CSE341：2016年春季编程语言课程
第1单元总结

标准描述：这个总结大致涵盖了课堂和复习部分的内容。 当回顾材料时，以叙述方式阅读材料并将整个单元的材料放在一个文档中可能会有所帮助。 请报告这些笔记中的错误，甚至是打字错误。 这个总结不能完全替代上课、阅读相关代码等。

目录

欢迎来到编程语言课程

课程网页的顶部有四个重要的文件，这里不再重复。 它们是教学大纲、学术诚信政策、挑战项目政策以及关于这门课程与Coursera上的一门课程相关的描述。请仔细阅读这些文件。

一个名为“编程语言”的课程可以有很多不同的含义。 对我们来说，它意味着有机会学习几乎在每种编程语言中都出现的基本概念。
我们还将对这些概念如何“组合在一起”以提供程序员所需的内容有一些了解。 我们将使用不同的语言来看看它们如何以互补的方式表示这些概念。 所有这些都旨在使您成为一名更好的软件开发人员，无论使用哪种语言。

很多人会说这门课程“教授”3种语言ML、Racket和Ruby，但这是相当误导的。 我们将使用这些语言来学习各种范式和概念，因为它们非常适合这样做。如果我们的目标只是让您在这三种语言中尽可能高效地工作，课程材料将会非常不同。 话虽如此，能够学习新的语言并识别不同语言之间的相似之处和差异是一个重要的目标。

大部分课程将使用函数式编程（ML和R*a*cket都是函数式语言），强调不可变数据（没有赋值语句）和函数，特别是接受和返回其他函数的函数。 正如我们将在课程中讨论的那样，函数式编程与面向对象编程完全相反，但也有许多相似之处。 函数式编程不仅是一种非常强大和优雅的方法，而且学习它可以帮助你更好地理解所有编程风格。

在课程的最开始，通常要激发学习兴趣，这种情况下，我们将解释为什么你应该学习函数式编程，更一般地说，为什么值得学习它。

不同的语言、范式和语言概念。 我们将在大约两周后再详细讨论这个问题。 在大多数学生更关心课程工作内容的时候，这个问题太重要了，而且更重要的是，在我们建立了几节共享术语和经验的讲座之后，这个讨论会更容易进行。 动机确实很重要；让我们推迟一下，但承诺它一定是值得的。

ML表达式和变量绑定

所以让我们开始“学习ML”，但以一种教授核心编程语言概念的方式，而不仅仅是“编写一些能工作的代码”。因此，请非常注意用来描述我们开始的非常简单的代码的词语。 我们正在建立一个基础，这个基础将在本周和下周迅速扩展。 暂时不要试图将你看到的内容与你已经了解的其他语言联系起来，因为这可能会导致困难。

一个ML程序是一系列的绑定。 每个绑定都会进行类型检查，然后（假设它通过了类型检查）进行求值。 一个绑定的类型（如果有的话）取决于一个静态环境，大致上是文件中前面绑定的类型。 一个绑定的求值方式取决于一个动态环境，大致上是文件中前面绑定的值。 当我们只说“环境”时，通常指的是动态环境。 有时候，“上下文”被用作“静态环境”的同义词。

有几种绑定方式，但现在我们只考虑一个变量绑定，在ML中的语法如下：

```
val x = e;
```

在这里， `val`是一个关键字， `x`可以是任何变量， `e`可以是任何表达式。 我们将学习很多种写表达式的方法。 分号在文件中是可选的，但在读取-求值-打印循环中是必需的，以便让解释器知道你已经完成了绑定的输入。

我们现在知道了变量绑定的语法（如何编写），但我们仍然需要知道它的语义（如何进行类型检查和求值）。 大多数情况下，这取决于表达式 `e`。 要进行变量绑定的类型检查，我们使用“当前静态环境”（前面绑定的类型）来检查 `e`（这将取决于表达式的类型）并生成一个“新的静态环境”，该环境与当前静态环境相同，只是 `x`具有类型 `t`，其中 `t`是 `e`的类型。 求值类似：要求值一个变量绑定，我们使用“当前动态环境”（前面绑定的值）来求值 `e`（这将取决于表达式的类型）并生成一个“新的动态环境”，该环境与当前环境相同，只是 `x`具有值 `v`，其中 `v`是求值 `e`的结果。

一个值是一个表达式，即“没有更多的计算”，即没有办法简化它。如下面更一般地描述的那样，`17`是一个值，但`8+9`不是。所有的值都是表达式。 并不是所有的表达式都是值。

