FrontWork

WorkScript

—入门就精通

- 语法简单，学习成本低
- 设计严谨，不失专业性
- 精心优化，运行速度快
- 可能是全世界最简单的编程语言

荆佳颉 主编

内部读物

目录

第一章：快速上手

1.1 获取 WorkScript

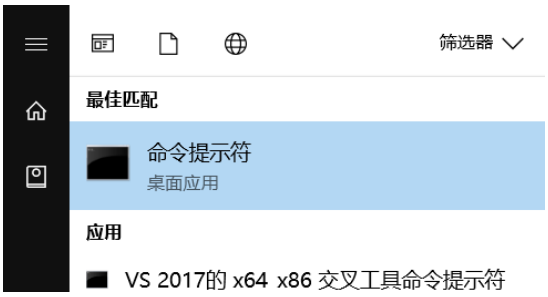
到我写这篇文档的时候，WorkScript 还没有官方网站以及任何官方渠道。WorkScript 解释器的获取只能联系荆佳颉。

1.2 安装与启动 WorkScript

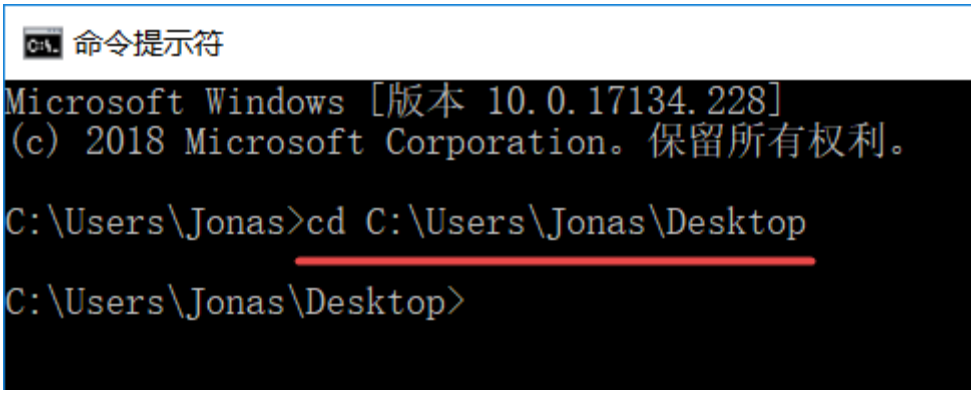
目前的 WorkScript 还没有来得及做图形化编辑界面，只有一个解释器命令行工具，并且没有安装程序。所以，启动 WorkScript 是一件略显麻烦的事情，但同时也是一项非常有意义的工作。

首先获取到 WorkScript 解释器的程序 wsi.exe，放置在任意你想要的位置。作为示例，我将其放置在桌面上。

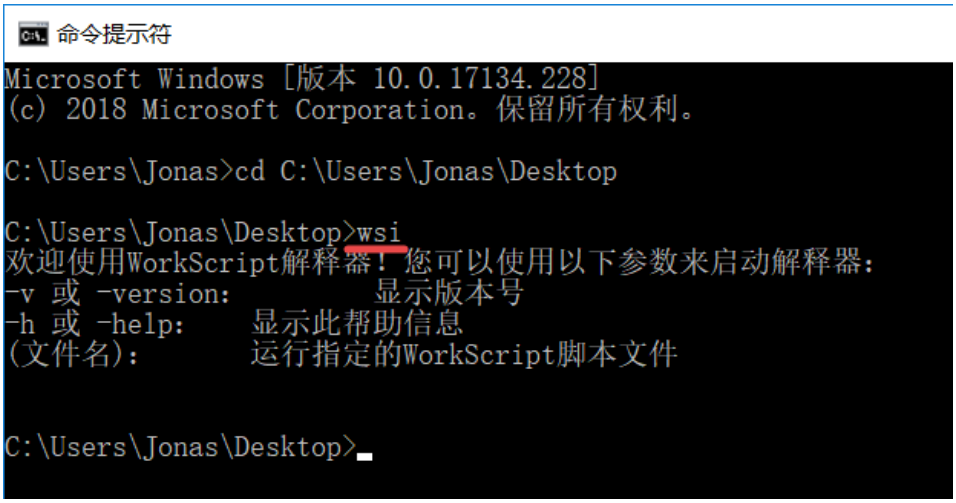
接下来，进入命令行。我们点击 Windows 桌面左下角的开始图标，打开开始菜单界面，直接输入”cmd”即可找到命令行。



进入命令行之后，我们将当前目录切换到 wsi 所在的目录。以上图为例，我们使用 cd 命令进行路径的切换到桌面目录。



目录切换完成之后，我们在当前目录直接输入 wsi（或者 wsi.exe，在 Windows 下允许省略 exe 后缀），即可启动 WorkScript 解释器。



可以看到，不同于其他晦涩难懂的编程语言，WorkScript 解释器的提示语是以中文显示的。这是因为 WorkScript 面向的对象是非计算机专业的人士。WorkScript 语言的设计一直是以简单，易懂为基本的设计原则，无论是在语法

的设计上，还是在逻辑思维上，WorkScript 都尽量贴近没有学过编程的人，在不失专业性的同时，为更加广大的群体提供了与电脑沟通的语言。

1.3 编写第一个程序

几乎所有的编程语言都是以“Hello World”作为开始的。打印一行文字实在是太过简单的一件事了，本章以一个更加有趣的例子作为 WorkScript 学习的开始。

现在让我们来做一件简单的事：求一个数字的绝对值。 想必大家在初中的数学课上都学过绝对值的概念：

大于 0 的数字的绝对值是它自己；

0 的绝对值是 0；

小于 0 的数字的绝对值是它乘以-1；

数学老师会在黑板上用数学中“函数”的概念为我们描述上面的逻辑关系。让我们将自己想象成当年的那个数学老师，你会如何将上面的三句话翻译成数学公式呢？

```
f(x > 0) = x

f(0) = 0

f(x < 0) = -x
```

