任何变量。
2. 匹配空对象 `()`，此时不能声明任何变量[2]。
3. 一个约束，不声明任何变量。
4. 无约束的自动类型推断，可以匹配任何实体并声明一个变量。
5. 带有约束的自动类型推断，会将待匹配的实体传给约束判断其返回结果是否为真，若是则声明一个变量。
6. 无约束的显式类型声明，可以匹配任何实体并将其隐式转化为给定类型后声明一个变量。
7. 带有约束的显式类型声明，会将待匹配的实体传给约束判断其返回结果是否为真，若是则将实体隐式转化为给定类型后声明一个变量。

原子模式是最常见的模式声明，其主要出现在变量的声明语句和函数的参数类型声明[3]。

```
const main = funct () -> do {
    auto x = 42;                    // 原子模式 auto x
    auto f = funct (x : Int) -> x;   // 原子模式 auto f 和 x : Int
};
```

原子模式中最为特殊的是 `otherwise` 这个占位符字面量。从其性质来看，它是一个极强的“混子”，所有和它相关的判断都会返回真。让我们尝试在其它地方使用 `otherwise`。

```
const main = funct () -> do {
    auto x : Int = otherwise;       // 编译错误，otherwise 不能作为常规表达式求值
    auto otherwise = 42;            // 无事发生，没有真的 otherwise 作为变量被声明
    print otherwise;                // 编译错误，otherwise 不能作为常规表达式求值
};
```

从上面的例子可以看出，`otherwise` 仅仅是模式声明中一个特殊的标识符，代表一个默认的匹配情形[4]。

[1] 包括 `undefined` 的情形，因此 `otherwise` 会造成函数是不严格的。**严格函数（Strict Function）**要求 `undefined` 输入一定会导致 `undefined` 输出。考虑 `otherwise -> 42`，这显然不是一个严格函数。这个特点和 Haskell 完全一样。不过我们可以类似地使用不加约束的原子模式，如 `x` 来强制求值。

[2] 敏锐的读者可能发现了这里的不和谐之处：`()` 是一个对象，但为什么只有它被单独列出来？其它的譬如 `42` 不能直接出现么？实际上，它们可以以约束的形式出现，比如 `? 42`，而 `()` 的特殊性是为了语言简洁性妥协的：我们避免类似于 `?()  -> ...` 的函数出现。

[3] 实际上，变量的提前声明对应了第 6 种情形，这也是唯一允许的提前声明语法。

[4] 可以类比 C++26 中的占位符 `_`。

7.2.2 元组模式

元组模式用于匹配一个形成元组结构的一系列模式。

语法 7.5 (元组模式 [tuple-pattern])

1. [pattern-list]
2. ...[pattern]

♣

这里 [pattern-list] 指的是由逗号分隔的多个 [pattern]。

1. 完全匹配的元组模式，将元组中特定元素一对一匹配出来。
2. 模糊匹配的元组模式，将元组匹配为一个**包 (Pack)**。包中声明的所有变量都将带有可展开性，只需利用包展开运算符 ... 即可将其展开为一个元组。

```
const main = funct () -> do {
    auto (x, ...xs) = (1, 2, 3);     // 元组模式 (x, ...xs)，其中包含了另一个元组模式 ...xs
    auto ...ys = ...xs;              // 元组模式 ...ys，右侧通过 ... 运算符将包展开为元组
    print ...ys;                     // 输出 2, 3
};
```

我们有必要详述包的特性。首先，包的类型并不是一个元组，而是一个定义为内部实现的库类型 Prelude.Pack。如上面所示，可以通过 ... 运算符将一个包转换为一个元组。不仅如此，对包进行的任何其它运算都会被视为对包中每个实体的运算。

```
const main = funct () -> do {
    auto tup = (1, 2, 3);
    auto ...xs = mut tup;
    xs <- xs - 2;                // 对包中每个元素 x 都进行 x <- x - 2
    print xs;                    // 输出 -1, 0, 1
};
```

此外，包还支持折叠表达式。参考下面的说明：

- (pack @ ...)：左折叠，即展开为 (...((arg1, arg2) @ arg3) @ ...)。
- (... @ pack)：右折叠，即展开为 (arg1 @ (arg2 @ (arg3 @ ...)...)...)

通过这种方式，我们可以轻松地求出一个数组中元素的和：

```
const main = funct () -> do {
    auto arr = (1, 2, 3);
    print (arr + ...);       // 输出 6
};
```

或者依次处理一个包中的元素：

```
const print_all = funct (...args) -> do {
    auto f = guard
        (x : Int) -> "This is an integer",
        (x : Real) -> "This is a real number",
        (x : String) -> "This is a string",
        otherwise -> "This is something else";
    (f(args), ...);
};
```

这里其实用到了 AutoScript 中的一个规则：在元组中每个元素会按照从左到右的顺序依次求值。

7.2.3 修饰模式

类型修饰符可以用来形成模式，包括 atom、dyn、mut、ref。

语法 7.6 (修饰模式 [qualifier-pattern])

1. atom [pattern]
2. dyn [pattern]
3. mut [pattern]
4. ref [pattern]

1. 匹配一个原子实体，将其原型以别名方式取出来并通过 [pattern] 匹配。
2. 匹配一个动态实体，将其原型以别名方式取出并通过 [pattern] 匹配。注意这里因为没有给出具体的实例类型，所以 [pattern] 可能被多种实例类型匹配。
3. 匹配一个可变实体，将其原型以别名方式取出来并通过 [pattern] 匹配。
4. 匹配一个引用实体，将其引用目标以别名方式取出并通过 [pattern] 匹配。

模式到子模式的过程总是以别名方式构建的，这可以保证模式匹配中最多生成一个新的对象，避免内存的浪费。如果依然需要将父模式声明为变量，可以在 [pattern] 的最前面加上一个整体声明：

语法 7.7 (整体声明 [total-decl])

[id]~[pattern]

下面是一些例子：

```
const main = funct () -> do {
    auto x = mut 42;
    auto y~(mut n) = x;      // 修饰模式 mut n，并使用了整体声明 y
    print typeof(y);         // 输出 mut Int
    print typeof(n);         // 输出 Int
};
```

总体声明出来的变量和它进一步匹配的模式声明之间是别名的关系，比如上面的例子中，如果我们修改了 y 的值，会影响 n 的内容。不过不要因此弄混了模式声明和待匹配对象之间的关系：等号左边是右侧的复制，因此虽然

7.2.4 可选修饰模式

上节介绍的修饰模式有一个变种，即可选修饰模式，它使用 atom?、dyn?、mut? 和 ref? 来选择匹配一个可能出现的修饰符。

语法 7.8 (可选修饰模式 optional-qualifier-pattern)

1. atom? [pattern]
2. dyn? [pattern]
3. mut? [pattern]
4. ref? [pattern]

这种模式可以灵活地匹配任何表达式，且去除这个修饰符。

```
const main = funct () -> do {
    auto x = 42;
    auto y = mut 42;
    auto mut? z = x;
    auto mut? w = y;
};
```