这整个关于ML程序意义的描述（绑定、表达式、类型、值、环境）可能看起来非常理论化或者晦涩，但这正是我们需要为几种不同类型的表达式给出精确而简洁的定义的基础。 下面是几个这样的定义：

- 整数常量：
  - –语法：一系列数字
  - –类型检查：在任何静态环境中类型为int
  - –评估：在任何动态环境中自身（它是一个值）

[1]这里的单词 *static*与其在Java/C/C++中的使用有一种脆弱的联系，但在这一点上解释起来太脆弱了。

- 加法：
  - 语法：e1+e2，其中e1和e2是表达式
  - 类型检查：类型为int，但仅当e1和e2在相同的静态环境中具有类型int时，否则不进行类型检查
  - 求值：在相同的动态环境中将e1评估为v1，将e2评估为v2，然后产生v1和v2的和
- 变量：
  - 语法：一系列字母、下划线等
  - 类型检查：在当前静态环境中查找变量并使用该类型
  - 求值：在当前动态环境中查找变量并使用该值
- 条件语句：
  - 语法为if e1 then e2 else e3，其中e1，e2和e3是表达式
  - 类型检查：使用当前静态环境，条件语句仅在（a）e1具有类型bool且（b）e2和e3具有相同类型时进行类型检查。 整个表达式的类型是e2和e3的类型。
  - 评估：在当前动态环境下，评估 e1。 如果结果为 true，则在当前动态环境下评估 e2的结果是整体结果。 如果结果为false，则在当前动态环境下评估 e3的结果是整体结果。
- 布尔常量：
  - 语法：要么 true要么 false
  - 类型检查：在任何静态环境中都是 bool类型
  - 评估：在任何动态环境中都是自身（它是一个值）
- 小于比较：
  - 语法： e1 < e2 其中 e1 和 e2 是表达式
  - 类型检查：类型为 bool，但仅当 e1和 e2在相同的静态环境中具有 int类型时，否则不进行类型检查
  - 评估：在相同的动态环境中将 e1评估为 v1，将 e2评估为 v2，然后产生 true如果 v1小于 v2，否则产生 false

每当你学习一个新的编程语言构造时，你应该问这三个问题：什么是语法？ 什么是类型检查规则？ 什么是评估规则？

使用用

当使用读取-求值-打印循环时，从文件中添加一系列绑定非常方便。

```
使用"foo.sml";
```

就是这样。 它的类型是 unit，其结果是 ()（类型为 unit的唯一值），但其效果是包含文件"foo.sml"中的所有绑定。

变量是不可变的

绑定是不可变的。给定`val x = 8+9;`我们产生一个动态环境，其中 `x`映射到 `17`。 在这个环境中， `x`将始终映射到 `17`；在ML中没有“赋值语句”来更改`x`映射到的内容。如果你正在使用 `x`，这非常有用。 你以后可以有另一个绑定，比如`val x = 19;`，但这只会创建一个不同的环境，其中后面的绑定对 `x`进行了屏蔽。当我们定义使用变量的函数时，这个区别将非常重要。

函数绑定

回想一下，ML程序是一系列绑定的序列。 每个绑定都会添加到静态环境（用于类型检查后续绑定）和动态环境（用于评估后续绑定）。我们已经介绍了变量绑定；现在我们介绍函数绑定，即如何定义和使用函数。 然后，我们将学习如何使用对和列表从较小的数据构建和使用较大的数据块。

函数有点像Java等语言中的方法-它是通过参数调用并产生结果的东西。 与方法不同，没有类、this等概念。我们也没有像return语句这样的东西。 一个简单的例子是计算 x^y的函数，假设 $y \geq 0$：

```
fun pow (x:int, y:int) = （仅当y >= 0时正确）
    如果 y=0
    那么 1
    否则 x * pow(x,y-1)
```

语法：

函数绑定的语法如下（我们稍后会对这个定义进行泛化）：

```
    fun x0 (x1 : t1, ..., xn : tn) = e
```

这是一个名为 `x0`的函数绑定。 它有 n个参数 `x1`, ... 类型分别为 `t1`, ..., `tn`，并且有一个表达式 `e`作为其主体。. 正如我们所知，语法只是语法 - 我们必须为函数绑定定义类型规则和求值规则。 但大致上来说，在 `e`中，参数被绑定到 `x1`, ...xn，而调用 `x0`的结果是求值 `e`的结果。