上面的三行公式想必非常容易理解。那么现在的问题是，我们需要将数学问题转换为计算机语言，让电脑帮我们解决问题。如何把这样的三行公式翻译成计算机编程语言呢？

答案是，不需要任何翻译。因为上面的三行公式，就是合法的 WorkScript 代码。让我们利用上面的三行程序，来计算-5，0 和 20 三个数的绝对值。接着编写如下程序：

```
print(f(-5))

print(f(0))

print(f(20))
```

将上面的 6 行程序保存到 a.ws 文件，然后使用命令“`wsi a.ws`”来运行 a.ws 文件。不出意外，会打印出 5 0 20 三个数字。

第二章：变量和值

2.1 值

在上一章，我们已经和值有过初步的接触了。和所有其他计算机语言一样，目前版本的 WorkScript 提供了以下几种类型的值：

数字，字符串，布尔值，函数

函数作为一种特性非常丰富的值，我们在下一章单独进行讲解。下面我们依次为大家讲解其他几种值及相关的运算。

2.1.1 数字

数字是我们生活中最常见的值了。相信只要学过数学的人，都会知道数字有整数，小数，正数，负数，二进制数，十六进制数等等诸多分类。在 WorkScript 中，所有的数字用常见的十进制表示。如下的数字是合法的：

```
1  100.5  0  -25  -30.6  +1  +14.8
```

而如下形式的数字是非法的：

```
a  1.b  2+  3- 4c  12._5
```

值得一提的是，如果数字和变量之间不加任何符号，则默认为乘号。例如上面的“4c”会认为是“4 * c”。这样的性质非常适合我们编写数学公式，在数值计算上提供了比较大的方便。但是这个特性的书写格式要求较为严格，我们会在下一章四则运算的小节来介绍这个特性。

2.1.2 字符串

字符串是计算机处理文字所使用的。所有的计算机语言都有字符串这个类型的值，但如果你是初次接触计算机语言，也许对字符串会感到陌生。那么下面的这个例子，就是一个标准的字符串：

```
“你好世界”
```

可以看到，字符串其实就是我们日常的说话。在 WorkScript 中，字符串需要使用单引号或者双引号将说话的内容括起来。在说话的内容里，允许有任何中英文，标点符号等字符，但不可以包含作为字符串边界的引号。

例如，下面两种字符串是不合法的：

```
“你听见他说”你真美”了吗”
‘你听见他说‘你真美’了吗’
```

对于这种情况，WorkScript 会认为“你听见他说”是一个字符串，“了吗”是一个字符串。而“你真美”是一个变量。如果需要表示上面的字符串，我们需要用如下方式：

```
‘你听见他说“你真美“了吗’
```

注意我们将两边的双引号换成了单引号，这样就不会引起歧义了。当然，以下形式也是合法的：

```
“你听见他说‘你真美’了吗”
```

2.1.3 布尔值

说到布尔值，如果你是第一次接触计算机语言，可能会感觉到陌生。但其

实它就存在于我们的生活中。例如，你的朋友问你是坐地铁上班的吗？你的回答可能是**“是”**或者**“否”**。老师问你“1+1”等于 2 对吗？你的回答可以是**“对”**或者**“错”**。在计算机的世界，我们将是，对，好等积极的回答统称为**“真值”。**将否，错，不好等反面的回答统称为**“假值”。**

在 WorkScript 中，我们定义了如下四种真值：

```
true  yes  ok  good
```

和如下三种假值：

```
false  no  bad
```

其中，所有的真值都是相等的，所有的假值也都是相等的。也就是说，在计算中，yes=true=ok=good，false=no=bad。所以，使用哪种值不影响程序的运行结果，但应根据语境选择更加合适的词语。

2.2 变量

2.2.1 变量的一般特性

变量是个熟悉的概念了，无论是在数学上，还是在计算机语言中，都存在变量的概念。变量拥有名字，称为变量名。每个变量可以存储一个值，例如数字，布尔值，字符串以及函数等。并且在需要的时候可以从变量中获得存储的值。WorkScript 中变量的名字必须符合如下规定：

必须由字母，数字，下划线，中文或其他语言的文字组成。并且第一个字符不能是数字。

也就是说，如下的变量名是合法的：

```
a  _b  a1  b2_  员工  成绩 1
```

而下列变量名是非法的：

```
01  1a  %b  ~员工  <成绩>
```

计算机中的变量和数学上的变量是不一样的。数学中，每个变量的值是固定的，即使我们不知道 x 的值，也可能通过公式推导，消元来算出 x 的值。这里面隐含的意义就是 x 的值从一开始就是固定的，不会随着我们的计算而改变。所以，与其叫作变量，不如称数学中的变量为“常量”更加合适。

计算机语言中的变量是真正可变的。我们可以对变量进行赋值：

```
X = 1

X = “hello world”

X = true
```

需要注意的是，在许多被称作“静态类型语言”的编程语言中，一个变量和其存储的值的类型是绑定的。WorkScript 属于“动态类型语言”，变量和值的类型不要求绑定。所以，上面依次对 x 赋数字，接着赋字符串，最后赋布尔值，都是合法的语句。

但是，熟悉 JavaScript 和 Python 的读者一定会第一时间站出来抱怨道：“动态类型虽然带来了很大的灵活性，但是同样也很容易造成错误!”。的确，由很多意想不到的错误都是因为编程语言放任随意的赋值行为而不加管束所导致的。为了解决这个问题，WorkScript 设计了类型约束，可以在合适的地方将变量与值的类型进行绑定，来减少程序出错的可能性。

2.2.2 变量的类型约束 [尚未实现]

对于一个普通的变量 x，我们可以为它赋任意类型的值。但是也许我们只将 x 作为一个数字并对其进行加减乘除运算，这时如果不小心为 x 赋了一个字符串，就会出现意想不到的错误。要特别注意的是，字符串也同样拥有加法运算！例如：

```
y = x + 1
```

这个时候 x 的值如果是 2，那么 y 的值将是 3。而如果此时 x 的值为”hello”，那么 y 会得到”hello1”的结果。更有甚者，将 x 赋值为”2”，那么 y 会出乎意料的得到”21”而不是 3。