这种模式和使用 `demut` 等函数相比，更加偏向声明式；如果有命令式的代码习惯，使用 `demut` 也无妨。下面是一个对比：

```
const f = funct (input : mut? Int) -> do {
    auto mut? x = input;
    auto y = demut input;
};
```

7.2.5 选择模式

正如上一节中我们介绍的，选择类型和抽象数据类型可以通过选择模式匹配。

语法 7.9 (选择模式 [choice-pattern])

1. `[tag-id] [pattern]`
2. `[tag-id]`

1. 对应了有参数的标签，将标签包含的实体复制出来并通过 `[pattern]` 匹配。
2. 对应了无参数的标签。需要注意这和 `[tag-id] ()` 是不同的：尽管它们都匹配了内存布局为 `()` 的标签类型，但前者对应了拥有 `@eager` 属性的构造函数，后者则没有该属性。

```
const main = funct () -> do {
    scan x : Option[Int];
    match (x)
        Some e -> print e           // 选择模式 Some e
        Null -> print "Nothing";    // 选择模式 Null
};
```

7.3 约束

7.3.1 约束声明

在模式匹配中，我们会使用**约束（Constraint）**来对待匹配实体进行额外限制。约束分为*值约束*和*类型约束*，顾名思义前者是待匹配实体值的*谓词*，后者则是其类型的*谓词*。所谓**谓词（Predicate）**，指的是返回可隐式转换为 `Bool` 类型的函数。

语法 7.10 (约束 [constr])

1. `? [pred]`
2. `:? [pred]`

上面的 ? 称为值约束提示符，:? 称为类型约束提示符。约束在变量声明处可能产生错误：如果类型约束不被满足会产生编译期错误；值约束不被满足则是运行期异常。在函数参数中出现的约束则作为重载决议判断的标准，编译器会按照顺序依次判断是否满足所有模式和约束。

```
const not_zero = funct (x : Int) -> x <> 0;
const not_int = funct (const T : Type) -> T <> Int;

const main = funct () -> do {
    auto x ? not_zero = 42;     // 运行期检查其是否大于零
    auto y :? not_int = 42;     // 编译期错误
};
```

类型约束不仅仅可以用常规的类型谓词，也可以使用类型、类族、模版[5]。

```
const main = funct () -> do {
    scan x : Option[Int];
    match (x)
        :? Int -> print "Just an integer"
        :? Option -> do {
            print "Could be null";
            match (x)
                Some y -> print y;
                Null -> print "Nothing";
        }
        :? Printable -> print "We can print this";
};
```

此时匹配的逻辑是：

- 若是一个类型，则判断待匹配实体的类型是否为该类型。
- 若是一个类族，则判断待匹配实体的类型是否属于该类族。
- 若是一个模版，则判断待匹配实体的类型是否从这个模版实例化得到。

注意区分类型约束和显式类型声明的区别：前者使用 :? 提示符，后者使用 : 提示符；前者会对待匹配实体进行检查，当且仅当待匹配实体满足给定的类型约束时才会完成匹配，后者则是在匹配成功后，为引入的新名字声明类型（并尝试进行隐式类型转换）。前者可能会匹配失败，但除非出现在变量定义语句中，否则不会产生错误；后者可能会出现隐式类型转换失败，错误一定会在编译期产生。

此外值得说明的是，类型约束中不能出现标签类型，这是因为它是一个运行期信息；我们需要记住它要通过选择模式声明来进行模式匹配。当然，我们可以用值约束判断它是否是特定标签，但无法由此得到其中的具体数据，随后还是需要使用选择模式声明。

7.3.2 约束合取与析取

约束的合取与析取操作是借助查询结构完成的。我们已经在第六章中见到了查询结构对结构类型对象的筛选功能。这个功能当然也可以延续到模式匹配的约束中。

```
const main = funct () -> do {
    const gt = funct (x : Int) -> funct (y : Int) -> y > x;
```

[5]有关类族和模版的内容我们会在高级知识的部分介绍。

```
    const lt = funct (x : Int) -> funct (y : Int) -> y < x;

    scan x : Int;
    match (x)
        [? gt(0)][? lt(42)] -> "Greater than 0 and less than 42",
        [? lt(-100), ? gt(100)] -> "Too small or too large",
        otherwise -> "Try a smaller interval";
};
```

如果需要析取一个合取式，可以利用嵌套的查询结构；合取析取式时，和正常使用没有差异。

```
import Math;

const main = funct () -> do {
    const gt = funct (x : Int) -> funct (y : Int) -> y > x;
    const lt = funct (x : Int) -> funct (y : Int) -> y < x;
    const sq = funct (x : Int) -> do {
        return x == Math.floor(Math.sqrt(x)) as Int;
    };

    scan x : Int;
    match (x)
        [? gt(42), [? sq][? lt(42)]] -> "Greater than 42 or is perfect square less than 42",
        [? sq][? lt(42), ? gt(42)] -> "Perfect square not equal to 42",
        otherwise -> "Other matchs";
};
```

此外，回忆函数之间可以进行运算，`f @ g` 相当于 `funct (...args) -> f(...args) @ g(...args)`。因此，我们可以用 `and` 和 `or` 来连接多个约束。

```
const not_zero = funct (x : Int) -> x <> 0;
const is_even = funct (x : Int) -> x `mod` 2 == 0;

const main = funct () -> do {
    auto x ? not_zero and is_even = 42;
};
```

7.3.3 动态约束类型

从约束出发，我们可以设计一些拥有动态约束的类型。可以在结构类型声明或其构造函数中设置值约束。

```
const not_zero = funct (x : Int) -> x <> 0;
const Point = struct (
    x : Real ? not_zero,
    y : Real ? not_zero
);

const main = funct () -> do {
    auto p : Point = (1.0, 0.0);    // 运行错误，约束未满足
};
```

第二部分

高级知识

第八章 名字空间与模块

内容提要

- 名字空间
- 全修饰名字
- 名字查找
- 模块
- 子模块
- import 语句
- 可访问属性
- 模块别名
- 部分模块

本节让我们讨论 AutoScript 中的模块机制，并细致地讲解*名字空间*的概念。它们决定了程序的构成、名字的可见性与可访问性。

8.1 名字空间

8.1.1 修饰名字

回忆我们前面介绍过的*修饰名字*是包含*域连接符* `::` 的名字。实际上，所有合法的 AutoScript 名字都能表示成以域连接符 `::` 开头，由域连接符分隔的多个不以数字和符号（`$` 除外）开头的非空字符串组成的，类似这样的名字称为*全修饰名字*。修饰名字中任一段以 `::` 分隔的部分都称为*分段名字*，全修饰名字中去掉最后的一或多个分段名字的部分被称为名字的*修饰前缀*。

举例来说，`::foo123::bar` 和 `::_123_` 都是合法的全修饰名字，但 `::1abc` 和 `::a;b` 不是；`abc` 和 `a_123` 都是分段名字，而 `a::b` 不是；`::abc` 是 `::abc::def` 的修饰前缀。

在所有全修饰名字中，有一些被保留作为编译器使用的名字，它们是以 `$` 作为任何分段名字的开头的名字，如 `::abc::$a`，以及在任何地方包含连续两个下划线的名字，如 `::abc::a__b`。编译器不会对这些名字进行报错，但非常不建议使用这些名字，很有可能会让编译过程出现意想不到的问题。