类型检查：

为了对函数绑定进行类型检查，我们在一个静态环境中对主体 `e`进行类型检查（除了所有之前的绑定之外），将 `x1`映射到 `t1`, ... 将 `xn`映射到 `tn`，将 `x0`映射到`t1 * ... * tn -> t`。. 由于 `x0`在环境中，我们可以进行递归函数调用，即函数定义可以使用自身。 函数类型的语法是“参数类型” `->`“结果类型”，其中参数类型由 `*`分隔（这恰好是用于乘法表达式的相同字符）。 对于函数绑定来说，主体 `e`必须具有类型 `t`，即 `x0`的结果类型。 根据下面的求值规则，这是有意义的，因为函数调用的结果是求值 `e`的结果。

但是，确切地说，是 `t`- 我们从未写下来过？ 它可以是任何类型，由类型检查器（语言实现的一部分）来确定 `t`应该是什么，以便将其用作`x0`的结果类型使得“一切都能正常工作”。目前，我们将其视为神奇的，但是类型推断（推断未写下的类型）是ML中的一个非常酷的功能，稍后在课程中讨论。 事实证明，在ML中，你

几乎不需要写下类型。 很快，参数类型 t1，...， tn也将是可选的，但是在稍后学习模式匹配之前不是。[2]

在函数绑定之后，将x0添加到静态环境中并附带其类型。 参数不会添加到顶级静态环境中 - 它们只能在函数体中使用。

评估：

函数绑定的评估规则很简单：函数是一个值 - 我们只需将x0作为一个稍后可以调用的函数添加到环境中。 对于递归，如预期的那样，x0在函数体中和后续绑定中都在动态环境中（但不像在Java中那样，对于前面的绑定不是，所以定义函数的顺序非常重要）。

函数调用：

函数绑定只在函数调用时有用，这是一种新的表达式。 语法是e0 (e1,...,en)如果只有一个参数，则括号是可选的。 类型规则要求e0的类型看起来像t1*...*tn->t，并且对于1$\leq i \leq n$，ei的类型是ti。 然后整个调用的类型是t。希望这不会太令人惊讶。 对于求值规则，我们使用调用点的环境来评估e0为$v0$，e1为$v1$，...，en为vn。 然后$v0$必须是一个函数（假设调用已经通过类型检查），我们在扩展的环境中评估函数的主体，使函数参数映射到$v1$，...，vn。

我们扩展哪个环境来包含参数？ 函数被定义时的“当前”环境，而不是调用时的环境。 这个区别现在不会出现，但我们将在以后详细讨论。

将所有这些放在一起，我们可以确定这段代码将产生一个环境，其中 ans是 64：

```
fun pow (x:int, y:int) = （仅当y >= 0时正确）
    如果 y=0
    那么 1
    否则 x * pow(x,y-1)

fun cube (x:int) =
    pow(x,3)

val ans = cube(4)
```

对偶和其他元组

编程语言需要一种将简单数据组合成复合数据的方法。 我们将在ML中学习的第一种方法是对偶。 构建对偶的语法是 (e1,e2)，它将 v1评估为 v1，将 v2评估为 v2，并且生成值为 (v1,v2)的对偶，它本身也是一个值。 由于 $v1$和/或 $v2$本身可能是对偶（可能包含其他对偶等），我们可以构建具有多个“基本”值的数据，而不仅仅是两个整数。
对偶的类型是 t1*t2，其中 t1是第一部分的类型， t2是第二部分的类型。

就像制作函数只有在我们可以调用它们时才有用，制作对只有在我们以后可以检索到各个部分时才有用。 在我们学习模式匹配之前，我们将使用 #1和 #2来访问第一个和第二个部分。 对于#1 e或#2 e的类型规则应该不会让人感到意外： e必须具有类似于ta * tb的某种类型，然后#1 e的类型为 ta，#2 e的类型为 tb。

下面是使用对的几个示例函数。 div_mod可能是最有趣的，因为它使用了一个对来返回具有两个部分的答案。

[2]我们在本单元和作业1中使用对读取构造，需要这些显式类型。

对返回具有两个部分的答案，这非常令人愉快。 这在ML中非常愉快，而在Java中（例如）从函数返回两个整数需要定义一个类，编写一个构造函数，创建一个新对象，初始化其字段，并编写一个返回语句。

```
fun swap (pr : int*bool) =
    (#2 pr, #1 pr)

fun sum_two_pairs (pr1 : int*int, pr2 : int*int) =
    (#1 pr1) + (#2 pr1) + (#1 pr2) + (#2 pr2)

fun div_mod (x : int, y : int) = (* 注意：在Java中返回一个对是真正的痛苦 *)
    (x div y, x mod y)

fun sort_pair (pr : int*int) =
    if (#1 pr) < (#2 pr)
    then pr
    else ((#2 pr),(#1 pr))
```