也许上面的例子比较简单，但是当程序复杂后，这样的错误会一个接一个顺着程序传递下去，直到很远的地方才会被发现。而发现错误后想要追根溯源

找到这里，会浪费大量的时间。下面我们用类型约束为 x 绑定数字类型的值：

```
y = x:number + 1
```

这个时候，如果 x 存储的值不是数字类型，那么会直接引起一个运行时异常（关于异常，将在后面的章节中讨论），从而令我们可以在程序的调试过程中直接发现这个错误，避免繁琐的追查过程。

当然，不论变量处于任何地方，都可以进行类型约束。例如上面例子中的 y 也可以添加类型约束:

```
y:number = x + 1
```

最重要的特性是，如果在函数的参数上增加类型约束（关于函数，我们将在后面的章节中讨论），那么将使得 WorkScript 这款动态类型语言拥有和静态类型语言一样的安全性和严谨性。

```
add1(x:number) = x + 1
```

第三章：一般运算

3.1 四则运算

四则运算和数学中的四则运算基本上没有区别，所以这里很简单的进行介绍。WorkScript 提供了加法，减法，乘法，除法，取模（取余数），乘方等运算。我们可以用如下的运算符分别来实现上述计算：

```
a + b  加法

a – b  减法

a * b  乘法

2x     乘法（省略乘号，见下面说明）

a / b  除法

a % b  取模

a^b  乘方（a 的 b 次方）
```

上面例子中的 a,b 和 x 都是变量名。如果直接使用值进行计算同样是可以的，例如 1 + 2。

需要特别注意的是，如果使用省略乘号的乘法语句，对书写格式要求较为严格：仅当乘法左边为数字，右边为变量名时才可以省略乘号。交换顺序是不可以的，例如 x2 会被认为名为”x2”的变量，而不是乘法。即使以空格分隔，例如 x 2 也是不允许的。

3.2 赋值运算

对于变量，我们往往需要进行赋值。在第二章中我们已经涉及到了对变量的赋值，在这里我们将对赋值操作进行具体的介绍。我们可以对变量进行赋值，也可以对计算结果为变量的运算进行赋值（例如函数返回了变量，我们在后面的章节中会进行讨论）。赋值的语法如下：

```
变量 = 表达式
```

其中表达式可以是变量，也可以是值，或者四则运算等。但是要注意的是，上面被赋值的必须是变量，不能是值，或者计算结果不是变量的表达式。例如下面的赋值是非法的：

```
2 = 1
```

显然，2 不是一个变量，它不能被赋值。

3.3 比较运算

当我们需要比较两个值的大小时，我们可以使用比较运算。WorkScript 提供的比较运算有：大于，大于等于，小于，小于等于，等于，不等于。它们的写法如下：

```
a > b      a 大于 b
a >= b   a 大于等于 b
a < b      a 小于 b
a <= b   a 小于等于 b
a = b     a 等于 b
a != b   a 不等于 b（或 a <> b）
```

比较运算的结果是布尔值 true 或者 false。例如如果执行 print(2>1)会打印 true，显然 2 大于 1。如果执行 print(2<1)会打印 false，因为 2 并不小于 1。如果执行 print(2=1)会打印 false，因为 2 也不等于 1。

也许细心的读者已经发现了，在上一小节我们刚刚举过一个反例“2=1”，2 不是一个变量，不能被赋值。为什么在这里却出现了“2=1”这样的表达式呢？原因是，上一小节的等号是赋值的含义，而在这一小节中，等号是判等的含义。虽然 2 不是变量，不能被赋值，但是作为一个值，它是可以用来判等的。

那么，如何区分等号到底是用作赋值还是判等呢？有没有更严谨的方式来表达赋值和判等的语义，能够避免这种歧义呢？我们在下一小结对此进行讨论。

3.4 赋值和判等的歧义判定

在上一小节提到，由于赋值和判等两种运算共用等号“=”，导致等号出现了歧义的问题。这看起来是一个棘手的问题，下面让我们看看，在历史的长河中，语言设计者们曾经绞尽脑汁地想到过哪些糟糕的主意：

C 语言：

```
a = b    赋值
a == b   判等
```

Pascal：

```
a := b   赋值
a = b    判等
```

Prolog：

```
A is B   赋值
A = B    判等
```

对于众多编程语言，在这里就不一一列举了。总之，没有一款编程语言让用户能够以一种正常的方式去完成这两件事，直到一向创造优秀产品的微软为我们设计的 Visual Basic 语言的问世：

Visual Basic:

```
a = b   赋值
a = b   判等
```

我们惊奇的发现在 Visual Basic 中，人们终于能够以习惯的方式来进行赋值和判等了。但是，这两者的歧义是如何被解决的呢？由于 WorkScript 的语

法和 Visual Basic 这门传统语言的语法有着天差地别，故感兴趣的读者可以自行翻阅 Visual Basic 的语法。在这里我们介绍 WorkScript 语言的规则：

当表达式处于独立位置的时候，它被认为是赋值表达式。

当表达式处于非独立位置的时候，它被认为是判等表达式。

什么是非独立位置呢？一般的，如果一个表达式被另一个表达式所嵌套，那么它就是非独立位置，例如：

print(a=b) 等号所在的表达式被 *print()*表达式所嵌套，它是判等表达式。

f(x=0) = x 这是一个函数声明，这里的“*x=0*”处于参数表位置，嵌套于函数中，故为判等。

f(x) = 0 when x = 0 这也是一个函数声明，但是 *x=0* 位于函数的约束中。这也是一个非独立位置，视为判等。

f(x) = (x = 1) 这也是一个函数声明，此处 x=1 嵌套于 f(x)函数中，故视为判等。

什么是独立位置呢？一般的，如果表达式不被其他表达式所嵌套，或者处于大括号中非嵌套的位置，则它是独立表达式。例如：

a = b 这显然是一个赋值表达式，没有任何理由将其认作判等表达式。

```
f(x) = {
    a = x
}
```

在这里，a=x 虽然被嵌套在 f(x)中，但处于大括号中非嵌套的位置，故视为赋值表达式。

上面的规则已经详细的说明了赋值和判等表达式的判定。虽然看起来有些复杂，但在实际的使用中是非常自然的，以至于你甚至感受不到赋值和判等所带来的歧义。最后，值得一提的是，如果你真的认为在有些地方你无法确定到底是赋值还是判等，没有关系，我们可以使用下面的 WorkScript 标准赋值表达式和标准判等表达式来避免不必要的麻烦：

```
a := b  赋值

a == b  判等
```