我们很少会使用全修饰名字，因为它冗长且不必要。参考下面的例子：

```
1 const add = func (x : Int, y : Int) -> x + y;
2
3 const main = func () -> do {
4     print ::add(1, 2);       // 使用了全修饰名字 ::add
5 };
```

这里，全修饰名字 `::add` 等同于我们在上面定义的 `add`。需要注意，虽然使用时可以使用全修饰名字，但是变量声明处不允许全修饰。

除了全修饰以外，我们也可以使用部分修饰名字。部分修饰名字可以用在变量声明处[1]。

[1]注意，这绝非提倡的写法；实际上，由于编译器也会像 `$` 和 `__` 一样自动生成含有 `::` 的名字，擅自定义一个部分修饰名字是有很大风险的；不过和前两者不同的是，有关 `::` 的生成规则有迹可循，编译器会严格按照作用域的结构生成（我们后面会提到），因此有办法避免和编译器冲突。

```
const good::add = funct (x : Int, y : Int) -> x + y;

const main = funct () -> do {
    print ::good::add(1, 2);    // 使用了全修饰名字 ::good::add
};
```

从这一角度来看，甚至可以将 :: 看作是名字中合法的字符序列，可惜事非如此：比如 :::: 并不是一个合法的名字；更重要的是，由 :: 定义出的分段名字和修饰前缀有它们的特殊含义。

8.1.2 名字空间和作用域

AutoScript 中，**名字空间(Namespace)**是全修饰名字的一个前缀。比如 ::abc::def::ghi 中，::abc 和 ::abc::def 都是一个名字空间，而 ::abc::def::ghi 不是。特别地，最顶层的名字空间称为*全局空间*，所有在文件中直接定义的名字[2]都属于这个空间。

名字空间代表了一个作用域，这可能来自于函数、块语句、结构等。

```
static Value = 42;              // 全局空间中的一个静态存储期变量
const Point = struct (          // Point 是全局空间的一个变量
    x : Real,                   // x 和 y 并非在全局空间中，而是在名字空间 ::Point 中
    y : Real                    // 我们此前已经知道，它们实际上是 ::Point::x 和 ::Point::y
);
const main = funct () -> ();    // 我们熟悉的 main 函数也是在全局空间中
```

名字空间通常是和作用域同时出现的，因此出现作用域的地方一定会出现新的名字空间。

```
const main = funct () -> do @name(body) {
    auto x = 42;
    @name(inner) {
        auto y = 24;
        print nameof(y);    // 输出 ::main::body::inner::y
    }
    print nameof(x)         // 输出 ::main::body::x
};
```

这里，我们利用内置函数 nameof 来得到一个名字的全修饰名字。可以看到，函数作用域、do 表达式的作用域和内部作用域都为变量添加了分段名字。以 x 为例，它全修饰名字中的 main、body 正好反应了其所在作用域从外到内的顺序。

当然，很多情况下，我们不会为作用域命名。此时编译器会对赋予这个作用域唯一的名字。

```
const main = funct () -> do {
    auto x = 42;
    print nameof(x);        // 可能输出 ::main::unnamed0::x;
};
```

由于我们无法提前推测编译器的取名，因此我们无法访问未具名的作用域中的变量。

[2]也就是不在任何函数、结构等内部定义的非修饰名字

8.1.3 名字查找

我们已经知道所有的名字都有一个和作用域有关的全修饰名字，那么在使用未修饰或部分修饰名字时，编译器是如何知晓我们实际指向的名字的呢？这就是**名字查找（Name Lookup）**规则发挥作用的地方。当编译器看到任一个名字时，它都会按照以下步骤来查找：

1. 如果这是一个全修饰名字，查找终止，这就是编译器想要的结果。
2. 如果这是一个部分修饰名字，则查找其第一个分段名字，随后将结果和后面的分段拼接得到最终结果。
3. 如果出现在点调用表达式中 . 的右侧，则尝试从左侧表达式的*解引用类型*的*实例类型*的作用域中查找这个名字。
4. 从当前作用域中查找这个名字。
5. 从当前作用域所在的作用域（从内到外）查找这个名字。

这实际上也解释了名字*覆盖*产生的原因。下面是一系列例子：

```
::x         // 寻找全局空间中的 x
m::x        // 寻找 m 后在 m 空间中寻找 x
f.x         // 寻找 instof(derefof(f)) 空间中的 x
```

如果编译器找不到任何满足要求的名字，则会产生报错：名字未定义。

对于函数（无论其类别），AutoScript 有一个特别的名字查找规则：**参数依赖名字查找（Argument Dependent Lookup, ADL）**[3]。编译器会首先尝试在两个参数所在的名字空间中寻找当前运算符的定义，随后再寻找当前作用域和其所在的作用域（从内到外）。

```
const main = funct () -> do {
    const WrappedInt = Int;
    const + = funct (x : WrappedInt) -> funct (y : WrappedInt) -> (x as Int) + (y as Int);
    @name(inner) {
        const + = funct (x : WrappedInt) -> funct (y : WrappedInt) -> 42;
        print (0 as WrappedInt) + (1 as WrappedInt);        // 输出 3
    }
};
```

从这个例子出发，我们可以得出“变量覆盖对函数不总成立”的结论。确实如此，不过这不是它的主要用处。下一节中我们会详细提到 ADL 的重要用处。

8.2 模块

8.2.1 模块的声明

AutoScript 并没有为模块设计特殊的机制，它完全复用了对象表达式。我们可以通过构建具名的对象表达式来声明一个模块。注意需要使用 `@module` 属性让编译器以模块的方式处理其中的元素。

```
@module
const MyModule = object {};      // 一个空模块，名字是 MyModule
```

因此，和我们在 object 表达式章节中学到的一样，所有结构作用域中自动存储期的变量都会被视作模块的一个可见的成员。此外，其它存储期的变量也总是可见的。

[3]这个特性来源于 C++ 的 ADL；虽然在通常使用中，C++ 的 ADL 并不必要（主要用在运算符重载上），但 AutoScript 中的 ADL 显然是语言运转的核心特性之一：这是因为所有的调用表达式，点运算符后面的名字都需要通过 ADL 来确定其名字空间。

```
@module
const MyModule = object {
    auto x = 42;
    {
        auto y = 10;            // 不可见，因为 y 不构成当前对象表达式的内存位置
    }
    const z = 0;                // 可见，且拥有常量存储期
};
```

如果要导入其它的模块，需要用到 import 关键字。编译器会在编译选项给定的源文件中寻找同名的模块，从而在随后的访问时能够据此找到具体的名字。

语法 8.1 (import 语句)

1. `import [module-id];`
2. `import [module-id] including ([id-list]);`
3. `import [module-id] excluding ([id-list]);`
4. `import [dot-qualified-id-list];`

1. 导入一个模块，使这个模块名可见。这里的 `[module-id]` 是一个由多个点分隔的名字组成的表达式，如 `a.b.c.d`。
2. 导入一个模块，并指定特定的模块成员可见。
3. 导入一个模块，并指定特定的模块成员外所有成员可见。
4. 导入一个模块中的一到多个成员。