第四章：函数

4.1 函数的介绍

由于函数是个众人皆知的概念，所以我们在前几章已经无数次地提到过函数。但是在这一章，我们将对函数进行详细的讨论。

在 WorkScript 中，函数也是一种值。和数学中的函数非常类似，一个函数拥有函数名，参数和返回值。在一般的计算机语言中，函数还被赋予了更多的特性，例如重载，甚至有些语言突发奇想的允许函数返回多个值（经过实践，这是一个糟糕的创意)。如果只有上述那些特性，那么你只能看到像其他编程语言那样晦涩难懂的代码，而不可能像 WorkScript 一样写出第一章那样简洁易懂的代码。要实现这样的语法，离不开 WorkScript 为函数提供的额外特性：约束重载。对于包含约束的函数，和无约束的函数，WorkScript 称其为特化函数和泛化函数。对于这些概念，我们将在本章进行一一介绍。

4.2 函数的组成

4.2.1 函数名

让我们举一个例子来展示一个完整的 WorkScript 的函数声明。设计一套评分系统，我们假设一个人的成绩区间是 0 到 100 分，其中 0-60 为不合格 ，60-85 为合格，85-100 为优秀。如果成绩不在 1-100 之间，结果为“错误“。我们使用”rate” 作为评分函数的函数名。

```
rate(x) = “不合格” when x >= 0 & x < 60
rate(x < 85) = “合格”
rate(x <= 100) = “优秀”
rate(x) = “错误”
```

在上面的示例中，我们声明了名为 rate 的函数。在后面当我们需要调用 rate 的时候，系统就可以找到这里的 rate 函数的声明，并对其进行调用。值得一提的是，函数名的本质其实是一个变量，当我们声明一个带函数名的函数时，其实相当于创建了这个名字的一个变量，其存储的值为我们的函数。作为变量，我们也可以对其进行赋值和取值等所有变量所具有操作，尽管我们一般从来不会这样做。对于一个函数来说，函数名并不是必须的。当我们声明的函数没有名字时候，就代表我们不需要上述的变量来存储这个函数，而是仅仅声明了一个函数类型的值而已。我们在后面的“匿名函数“小节中会对此进行详细介绍。

4.2.2 参数

分析上面的例子，我们发现 rate 函数均接收一个参数 x。在 WorkScript 中，一个函数可以有 0 个到若干个参数，参数必须写在函数名后面的圆括号中。即使一个函数没有参数，圆括号也不可以省略。如果有多个参数，每个参数之间以逗号分隔。逗号不可以省略。

```
f() = x          //没有参数的情况
f(a) = x         //有一个参数的情况
f(a,b,c) = x     //有多个参数的情况
```

在 WorkScript 中，函数的最后一个参数如果是一个变量，那么它可以匹配所有多余的调用参数。这是一个重要的性质，这个性质在后面的“贪婪匹配参数“小节再做探讨。

4.2.3 约束

上面的 rate 函数中，除了最后一个声明之外，前三个声明都增加了约束条件。WorkScript 的函数的约束条件可以用”when”子句来实现。” when 子句”可以写在函数名之前，函数参数列表和函数实现中间，以及函数实现之后。when 子句的形式如下：

```
when x=1 f(x) = x + 1  //when 在函数名之前
f(x) when x=1 = x + 1  //when 在中间(如果在合适的地方加括
```

```
                                    号，看起来会舒服许多。）
f(x) = x + 1 when x=1   //when 在最后
```

不管 when 子句在哪里，如果 when 子句中包含多行语句，我们都需要增加大括号来包含多条语句（一条语句也可以增加大括号，但显得有些冗余）。以 when 在最后的形式来举例：

```
f(x,y) = x + y
when{
  x > 0
  y > 0
}
```

要注意的是，when 中的两条约束只有全部计算为 true，才可以成功调用该函数。任意一行计算为 false，都会使得匹配失败。

另外，大家也注意到了。rate 函数的例子中，只有第一条使用了 when 子句，而第二和第三两条直接在参数列表里写做了“x <= 100“这样的约束。其实，这些属于 WorkScript 为了方便而设计的语法糖。你可以直接在函数的参数列表中填写大于，小于等约束条件，编译器会自动转换成相应的 when 子句。让我们来看一下这些约束的简单写法，以及他们会在运行的时候自动被转换成什么样的 when 子句：

```
//第一种，比较约束：
f(x <= 100) = x         //转换前
f(x) = x when x<=100   //转换后
```

注意：比较约束的表达式左部必须是参数名。右部可以是任意表达式

上述 x <= 100 不能写作 100 >= x

```
//第二种，值约束：
f(5) = 6                //转换前
f(_0) = 6 when _0=5  //转换后（注意，_0 虽然看起来比较奇怪，
                            但也是合法的变量名）
//第三种：类型约束
f(x : number) = x + 1                  //转换前
f(x) = x + 1 when typeof(x)=number  //转换后

//第四种，参数名约束
equal(x, x) = true                  //转换前
equal(x, y) = false

equal(_0, _1) = true when _0=_1   //转换后
equal(x, y) = false
```

当然，还有一种特殊的约束，不会被转换为 when 子句，它就是返回值的类型约束：

```
f(x) : number = x+1
```

注意，这个 number 的类型约束可不是加在参数 x 上的，而是加在函数 f

上的。它的含义是，函数 f 的返回值一定是数字类型。

最后需要强调的是，当一组参数符合一个函数的多种约束的情况下，书写靠前的约束优先匹配。例如评分的函数 rate 拥有 4 个声明，那么如果输入的参数为 90 的话，显然第三条和第四条声明都可以匹配成功。在这时 WorkScript 约定第三条匹配成功，而不会匹配第四条。

4.2.4 实现

上面的 rate 函数中“合格“,”不合格“,”优秀“和”错误“都叫做函数的实现。只不过上面的例子比较特殊，只有一个值。这样只有一个值的实现，这个值就被当作了返回值（下一小结即将介绍)。一般情况下，函数的实现和约束一样，也可以有一行或者多行语句，并且对于多行语句，必须要用大括号将其包含。

```
f(x) = x + 1  //一行实现
f(x) = {
  x + 1        //一行实现，不省略大括号
}

f(x) = {       //多行实现，不能省略大括号
  y = x
  y + 1
```

```
}
```

值得一提的是，如果一个函数的实现没有省略大括号，那么它可以省略等号。例如带大括号的函数实现，我们可以写作如下形式：

```
f(x) {
  x + 1
}
```

另外，大括号可以选择换行开始或者不换行开始。上面的示例全部使用了不换行的格式，但如果你喜欢将语句写得稀疏一些，也可以考虑下面这两种形式：

```
f(x) =     //这里的=可以省略
{
  x + 1
}
```

另外，考虑到一些简短的函数声明如果使用等号，会显得比较丑陋，因此规定：在任何形式的函数声明中，你都可以使用箭头号”=>”来代替等号”=”。例如下面的函数声明同样是合法的：

```
f(x) => x+1
```

4.2.5 返回值

一个函数的返回值（在数学中称作函数值），是函数计算的结果。例如

rate(x <= 100) = “优秀”，其中“优秀“就是这个函数的返回值。如果一个函数的实现只有一行，那么这一行的计算结果就是返回值。如果函数的实现有多行，那么最后一行的计算结果就被作为返回值。让我们举一个例子来看看多行函数的返回值：

```
add1(x) = {
  print(x)
  x + 1
}
```

显而易见，x + 1 作为 add1 函数的最后一行语句，它的计算结果会被作为 add1 这个函数的返回值。而之前的 print(x)也会被执行，从而打印出参数 x 的值，但计算结果不会影响返回值。如果我们调用 print(add(1))，将会输出如下结果：

```
print(add(1))  //打印出 1 2
```

4.3 函数的调用

4.3.1 按参数顺序传递参数的函数调用

常规的函数调用就是我们日常见到的函数调用，例如 print(1,2,3,4,5)，使用函数名 print 来对 print 函数进行调用。提供 5 个参数，分别按位置传递参数，即第一个参数为 1，第二个参数为 2... 第五个参数为 5。这样的函数调用非常容易理解。

4.3.2 按参数名传递参数的函数调用[尚未实现]

函数的每个参数都是有名字的，我们在调用的时候，可以直接指定某个参数的值。例如当我们想要求一个长方体的体积，我们需要知道其长宽高三个参数，于是我们可以声明如下函数：

```
volume(length, width, height) = length*width*height
```

这就是一个普通的函数声明，我们在调用时当然可以按三个参数的顺序依次传递参数。但是，同样我们也可以按照参数的名称来指定个别的或全部参数，比如先指定宽，然后按顺序传递长和高。

```
volume(width:=5,10,2)
```

这里先指定了 width=5，然后按照顺序，剩下的参数分别是长和高，按照顺序传入了 10 和 2 两个值。这样调用函数，我们同样可以得到 100 的计算结果。需要强调的是，对参数的指定，我们必须使用明确的赋值符号:=，因为在这个语境下，如果使用=，将会被视为判等，例如上面的调用，会被认为是名

为 width 的变量是否等于 5，计算结果为布尔值 true 或 false。

对于按参数名赋值，相当于对参数提前赋值。按名称赋值不会影响正常的按顺序赋值，但是已经被按名称赋值过的参数不会被按顺序重复赋值。例如，上面的例子中，首先对 width 参数进行了赋值(5)，那么后面的 10 和 2 在赋值时，会先对 length 赋值 10，然后跳过 width，继续对 height 赋值为 2。

虽然按参数名赋值不会影响正常的按顺序赋值，但从实践上来说，建议在按顺序赋值之后再按参数名赋值，不要将按参数名赋值和按顺序赋值混在一起，这样非常容易造成错误。

4.3.3 匿名函数的调用

函数的本质也是一种值，当声明的函数没有函数名的时候，这个函数就仅仅是一个函数类型的值，我们将这种函数称为匿名函数。对于匿名函数性质，在下一小结有更加详细的讨论，这里仅讲解对匿名函数的调用。对于匿名函数的调用，和具名函数的调用规则是一样的。从实践上来说，当我们使用匿名函数的时候，往往是临时使用一次，而不需要将其赋值到变量里。那么，我们可以这样直接进行调用（具名函数同样可以）：

```
((x)=>print(x))(“hello world”)
```

这和下面的代码是等效的：

```
f = (x)=>print(x)
f(“hello world”)
```

4.4 函数的性质

4.4.1 函数的重载

重载在所有的现代编程语言中几乎都有提供支持，WorkScript 也不例外。其实在上一节的 rate 函数已经使用了重载的特性，而读者一定并没有特别地感觉到上面的例子有什么特别的地方。因为重载这个特性，几乎是一个看起来理所应当的性质。那么什么是重载呢?

多个函数拥有相同的名称和不同的参数或约束，则互为重载

注意，重载没有标准的定义，因为各个编程语言的语法千差万别。这个定义是在 WorkScript 的语法下，最符合重载特性的一种定义了。举例说明，上面的 rate 函数有四种声明。其实这是四个函数，只不过都叫做 rate 而已，他们拥有不同的参数列表，从而互相构成了重载。重载的重要作用在于，函数可以会根据运行时的参数来动态地确定匹配哪一个重载。

一般我们从来不会在重载的问题上犯错，除了需要强调，下面这种情况不是重载:

```
f(x): number = x + 1

f(x): string = x + ”world”
```

因为两个函数拥有相同的参数列表(x)，并且同样地没有任何约束，所以不属于重载。试想，当你调用 f(1)的时候，由于两个函数的参数列表完全相同，都可以接收 1 这个参数。那么该匹配哪一个函数呢?