下面是一个在同一个文件中定义并引用模块的例子：

```
@module
const ModuleSpace = object {
    @module
    const MyModule = object {   // 相当于一个子模块，嵌套在外部模块 ModuleSpace 中
        auto x = 42;
    };
};

import ModuleSpace.MyModule;    // 读取同文件中的模块，将 MyModule 在当前作用域中可见

const main = funct () -> do {
    print MyModule.x;           // 输出 42
};
```

不过，如果我们想要将上面的代码复现到不同文件的模块上就会遇到问题：

```
// my_module.as
@module
const ModuleSpace = object {
    @module
    const MyModule = object {
        auto x = 42;
```

```
    };
};

// main.as
import ModuleSpace.MyModule;

const main = funct () -> do {
    print MyModule.x;           // 编译错误，MyModule 不可访问
};
```

这也就引出了 AutoScript 中名字的可访问性。

8.2.2 可访问属性

我们可以通过 `import` 语句来让某个名字可见，但是当前作用域或许没有足够权限访问这个名字。这是因为 AutoScript 中所有名字都有**可访问性（Accessibility）**的概念。下面是所有的五种可访问性：

- 导出访问性：用属性 `@export` 表示。该名字会被导出当前模块，因此在任何地方都可访问。
- 内部访问性：用属性 `@internal` 表示。该名字只在当前模块中可访问。
- 局部访问性：用属性 `@local` 表示，该名字只在当前作用域中可访问。
- 私有访问性：用属性 `@private` 表示，该名字无法被外界访问。

可访问属性可以在任何变量声明的地方出现。如果没有对变量使用可访问属性，则它会继承其所在名字空间的访问修饰符。特别地，全局模块中所有名字默认拥有局部访问性，而 `do` 表达式和 `monad` 表达式的语句块中所有名字默认拥有局部访问性，且其中自动存储期的变量最多只能拥有局部访问性。最后，名字空间中任何名字的访问性不能超过名字空间本身的访问性。下面是一些例子：

```
@export
const Point = struct (x : Real, y : Real);  // Point 以及 x 和 y 都拥有导出访问性
const main = funct () -> do {   // main 拥有局部访问性
    auto x = 42;        // x 拥有局部访问性
    @export
    auto y = 24;        // 编译错误：do 表达式中的自动存储期变量最多只能拥有局部访问性。
    const z = 0;        // z 拥有局部访问性
};
```

通常情况下，模块的访问性不需要改变（即局部访问性），因为希望其能在全局作用域中可访问。不过如果需要隐藏这个模块，可以主动使用私有访问性。模块中的普通变量则根据需要声明为 `@export`（希望外部能直接使用）、`@extend`（希望相关类型能直接使用）、`@internal`（希望模块内能直接使用，包括不同文件中的同名部分模块，我们很快会提到）、`@local`（希望同作用域能直接使用）、`@private`（不希望能在任何地方被使用）。这里面，`@private` 是非常严格的访问限制，可以认为这个名字经声明之后就不能被使用了：

```
@module
const MyModule = object {
    @private
    auto secret = 42;
};

const main = funct () -> do {
    import MyModule.secret;     // 编译错误，MyModule.secret 不可见
    import MyModule;
    print (decay MyModule)[0];  // 输出 42
};
```

可以看到，我们依然可以通过取巧的方式拿到私有访问性的名字。不过这不是推荐的使用方式。

8.2.3 模块别名

我们可以利用 alias 关键字声明模块的别名。

语法 8.2 (模块别名)

1. `alias [id] = import [module-id];`
2. `alias [id] = import [dot-qualified-id];`

♣

它本质上就是在导入模块后用别名语句设置别名。

```
alias m = Math.min;

const main = funct () -> do {
    print m(1, 2);      // 输出 1
};
```

8.2.4 部分模块

到现在为止我们接触到的模块特性有一个巨大的缺陷：所有的子模块必须定义在同一个文件中，这是因为它们必须同属一个 object 表达式。这和大多数语言中模块的特点不同。为了支持将子模块定义在不同文件，AutoScript 引入了**部分对象（Partial Object）**的概念，这需要用到 @partial 属性。

```
// a.as
@partial @export
const MyModule = object {
    const SubModuleA = object {
        auto x = 42;
    };
};
// b.as
@partial @export
const MyModule = object {
    const SubModuleB = object {
```

```
        auto x = 24;
    };
};
// main.as
import MyModule;

const main = funct () -> do {
    print MyModule.SubModuleA.x;    // 输出 42
    print MyModule.SubModuleB.x;    // 输出 24
};
```

注意，每个部分对象的声明处都应该添加 `@partial` 属性。

部分模块的本质是，编译器为每个模块的名字上添加了文件和行列数信息，这样每次定义的模块都是独一无二的。在 `import` 相应模块时，会将所有名字为特定前缀的模块作为查找目标。

```
// a.as
@partial @export
const MyModule = object {
    auto x = 42;
};

// 编译器会生成下面的代码
@export
const MyModule__a_dot_as = object {
    auto x = 42;
};
```

第九章 异常处理与单子

内容提要

- 异常
- catch 子句
- throw 语句
- 异常类型推断
- 异常类型声明
- exceptof 函数
- errorof 函数
- 单子
- monad 表达式
- 单子异常
- value 和 error 标注
- 修饰性单子

9.1 异常处理

9.1.1 异常的抛出和捕获

AutoScript 中的异常处理是一个用于处理程序错误状态的机制。当程序遇到一个意料之外的状况时，可以通过 throw 语句抛出一个异常。这个异常会顺着函数的调用链一层层回溯，直到跳出 main 函数后终止程序。

```
const main = funct () -> do {
    const f = funct () -> throw 42;
    f();                    // f 抛出一个异常 42，随后其再次被 main 函数抛出，程序终止
};
```

如果不希望异常继续回溯，可以在可能产生异常[1]的表达式中使用 catch 子句。

语法 9.1 (catch 子句)

1. `catch ([id]) [funct-expr-list]`
2. `catch [funct-expr-list]`
3. `catch`

♣

可以看到，catch 子句和 match 的语法高度相似。上面的第一种相当于为捕获的异常类型声明了一个变量，在捕获时会优先将异常对象初始化到这个变量上，随后再进行模式匹配；第二种则忽略了异常对象的整体值；第三种直接忽略所有捕获的异常。下面先展示一个最基本的捕获示例：

```
const throwable = funct () -> do {
    scan x : Int;
    if (x == 0) {
        throw 42;
    }
    throw "abc";
};

const main = funct () -> do {
    throwable() catch
        Int i -> print "Zero",
```

[1]即异常类型为 Void 的表达式。从设计上来说，AutoScript 的所有表达式均有值类型和异常类型，只不过它们可能分别甚至同时为空类型。

```
        otherwise -> print "Not zero";
};
```