4.4.2 泛化函数与特化函数

泛化函数与特化函数的概念非常简单：

对于有参数的函数来说，有约束的函数都是特化函数，没有约束的函数都是泛化函数。如果一个函数没有参数，那么它既是特化函数，也是泛化函数。

下面这个函数是泛化函数：

```
f(x,y,z) = 1
```

而下面这些函数都是特化函数：

```
f(0,y,z) = 1

f(x,y,z>10) = 1

f(x:number, y, z) = 1
```

对于特化函数，不要求必须存在对应的泛化函数。例如下面的特化函数可以单独存在：

```
f(0) = 1
```

而不要求必须存在对应的泛化函数：

```
f(x) = 1
```

4.4.3 递归函数

4.4.3.1 递归函数的定义

“要理解递归，就得先了解什么是递归“，一般来说，调用自己的函数就

是递归函数。下面这个例子是递归的一个非常好的例子：

计算斐波那契数列，数列前两项都是 1，其余每一项都是前两项的和。

为了计算斐波那契函数，我们可能会写出如下的代码：

```
fib(0) = 1

fib(1) = 1

fib(x) = fib(x-1) + fib(x-2)
```

可以看到，fib 函数的第三个重载 fib(x) = fib(x-1)+fib(x-2)中，fib 函数两次调用到了 fib 函数自己。这样的函数就叫做递归函数。由此我们也可以看到，当递归函数调用自己的时候，它不一定真的调用的是自己本身，例如当 x 等于 2 的时候，fib(x-1)会调用第 2 行的 fib(1) = 1 ，它调用的可能是另一个重载（这在其他编程语言里简直是件不可思议的事情）。对递归的掌握，是编写 WorkScript 语言的一项重要技能，每个人都应该熟练掌握。

4.4.3.2 尾递归优化

考虑上面的 repeat 函数，其是符合尾递归优化的条件的。观察 repeat 的第二个重载，我们发现这个函数的最后一行语句为 repeat(n-1, f)。

如果一个递归函数最后一行语句是递归调用它自己，那么它不会真正调用，而是会将当前这次调用的参数更新为新的参数值，然后跳转到函数自己的第一行语句继续执行。这就是尾递归优化。

尾递归优化可以大幅度提高程序的运行性能，同时可以避免当递归层次过深的时候，编程语言会发生“爆栈“而导致崩溃的问题。当然，如果你不是计

算机专业人士，大可不必担心这一点。因为正常的递归逻辑几乎不可能引起爆栈崩溃等问题，如果出现了这样的问题，一定是程序的算法书写错误引起的。

4.4.4 参数贪婪匹配

之前我们提到，如果一个函数的最后一个参数如果是变量，那么在调用函数时当传递的参数个数多于函数声明的参数个数时，函数声明的最后一个参数可以匹配调用时实际传递的所有剩余参数。

我们以一个 max 函数作为例子。max 函数的功能是求出输入数字中最大的数字。我们编写 max 函数，并利用 max 函数求出 1 3 5 2 4 6 中的最大值：

```
max(x, y >= x) = y

max(x, y < x) = x
```

看起来我们已经完成了接收两个参数的 max 函数了。但是如果仅仅是这样显然还不够。难道我们要先先比较 max(1,3)，再比较 max(3,max(1,3))？这实在是太麻烦了。我们希望直接使用 max(1, 3, 5, 2, 4, 6)来得到结果 6。为了实现这样的效果，接下来我们基于上面的两个 max 函数来编写一个能够接收任意多个参数的 max 函数：

```
max(head, tail) = max(head, max(tail))
```

不知道你能不能一眼看懂这个 max 函数在做什么。在这个函数中有两个参数 head 和 tail。顾名思义，第一个是用来匹配输入的第一个参数，而第二个 tail 参数用来匹配输入的所有剩余参数。下面让我们来模拟一下当我们输入 1 3 5 2 4 6 这几个参数时的执行过程：

1. 调用 max(1, 3, 5, 2, 4, 6)，匹配到 max(head,tail) = max(head, max(tail))

   此时 head = 1,tail = 3,5,2,4,6

2. 接下来会计算等号右边的 max(head, max(tail))

3. 于是为了求解 max(head, max(tail))，需要先计算 max(tail)

   也就是 max(3, 5, 2, 4, 6)

4. max(3, 5, 2, 4, 6) 再次匹配到上面的 max(head, tail)函数，

   此时 head = 3 tail = 5 2 4 6

5. 以此类推，重复之前的步骤，直到 max(2, 4, 6) = max(2, max(4, 6))

6. 为了求解 max(2, max(4, 6))，需要先求解 max(4, 6)

7. max(4,6)这次有些不同，它不再匹配 max(head,tail) = max(head, max(tail))，

   而是匹配到 max(x，y >= x) = y 这个重载，因为这个重载的声明更加靠前。

   于是得到 max(4,6) = 6

8. 由于 max(4, 6)得到了 6 这个结果，于是外层的 max(2, max(4, 6))终于可以化简为

   max(2, 6)

9. max(2, 6)也会匹配到 max(x，y >= x) = y 这个重载，得到 6

10. 于是更加外层的 max(5, max(2, 6))也可以计算出结果 6

11. 以此类推，最终计算出最外层的 max(1,6)的结果也为 6

12. 于是最终结果为 6

有时候简单的算法用语言描述就会变得非常繁琐。当你花费了很长时间终于把上面的例子看懂之后，也许你也会惊讶道，原来这么简单。也许你会有疑问，为什么一开始 max(1, 3, 5, 2, 4, 6)没有直接匹配到 max(x, y >= x) = y