和 match 表达式一样地，catch 语句要求模式匹配中必须出现一个 otherwise 模式，这是为了保证待匹配值一定会落入一种情况中。

一些情况下，我们可能希望将捕获的异常再次抛出。此时可以用上面第一种捕获语法并再次抛出，或使用上下文关键字 again。

```
const ErrorType = data (FatalError | NormalError);

const f = funct () -> do {
    scan x : Int;
    if (x < 0) {
        throw FatalError;
    }
    throw NormalError;
};

const main = funct () -> do {
    f() catch
        NormalError -> print "It's fine";
        FatalError -> throw again;          // 再次抛出异常
};
```

throw 虽然是一个返回语句，但它和 return 语句最大的不同在于它可以“穿透”途径的语句块。如果我们不为返回了异常对象的语句设置 catch 子句，它会再次抛出这个异常。我们可以认为编译器会为所有异常类型为非空的表达式处生成下面的代码：

```
const f = funct () -> do {
    throw 42;
};

const main = funct () -> do {
    f();                                // 这个表达式的异常类型非空
    f() catch otherwise -> ();          // 手动捕获后，编译器就不会自动生成代码了

    // 编译器会生成下面的代码
    f() catch otherwise -> throw again; // 如果没有异常处理则再次抛出
};
```

值得一提的是，catch 虽然是一个子句关键字，但它可以通过符号 !! 表示，它本身也可以组成一个表达式。

```
const f : Int -> Int;

const main = funct () -> do {
    print f() !! otherwise -> 42;       // 如果 f() 抛出异常，则输出 42
};
```

下一节我们会详细讨论和异常相关的类型信息。

9.1.2 异常类型推断

throw 语句中，throw [expr] 的部分是一个表达式，其本身类型为 Void，并拥有异常类型 typeof([expr])。编译器会通过这个类型来推导其实际“影响”的类型：

- 如果 throw 语句成功抛出当前最内侧的 do 或 monad 表达式，则其作为这个表达式的异常类型参考 [2]。
- 如果 throw 语句被某个语句中的 catch 子句捕获，则会根据模式匹配决定 catch 子句最终产生的类型 [3]。

作为异常类型参考的 throw 语句会类似于声明了 choice 的 do 表达式中的 return 语句，编译器会结合所有的 throw 语句并将其选择类型作为异常类型。

```
const f = funct () -> do {
    scan x : Int;
    if (x < 0) {
        throw 42;        // 异常类型为 Int
    }
    else if (x > 0) {
        throw "abc";     // 异常类型为 String
    }
    else {
        throw True;      // 异常类型为 Bool
    }
};                       // do 表达式的异常类型为 Int | String | Bool
```

除了参考 throw 语句的异常类型，任何表达式的异常类型都会参与到编译器的判断中。比如内置的除法运算会产生 ZeroDivisor 异常，因此包含除法的表达式拥有异常类型 ZeroDivisor。

```
const f = funct () -> do {
    scan x : Int;
    scan y : Int;
    print x / y;         // 异常类型为 ZeroDivisor
};                       // 异常类型为 ZeroDivisor
```

当然，如果在可能产生异常的语句处使用 catch 子句，且不再次抛出，则这个语句不参与编译器的异常类型推断（因为异常类型被 catch 子句“洗白”了）。

```
const f = funct () -> do {
    scan x : Int;
    scan y : Int;
    print x / y catch;   // 异常类型为 Void
};                       // 异常类型为 Void
```

此前我们给出的例子都是 do 表达式中出现的异常类型判断。但在 monad 语句中其实会出现相同的效果，这里编译器并没有将表达式包装为声明的单子类型：

[2]和 return 语句不同地，一个 throw 语句无法决定最终的异常类型，我们下面很快就会提到。

[3]经过 catch 子句最终得到的表达式类型是它的值类型。某种角度来说，catch 可以“洗白”有问题的异常语句。

```
const main = funct () -> do {
    print monad Option {
        throw 24;                // 直接抛出 24 而不是 Some 24
    } catch otherwise -> 42;
};
```

本质的原因是，单子本身就带有了异常类型的语义，因此不需要再次为异常对象包装。我们会在后面的小节中详细介绍。

9.1.3 函数的异常类型声明

异常类型声明是可选的，且出现在函数类型的最后，用关键字 except 表示。

语法 9.2 (异常类型声明)

1. except [type-expr]
2. except

♣

第一种是显示给出所有可能的抛出类型，第二种则仅声明函数会产生异常，并委托编译器通过函数定义来推断具体的异常类型。后者是更加常用的形式。函数异常声明中出现的类型应该是函数定义中异常类型的超集[4]。

```
const f : Int -> Void except Int | String;
const f = x -> throw 42;                    // 没问题
```

为了得到表达式的异常类型，我们可以使用 exceptof 关键字。这是一个静态类型函数。如果想要动态获得异常类型，可以使用 errorof 关键字。

```
const f = funct (b : Bool) -> do {
    if (b) {
        throw 42;
    }
    throw "abc";
};

const main = funct () -> do {
    print exceptof(funct(True));        // 输出 Int | String
    print errorof(funct(True));         // 输出 Int
};
```

其中的原理和 tagof 类似，本质上就是取选择类型对象的标签。

[4]这实际上就是提前声明的兼容原则：定义处应是声明处的子类型。对于返回值拥有异常类型的函数，更少的异常可能是更多异常可能的子类型（好好思考这一点）。

9.2 单子

9.2.1 单子的引入

AutoScript 中，**单子（Monad）**是对拥有特定特征的类型的称呼。这类类型一定是一个选择类型（包括 ADT 类型），且其中包含一个*值类型*和一系列*空类型*；这里空类型也被称为*异常类型*[5]。首先让我们看看下面的一些例子：

```
const make_option = funct (x : T) -> Some x;
const make_list = funct (x : T) -> Vector[x];

const main = func () -> do {
    auto i = 42;
    auto r = ref i;                          // 创建对象
    match (r)
        ref x -> print x;                    // 取出内部的对象
        otherwise -> print "Null reference";// 空引用的情形

    auto opt = make_option(42);              // 创建对象
    match (opt)
        Some x -> print x,                   // 取出内部的对象
        Null -> print "Null object";         // 空对象的情形

    auto list = make_list(42);               // 创建对象
    match (list)
        x -> print x;                        // 取出内部的对象（情形 1）
        ...xs -> print "Multiple elements"; // 取出内部的对象（情形 2）
        () -> print "Empty list";            // 空对象的情形
};
```

在上面的例子中，我们对于引用类型、`Option` 和 `List` 展示了一些常见的使用代码。注释中已经给出了这些行为的总结：

- 包装：这些类型需要一个包装函数。
- 对包装对象执行操作：在对象非空的情况下，需要直接对其中的值进行操作。很多时候这会返回一个包装过的同类型对象，此时这个操作称为绑定：比如对一个包装过的对象 `x` 依次执行 `f`、`g`，其中的每个函数都会解包对象并返回一个变换后对象的包装。理想情况下，我们可以借助这个操作简便地将 `x` 依次传入 `f` 和 `g` 而无需手动解包两次。

AutoScript 中，利用 `@monad` 属性为类型设定上面列出的两个操作的类型就是单子类型[6]，但对于选择类型来说，其中第一个选择会被编译器当作值类型，而随后的所有标签都会被当作不同的异常类型；非选择类型会把其本身的类型当作值类型，而异常类型为 `Void`。下面是一些例子：

[5]注意到这里称谓和此前介绍的表达式类型雷同，这是因为所有 AutoScript 单子都可以兼容到表达式异常这个机制中，我们很快会介绍到这一点。

[6]AutoScript 从未要求单子类型一定是一个选择类型，甚至 ADT 类型。

```
@monad(make_option, option_bind)    // 为 monad 属性提供包装和绑定操作
const Option = data (Some T | Null);

const make_option = funct (x : T) -> Some x;

const option_bind = funct (opt : Option[T])
    -> funct (f : T -> Option[T]) -> match (opt)
        Some x -> f(x),
        Null -> Null;

@monad(default, default)
const SpecialMonad = data (A Int);  // 异常类型为 Void，相当于声明了 data (A Int | Void)

@monad
const AlsoMonad = Int;              // 值类型为 AlsoMonad，异常类型为 Void
```

从这个角度来看，AutoScript 中的所有类型都可以当作单子。上面在 `@monad` 属性中出现的是指定的单子包装函数和单子绑定函数，其签名是确定的因此不需要提前声明。如果将其中的参数设定为 `default`（或当两个均为默认时可以直接省略 `@monad` 属性的参数），则会通过下面的默认方式创建单子包装和单子绑定函数：

```
@monad
const SpecialMonad = data (A Int);
@monad
const TrivialMonad = Int;

// 编译器会生成下面的代码
const make_SpecialMonad = funct (x : Int) -> A x;
const bind_SpecialMonad = funct (x : SpecialMonad)
    -> funct (f : Int -> SpecialMonad) -> match (x)
        Int y -> f(y),
        otherwise -> throw Unreachable;

const make_TrivialMonad = funct (x : Int) -> x;
const bind_TrivialMonad = funct (x : TrivialMonad)
    -> funct (f : Int -> TrivialMonad) -> match(x)
        Int y -> f(y),
        otherwise -> throw Unreachable;
```

后面的小节中我们将介绍如何使用单子类型。

9.2.2 monad 表达式

AutoScript 为了让上一节中提到的语法糖生效，特地设计了 `monad` 表达式。在 `monad` 表达式中，许多表达式的含义会发生变化。让我们以 `monad Option` 为例：

```
const main = funct () -> do {
    auto res = monad Option {   // 声明当前语句块为 Option 的单子块
        scan x_input : Int;     // 读取一个 Int 类型的参数
        auto x = x_input;       // 这里会将 x_input 的值自动装包为 Some x_input
        if (x <= 0) {           // if 中的判断表达式会对 x 进行拆包，如果是 Null 则跳过
            return 42;          // 返回的实际是 Option 42
        }
        x <- x + 1;             // 对 x 进行拆包且不为 Null 才进行操作。
        return x;               // 若 x 为 Null 返回 Null，此外返回的实际是 Some x
    };
    print res;
};
```

上面，无论是在 `x <= 0` 的判断处、`x + 1` 的运算处，还是返回语句等位置，编译器都为这些操作添加了额外的拆包和装包操作。可以总结为：除了显式给出表达式类型的情形，编译器会为所有表达式中的变量尝试装包和绑定。我们实际上可以写出下面的等价代码：

```
const main = funct () -> do {
    auto res = do {
        scan x_input : Int;
        auto x = make_option(x_input);
        if (option_bind(x) (<= 0)) {
            return make_option(42);
        }
        x <- make_option(option_bind(x) (+ 1));
        return make_option(x);
    };
    print res;
};
```

`monad` 表达式中虽然允许出现多种单子类型，但其返回类型必须是唯一可以确定的，这点和 `do` 表达式没有不同。

```
const main = funct () -> do {
    auto res = monad Option {
        scan b : Bool;
        if (b) {
            return 42;      // 编译器推断 monad 表达式返回类型为 Option[Int]
        }
        return "abc";       // 编译错误，表达式类型前后判断矛盾
    };
};
```

有时，我们并不需要编译器自动将表达式包装为单子类型，此时可以使用表达式修饰符 `fixed` 来令某一层表达式不受单子包装影响。

```
const main = funct () -> do {
    auto res = monad Option {
        auto x = fixed 0;       // 0 不会被自动装包为 Some 0
        if (x <= 0) {           // 若 x 不是单子类型，这里不会自动拆包
            x <- fixed 42;      // 由于左侧是 Int 类型，右侧不需要包装为 Some 42
        }
        return x;
    };
};
```

fixed 只能作用于一次表达式求值。如果需要嵌套的多个表达式都不受单子装箱影响，可以直接使用 do 表达式。

```
const main = funct () -> do {
    auto res = monad Option {
        scan x : Int;
        if (fixed x <= 0) {
            x <- fixed do { return x * x + 1; };    // 等效于 fixed (fixed x * x) + 1
        }
        return fixed x;                             // 这里会返回一个 Int 而非 Option[Int]
    };
};
```

从上面的例子更加可以看出 monad 表达式只是 do 表达式建立在 @monad 类型上的语法糖；我们完全可以让 monad 表达式中不出现任何单子类型。

嵌套的多个 monad 和 do 语句相互独立，不会受到单子声明影响，这也是为什么上面的代码中 do 的语句块屏蔽了外层语句块的装箱操作。不过，do 表达式本身是一个表达式，因此它的返回值会被装箱为 Option 类型，因此上例中我在最外层使用了 fixed 修饰符。

9.2.3 单子异常

由于单子类型自带“异常类型”，AutoScript 提供了将单子对象作为异常类型抛出的方式。我们需要在返回语句中使用 value 或 error 关键字。

```
const throw_option = funct () -> monad Option {
    scan b : Bool;
    if (b) {
        return value 42;
    }
    return error Null;
};

const main = funct () -> do {
    print typeof(throw_option);     // 输出 () -> Int
    print exceptof(throw_option()); // 输出 Null
};
```

可以看到，这两个关键字为函数添加了一个异常类型，也即其空类型。value 并不意味着编译器会直接将其后的表达式作为函数的返回类型：它依然会当场构建一个 Some 42 对象，随后会根据 (Some 42).value() 得到其中包含的值类型。error 后面的对象也是根绝 Null.error() 推断得到其类型的。

经过这样调整的函数会和使用了抛出语句的函数行为非常相似：编译器会将 return error 语句修改成 throw 语句，因此（正如在上一节中介绍的），所有调用该函数的地方都会添加一个默认捕获并再次抛出的 catch 子句。相应地，我们可以手动添加 catch 子句来捕获这些函数的异常。

```
const f = funct () -> monad Option {
    scan b : Bool;
    if (b) {
        return value 24;
    }
    return error Null;
};

const main = funct () -> do {
    f() catch
        Some i -> print "value = $(i)",
        Null -> print "Nothing is there.";
};
```

9.2.4 修饰性单子（实验性）

在前面的小节中，我们在介绍单子的性质时曾经将引用类型作为例子。实际上，所有类型都可以看作单子——从修饰符的角度来看。我们可以轻松地定义出它们的包装和绑定函数，以引用类型为例：

```
const make_ref = funct $x -> ref x;
const bind_ref = funct ($r : ref T) -> funct ($f : T -> ref T) -> match (r)
    ref x -> ref f(x),
    otherwise -> r;
```