呢？答案是因为这个函数中的 y 有一条约束：y>=x。如果尝试匹配这一条，会得到 x=1, y=3, 5, 2, 4, 6。所以 y >= x 会被解释成 3, 5, 2, 4, 6 >= 1 显然，3, 5, 2, 4, 6 并不是一个数字，不能进行大于等于的比较。所以这条匹配就失败了。

看到这里你也许意识到了，如果当你不希望你所编写的函数进行参数贪婪匹配时，你有一些方法来避免参数贪婪匹配特性：

1. 对最后一个参数添加比较约束，例如 f(x, y >= x)
2. 对最后一个参数添加类型约束，例如 f(x, y:number)
3. 对最后一个参数添加值约束，例如 f(x, 0)

除此之外，如果一个函数是外部函数（在后面的小节会进行介绍），那么该函数也不具有参数贪婪匹配的性质。

4.4.5 匿名函数

对于一个函数来说，其函数名不是必须的。在许多情况下，我们需要临时生成函数，这个时候生成的函数往往不需要具有名字。匿名函数的声明语法和普通的函数声明完全一样，也同样可以拥有约束，但仅仅省略函数名：

```
(n) = n + 1 when n > 0
```

这是一个合法的匿名函数，当 n 大于 0 时，返回 n+1 的值。

如果你觉得这样的函数看起来有些别扭，我们可以将函数声明的等号换成箭头=>符号，这样方便我们在茫茫的代码中一眼辨认出匿名函数。

```
(n) => n + 1 when n > 0
```

需要注意的是，匿名函数也可以进行重载！一旦我们将匿名函数赋值到某一个变量中，然后再继续声明与此变量名相同的函数，那么新声明的函数会作为一个重载加入到这个变量所存储的匿名函数中。例如：

```
f = (x)=>x+1

f(0) = 0
```

那么第二行的 f(0)就和第一行的变量 f 中所存储的函数构成了重载。但是请注意，如果按下面的方式赋值，则不能构成重载：

```
f = (x)=>x+1

f = (0)=>0
```

这样仅仅是对变量 f 进行两次赋值。当第二次赋值的时候，第一次赋的值就不再存在于变量 f 中了。由此可见，函数名不仅是变量名，它还具有将函数主动加入重载的特性。而如果使用普通的变量赋值，就不具有这样的特性了。

实际使用中，我们几乎不会用匿名函数进行重载，所以匿名函数往往只有一个声明。在这种情况下，一般我们不为匿名函数设定约束或者仅仅设定比较宽泛的约束（如类型约束），来包容各种可能的输入。

4.4.6 偏函数 [尚未实现]

当一个函数调用的实际参数数量少于函数的参数数量时，该函数调用不会失败，而是会返回接收剩余参数的一组重载，我们称这组重载中的每个函数为

偏函数。这种特性可以用于将一个多参数的函数拆分成多个函数。与之相反的，我们也可以利用柯里化的特性，将多个单参数的函数组合成一个多参数函数。

让我们举个例子：有一个函数 f 接收两个参数 a,b 并依次打印出 a 和 b。它的声明如下：

```
f(a,b) = {
  print(a)
  print(b)
}
```

假如我们调用 f(20,30)，毫无疑问一定会依次打印出 20 和 30。但是如果此时我们调用 f(20)会怎么样呢?

```
f(20)      //什么都不会打印
f(20)(30)  //依次打印 20, 30
```

上面的例子想必很好理解。f(20)的返回值为一个包含参数 20 偏函数。对这个偏函数继续调用，传入参数 30。该偏函数本身拥有一个参数 20，继续接收到 30 后，具有了两个参数，满足函数 f 的声明，故成为普通的函数调用，并进行了执行，打印出 20 和 30。

偏函数在实际的应用中可以配合命名参数的特性发挥出巨大的作用。试想比如我们需要绘制一个图形，这个图形有许多属性，例如长，宽，高，颜色等。那么绘图函数的声明如下：

```
paint(length, width, height, color) = {
```

```
  …（实现省略）
}
```

当我们需要绘制三个颜色相同，形状类似仅仅长度不一样的图形时，我们可以事先将确定的宽度，高度和颜色赋给 paint 函数，获得一个具有上面三个参数的偏函数：

```
my_paint = paint(width:=10, height:=5, color:=’blue’)
```

接下来，绘制三个长度不一样的图形，我们只需要调用 my_paint 函数就可以了。

```
my_paint(5)
my_paint(8)
my_paint(20)
```

这样能够大幅简化我们的代码。同时，如果以后我们需要绘制的图形发生了一些变化，我们只需要修改一遍 my_paint 偏函数中的属性就可以了，而不用繁琐地修改每个绘制图形的函数，大大提高了程序的可维护性。

4.4.7 回调函数

回调(diào)函数，是一种经典的函数使用方法。一般来说，把一个函数传递给其他地方，等合适的时候由其进行调用，那么被传递的这个函数就称作回调函数。下面我们举一个回调函数的例子：

定义一个 repeat 函数，用来重复执行指定的函数若干次。接收两个参数，

第一个参数 n 代表需要重复多少次，第二个参数 f 代表需要执行的函数。

```
repeat(0, f) = ok

repeat(n > 0, f) = {
  f()
  repeat(n-1, f)
}
```

很容易理解，repeat(0, f)=ok 当执行次数是 0 次的时候，直接返回布尔值 ok，而不进行任何其他操作。否则当 n 大于 0 的时候，先执行传入的 f 函数，再执行 repeat(n-1, f)。例如我们按如下方式调用：

```
x = 0

add1() = (x = x+1)

repeat(10,add1)
```

执行完成后，x 的结果会变成 10。显然 add1 函数被调用了 10 次。

在上面这个例子里，add1 就可以被称作回调函数。

上面的例子过于繁琐，类似这种情况，我们往往直接使用匿名函数：

```
x = 0

repeat(10,()=>(x = x+1))
```

另外，使用偏函数作为回调函数是一种常用的手法，实践中我们常常这么做，来简化我们的代码。

4.4.8 闭包函数 [尚未实现]