这里出现的 $ 是一个别名声明，其中的参数会以内联的方式在编译期展开，因此作为参数的 x 不会在传递时被复制出来。具体内容详见别名和宏的章节。

第十章 模版与元编程

内容提要

- 推导类型
- 函数模版
- 模版实例化
- 结构模版
- 抽象数据类型模版
- 对象模版

10.1 函数模版

10.1.1 推导类型

在这个章节以前，我们遇到的所有显式类型声明都是类似于 `x : T` 的形式，其中 `T` 是一个已知的类型，编译器会将初始化表达式的值隐式转换为 `T` 后绑定到 `x` 上。如果我们希望编译器严格按照初始化表达式的类型声明变量，则可以将显式类型声明省略掉。不过有时候我们希望能够得到这个类型，此时就需要额外使用一次 `typeof`：

```
const main = func () -> do {
    auto x = 42;
    const T = typeof(x);
};
```

AutoScript 中为上面这样的操作设置了一个语法糖：我们可以用关键字 `infer` 在变量声明时同时声明一个 **推导类型（Inferred Type）**：

```
const main = func () -> do {
    auto x : infer T = 42;
};
```

推导类型也可以出现在提前声明的地方。

```
const main = func () -> do {
    auto x : infer T;       // 声明了一个拥有推导类型 T 的变量 x，这样的实体也被称为对象模版
    print(typeof(x));       // 输出 Unresolved，这是定义在 Prelude 模块中的，表示未确定类型
    auto x = 42;            // x 的初始化
};
```

在函数参数列表中，我们可以声明一个对象模版。编译器会根据函数调用处的参数类型来推断推导类型的实际类型[1]。

```
const add = funct (x : infer T, y : T) -> x + y;

const main = funct () -> do {
    print add(1, 2);        // 输出 3
    print add(1.0, 2.0);    // 输出 3.0
};
```

[1] 注意 `x : T` 和 `x : infer T` 的相同和不同之处：两者都保证 `x` 的类型是 `T`，但前者（在 `T` 已知时）会将初始化表达式隐式转换为 `T`，后者则会通过初始化表达式的类型来确定 `T`。

在上面的例子中，我们只在参数 x 后使用了推导类型的声明，因此编译器只会根据这个参数来推断 T，至于 y 则会尝试将传入的参数隐式转换为类型 T。当然，我们也可以同时为 y 声明推导类型 infer T，此时编译器会强制要求两个参数的类型推断结果完全相同。

```
const add = funct (x : infer T, y : infer T) -> x + y;

const main = funct () -> do {
    print add(1, 2.0);      // 编译错误，推导类型冲突
};
```

诸如上面这些参数中包含推导类型的函数称为**函数模版（functtion Template）**。函数模版中，可以省略推导类型的声明（即省略 infer 关键字），此时编译器会将第一次出现的类型作为推导类型的推断标准。

```
const add = funct (x : T, y : T) -> x + y;
// 上面的函数相当于 funct (x : infer T, y : T) -> x + y
```

更简洁地，我们可以完全省略函数的参数类型声明，此时编译器会将每个这样的参数视作声明了一个推导类型。

```
const add = funct (x, y) -> x + y;
// 上面的函数相当于 funct (x : infer T, y : infer U) -> x + y
```

从函数上推导类型的简化形式可以反推对象模版（也就是最原始形式的推导类型）也可以省略 infer 关键字，甚至推导类型声明[2]：

```
const main = funct () -> do {
    auto x : T;             // 声明了一个对象模版
    auto y;                 // 声明了一个对象模版
};
```

可以看到，函数模版为我们提供了一种轻松的函数声明方式。不过，编译器是怎样妥善处理不同参数造成的差异呢？这就引出了*模版实例化*的概念。

10.1.2 模版实例化

编译器在遇到一个函数模版时，不会对其返回类型进行推断（因为它通常做不到）。在函数模版被调用时，编译器会根据参数类型生成一个模版的**实例化（Instantiation）**。实例化的可访问性与所在作用域和模版本身相同。

```
const add = funct (x, y) -> x + y;

const main = funct () -> do {
    print add(1, 2);        // 编译器会实例化一个函数 add__Int__Int : (Int, Int) -> Int
    print add(2.0, 0.0);    // 编译器会实例化一个函数 add__Real__Real : (Real, Real) -> Real
};
```

因此，所有的模版函数在使用时实际上调用的是编译器生成的实例化函数。

实例化有时会失败，这是因为实例化的参数不一定满足模版中使用的函数（和运算符）的声明，或者函数本身有缺陷。此时会产生编译错误。

[2]细心的读者会想到，由于 auto 本身可以被省略，那么莫非 x 本身就是一个变量声明？在函数中确实是这样的，在其他地方也是如此。如果 x 不曾被声明，那么 x; 这个语法就构成了它的声明。显然这样的写法不应被提倡，但它的有效性是水到渠成的。

```
const f = funct (x) -> do {
    return f(x);            // 不存在实例化时不会报错
};

const main = funct () -> do {
    f(42);                  // 编译错误，无法推断 f 的类型，实例化失败
};
```

这里让我们看向一个特别的情况：约束失败并不是实例化失败，它是模式匹配中常见的情形，编译器会自动跳转到下一个候选继续判断。

```
const f : (infer T) -> T = guard
    x :? Int -> x,
    :? String -> "Hello, world",
    otherwise -> undefined;

const main = funct () -> do {
    print f("abc");         // 输出 Hello, world
};
```

10.1.3 函数模版的本质

如果对一个函数模版使用 typeof 函数，其结果可能有些出乎意料。

```
const add = funct (x, y) -> x + y;

const main = funct () -> do {
    print typeof(add);      // 输出 (Type, Type) -> Unresolved
};
```

可以看到虽然 add 的类型确实是一个函数，但它却是接受两个常量存储期类型，并返回不确定类型 Prelude.Unresolved 的函数。事实上，编译器在看到一个带有 infer 的参数列表时，会将其作如下处理：

```
const add = funct (x, y) -> x + y;

// 编译器会生成下面的代码
const add = funct (const T : Type, const U : Type)
    -> funct (x : infer T, y : infer U) -> x + y;
```

对于这样接收的参数包含常量存储期的函数，在调用时可以跳过一步调用，并通过后续调用中的实际参数的类型推断来得到这些参数的值。这也就是模版实例化的本质。如果依靠 infer 的层级超过了一次调用，则不能用来推断这些常量存储期的参数。

```
const f = funct (const T : Type, const U : Type)
    -> funct (x : infer T) -> funct (y : infer U) -> x + y;

const main = funct () -> do {
    f(2)(3);            // 错误，U 未初始化
```

```
    f[U = Int](2)(3);   // 没问题
};
```

10.2 类型模版

10.2.1 结构模版

在一个结构对象声明中，我们同样可以使用推导类型来标注一个范型成员。

```
const Point = struct (
    x : infer T,
    y : T
);

const main = funct () -> do {
    auto p : Point = (1, 0);    // 类型为 Point [T = Int]
};
```

观察 p 的类型就会发现其形式类似于一个函数调用。确实，结构模版的本质也是一个函数：

```
const Point = struct (x : infer T, y : T);

// 编译器会生成下面的代码
const Point = funct (const T : Type) -> struct (x : infer T, y : T);

const main = funct () -> do {
    auto p1 : Point = (1, 0);
    auto p2 : Point[T = Int] = (1, 0);
    auto p3 : Point(Int) = (1, 0);
};
```

上面的 p2 和 p3 都是不需要模版知识就可以理解的显式类型声明；p1 中 Point 本来并不是一个类型，但借助于 AutoScript 的模版实例化机制，编译器会尝试将等号右侧的内容作为类型推断的依据。

这个性质也可以运用在带有 @ctor 属性的函数上。

```
@ctor
const Pair = struct (x : infer T, y : T);

// 编译器会生成下面的代码
const Pair = funct (const T : Type) -> struct (x : infer T, y : T);
const Pair__ctor = funct (x : infer T, y : T) -> (x, y) as Pair(T);

const main = funct () -> do {
    auto p = Point(1, 0);

    // 编译器会生成下面的代码
```

```
    auto p = Pair__ctor(1, 0);
};
```

10.2.2 抽象数据类型模版

以 Prelude.Option 为例：

```
const Option = data (Some infer T | Null);

// 编译器会生成下面的代码
@ctor
const Some = funct (const T : Type) -> struct (t : infer T);
@ctor @eager
const Null = funct (const T : Type) -> struct ();
const Option = funct (const T : Type) -> Some[T] | Null;
```

ADT 表达式中出现的全是没有出现的标识符，其中 Some 和 Null 都是标签类型，而 T 是一个模版类型。

10.3 类族

10.3.1 类族表达式

本节中我们将介绍一种特殊的函数模版，它的实例化方式比较特殊。首先让我们看一个普通的函数模版：

```
const add = funct (x : T, y : T) -> x + y;
```

这里我们没有对 T 做任何假设，但如果实例化时 T::+ 无定义，上面的函数在实例化时就会失败。这个检查是多次进行的，也即每次遇到一个 x + y 时编译器都需要取搜索是否有函数 T::+ 并做出相应的反应。好消息是，我们有方法将这个结果“缓存”到一个地方。

```
const add = funct (x : T forall class { x : T; y : T; x + y; }, y : T) -> x + y;
```

这里我们用到了 *类族提示符* forall 和类族表达式。由于 forall 是一个阐述关键字，我们可以用符号 :! 来代替它。

语法 10.1 (类族表达式 [class-expr])

```
class [block]
```

♣

简单来说，类族表达式就是一个代码块，不过其中存储了一个类似于缓存的机制：所有没有通过检测的类型都会被标记为 False。当类族表达式中不存在任何推导类型时，它对任何类型都会输出 True 或 False（取决于其中的代码是否合法）。此时的类族没有任何意义。

```
auto SimplyWrong = class {
    x : String;
    y : String;
    x * y;
};                  // 编译器不会做任何事（虽然它会将 () 标记为 False

const main = funct () -> do {
```

```
    print SimplyWrong ();           // 输出 False
    print SimplyWrong (Int, Int);   // 输出 False
};
```

我们通常只对拥有推导类型的类族表达式感兴趣：

```
const Addable = class {
    x : T;
    y : T;
    x + y;
};
```

此时我们可以通过类族提示符将其用在任何函数声明中：

```
const add = funct (x : T forall Addable) -> x + y;
```

对比此前的代码可以看到我们只是将类族表达式声明为变量了而已。

当省略推导类型声明时，我们也可以直接将类族放在类型的位置上（为了简洁，我们使用的是符号 `:!`）.

```
const Printable = class {
    x : T;
    print x;
};
const print_twice = funct (x :! Printable, y :! Printable) -> do {
    print x;
    print y;
};
```

不过要注意上面这个例子对应的是 `funct (x, y) -> do { ... }` 的情形，因此 `x` 和 `y` 完全可以是不同类型的变量。如果要求两者的类型相同，可以先表明其中一个的类型所在的类族：

```
const Addable = class {
    x : T;
    y : T;
    x + y;
};
const add = funct (x : T forall Printable, y : T) -> x + y;
```

除了在函数参数中判断类族，我们也可以用内置运算符 `hasclass` 来判断某个类型满足特定类族。

```
const Addable = class {
    x : T;
    y : T;
    x + y;
};

const main = funct () -> do {
    print typeof(1) hasclass Addable;   // 输出 True
    print String hasclass Addable;      // 输出 False
```

```
};
```

和函数参数中出现的类族一样，这里一经判断编译器就会将它记录下来，此后的判断不需要再检阅类族定义和类型定义来决定该类型是否满足这个类族了。

10.3.2 类族的实现

除了让函数在调用时判断类族是否被实现，我们也可以在任何时候用 implement 语句来声明某个类型实现了类族中的全部要求。

语法 10.2 (implement 语句)

```
implement [class-id] [type-id];
```

♣

```
const Addable = class {
    x : T;
    y : T;
    x + y;
};
const Pair = struct (
    x : Int,
    y : Int
);

implement Addable Pair;      // 错误，Pair 没有实现 + 运算符
```

上面出现的类型不必出现过完整定义：我们只要保证和类族相关的声明在 implement 语句之前出现过，就能让上面的判断通过。

```
const Addable = class {
    x : T;
    y : T;
    add(x, y);
};
const Pair : Type;
const add : (x : Pair, y : Pair) -> Pair;
implement Addable Pair;      // 通过编译
```

虽然 implement 语句可以出现在任何地方，但由于它只在声明所在的作用域生效，因此我们常常将它直接放在模块中。

```
const Addable = class {
    x : T;
    y : T;
    x + y;
};
implement Addable Int;
implement Addable Real;
```

特别地，对于 `@inline` 的模块，其中声明的 `implement` 语句会在其所在作用域中也有效。因此，标准库 `Prelude` 中的类族实现默认在所有文件中都有效。

10.3.3 多参数的类族

类族中可以出现多个模版参数，但从此前的例子可以看到，类族只能用于修饰一个类型，因此我们需要保证它在使用时只剩下一个未应用的参数。

```
const Convertible = class {
    f : From;
    t : mut To;
    t <- f;
};

const f = funct (x forall Convertible[To = T], y : T) -> do {
    auto res = mut y;
    scan b : Bool;
    if (b) {
        res <- x;
    }
    return res;
};
```

注意到上面我们虽然在 `x` 和 `y` 的类型声明中都使用了推导类型 `T`，但它只会通过 `y` 的类型来推断，这是因为类族中的参数不能参与类型推导（尽管它是第一次出现）。相同的道理也应用于其它各种模版。

10.3.4 类族与约束的对比

约束同样可以用在参数列表中，不过它不提供缓存机制；同时作为模式匹配的一部分，它不构成函数的*原型*。与之相比的，类族可以用在函数类型声明中，并作为其类型的一部分。

```
const Addable = class {
    x : T;
    y : T;
    x + y;
};

const succ : (T forall Addable) -> T;
const succ = (x : Int) -> x + 1;
```