当一个函数的声明处于另一个函数的内部时，如果该函数引用了外层函数的局部变量，并且将这个函数作为外层函数的返回值进行返回，那么被返回的内层函数就称作闭包函数。对于闭包函数，外层函数虽然已经执行完毕，局部变量都已经不复存在，但是由于闭包函数引用了外层函数的局部变量，所以被引用的局部变量的生命被继续延长，不会随着外层函数的执行结束而销毁。下面举一个例子：

函数 add_n 接收一个参数 n，并生成一个新的匿名函数返回，此匿名函数接收一个参数 x，返回 x+n 的值。

```
add_n(n) = {
  (x) = x + n  //注意这是一个匿名函数，这里 x+n 中的 n 是 add_n
                函数的局部变量
}
```

利用 add_n 函数，我们可以轻易的生成出我们定制的 add 函数，例如：

```
add_1 = add_n(1)
print(add_1(5))  //会打印出 6
```

当我们调用 add_1(5)的时候，就可以得到 6。可以看到，只要 add_1 函数存在，那么其对应的 add_n 里面的局部变量 n 就不会被销毁，其生命周期由于闭包函数 add_1 而得到了延长。

4.4.9 纯函数

4.4.9.1 纯函数的定义

纯函数是一个重要的概念，我们将符合如下条件的函数称为纯函数：

一个函数的返回值只依赖于它的参数，

并且在执行过程里面没有副作用。

什么是返回值只依赖于它的参数呢？让我们首先来看一个反例，你就很容易明白这句话的含义了。

```
next_n_day(n) = day() + n
```

我们定义了一个叫做 next_n_day 的函数，用来计算今天之后 n 天的日期。显然，加入我们在不同的日期输入同样的 n=0,那么 8 月 26 日的时候，next_n_day(0)的返回值为 26，8 月 27 日的时候，返回值则为 27。这时此函数的返回值不仅仅依赖于它的参数，还依赖于当前的时间，所以它不是纯函数。对于纯函数，每次 n=0 输出的结果一定是相同的。

下面解释没有副作用的含义。函数副作用是指当调用函数时，除了返回返回值之外，还对产生附加的影响。例如修改外部变量（函数外的变量）或修改参数等。下面的函数就是有副作用的函数：

```
x = 1

f() = (x = x + 1)
```

函数 f 修改了外部变量 x，所以它具有副作用，不是纯函数。

在一般的编程开发中，纯函数非常“靠谱”，执行一个纯函数你不用担心它会干什么坏事，它不会产生不可预料的行为，也不会对外部产生影响。不管何时何地，你给它什么它就会乖乖地返回什么。如果你的应用程序大多数函数都是由纯函数组成，那么你的程序质量会非常优秀。

值得一提的是，所有的输入输出操作都不是纯函数，例如 print 函数，input 函数等。因为这些函数的执行依赖于显示屏，键盘这些外部设备。所以，当你的函数调用了这些函数之后，你的函数也会失去纯函数的特性，从而不能享受到纯函数特有的优化。

4.4.9.2 纯函数结果缓存［尚未实现］

在 WorkScript 中，有一个其他编程语言从未有过的重要的特性，那就是纯函数结果缓存。纯函数结果缓存很好理解，如果一个函数是纯函数，那么它的调用结果会被缓存，下次调用的时候直接读取缓存结果，而不会再次调用此函数。例如上一小节提到的斐波那契数列函数就是一个纯函数结果缓存的很好的例子。如果没有纯函数结果缓存，假设我们想要计算 fib(20)，那么会经历多少次计算呢？答案可能超出你的想象：20000 多次。

为什么会这么多呢？试想，当我们计算 fib(20) = fib(19)+fib(18)时，我们已经计算了一次 fib(18)。接下来，计算 fib(19) = fib(18)+fib(17)的时候，又一次计算了 fib(18)。所以，在 fib(20)的计算过程中，fib(18)被计算了两遍。根据这个道理，我们可以依次推出 fib(20) fib(19) fib(18) fib(17) fib(16) fib(15)...的计算次数分别是 1 1 2 3 5 8... 读者可以自行推算 20 项的计算次数

总和。那么问题来了，20000 多次计算显然是不必要的，能不能优化呢?

答案是：纯函数结果缓存。显然，fib(x)函数都是纯函数，他们都可以被执行纯函数结果缓存。那么在这种情况下，我们计算 fib(20)只需要计算一次 fib(0), fib(1), fib(2) ... fib(19)的结果就够了。因为下次调用的时候，相同的函数结果会取自缓存，而不会再次计算。这样我们需要多少次计算呢？20 次。

整整减少了 1000 倍的计算量，这就是纯函数结果缓存带来的巨大效益。

4.4.10 外部函数 [尚未实现]

一个孤独的人是做不成大事的，一门孤立的语言也是难以成器的。我们可以使用 WorkScript 来调用其他语言，例如 C/C++程序编写的代码。代码的调用是以函数为单位的，我们可以声明一个外部函数，该函数使用 extern 关键字进行修饰，并且只有函数名，参数列表，类型约束。没有实现和其他约束。例如下面的 f 函数是一个合法的外部函数声明：

```
extern f(x:integer, y:string)
```

需要注意的是，外部函数可以和 WorkScript 函数构成重载。对于外部函数有一些额外的要求，具体如下：

外部函数必须有函数名，可以有 0 到多个参数，但每个参数必须有且仅有类型约束。外部函数不可以有 when 子句和实现部分。

另外需要注意，当一个值作为参数被传递给外部函数，那么外部函数对该值如何处理，WorkScript 是无法知晓的。WorkScript 将依旧以自己的方式去

管理这些值所在的内存部分。所以，不可以在外部函数中释放属于 WorkScript 传入的值，否则当 WorkScript 在释放这个值的时候会引起重复释放的错误而导致崩溃。另外，如果外部函数中需要长期存储由 WorkScript 传入的值，请确保同时在 WorkScript 中保持对该值的引用，以免 WorkScript 的 GC 垃圾回收器将该值认为已经不可达而将其释放。