事实上，ML通过允许任意数量的部分来支持元组。 例如，一个由整数组成的3元组（即三个整数）具有类型int*int*int。 一个例子是 (7,9,11)，你可以用#1 e，#2 e和#3 e来检索部分，其中e是一个求值为三元组的表达式。

对偶和元组可以嵌套在任何你想要的地方。 例如，(7，（true，9）)是一个类型为int * （bool * int）的值，这与((7，true)，9)的类型(int * bool) * int或(7，true，9)的类型int * bool * int不同。

列表

虽然我们可以嵌套成对的成对（或元组）深入到我们想要的地方，但对于任何具有对的变量，返回对的任何函数等，都必须有一对的类型，该类型将确定“真实数据”的数量。即使使用元组，类型也指定了它有多少部分。这通常太过限制性；当我们进行类型检查时，我们可能需要一个数据列表（例如整数），但列表的长度尚未知道（可能取决于函数参数）。 ML有列表，它比对更灵活，因为它们可以具有任意长度，但不够灵活，因为任何特定列表的所有元素必须具有相同的类型。

空列表，语法为 []，有0个元素。 它是一个值，所以像所有值一样，它立即求值为自身。
它可以具有类型t list，其中 t可以是任何类型，ML将其写作'a list（发音为“引号 a list”或“alphalist”）。 一般来说，类型t list描述了列表中所有元素的类型为 t。 无论 t是什么，对于 []都成立。

具有 n个值的非空列表写作[v1,v2,...,vn]。 你可以用[e1,...,en]创建一个列表，其中每个表达式都会求值为一个值。 更常见的是用 e1 :: e2创建一个列表，发音为“ e1consed onto e2”。这里 e1求值为“类型为 t的项”， e2求值为“类型为 t的值的列表”，结果是一个以 e1的结果开头，然后是 e2中的所有元素的新列表。

与函数和对一样，只有在我们可以对它们做一些操作时，制作列表才有用。 与
对一样，在我们学习模式匹配之后，我们将改变如何使用列表，但现在我们将使用ML提供的三个函数作为参数来使用列表。

- null对于空列表返回true，对于非空列表返回false。
- hd返回列表的第一个元素，如果列表为空则引发异常。

- `tl`返回列表的尾部（与其参数相似的列表，但不包含第一个元素），如果列表为空则引发异常。

这里有一些接受或返回列表的简单示例：

```
fun sum_list (xs : int list) =
    if null xs
    then 0
    else hd(xs) + sum_list(tl xs)

fun countdown (x : int) =
    if x=0
    then []
    else x :: countdown(x-1)

fun append (xs : int list, ys : int list) =
    if null xs
    then ys
    else (hd xs) :: append(tl xs, ys)
```

几乎所有生成和使用列表的函数都是递归的，因为列表的长度是未知的。 编写递归函数时，思考过程包括考虑基本情况 - 例如，对于空列表应该是什么答案 - 以及递归情况 - 如何用列表的其余部分的答案来表示答案。

当你以这种方式思考时，许多问题变得更简单，这让习惯于思考while循环和赋值语句的人感到惊讶。 一个很好的例子是上面的append函数，它接受两个列表并生成一个将一个列表附加到另一个列表的列表。 这段代码实现了一种优雅的递归算法：如果第一个列表为空，则可以通过直接返回第二个列表来进行附加。 否则，我们可以将第一个列表的尾部附加到第二个列表。 这几乎是正确的答案，但我们需要“cons on”（使用::被称为“consing”几十年）第一个列表的第一个元素。

这里没有什么神奇的地方 - 我们不断地使用较短的第一个列表进行递归调用，然后
当递归调用完成时，我们将递归调用中删除的列表元素重新添加回去。

最后，我们可以随意组合对和列表，而无需为我们的
语言添加任何新功能。 例如，这里有几个函数，它们接受一组整数对的列表。 请注意，
最后一个函数重复使用之前的函数，以实现非常简短的解决方案。 这在函数式
编程中非常常见。 事实上，我们应该感到困扰的是， `firsts`和 `seconds`如此相似，但我们没有让它们
共享任何代码。 我们将在以后学习如何修复这个问题。

```
fun sum_pair_list (xs : (int * int) list) =
    if null xs
    then 0
    else #1 (hd xs) + #2 (hd xs) + sum_pair_list(tl xs)

fun firsts (xs : (int * int) list) =
    if null xs
    then []
    else (#1 (hd xs))::(firsts(tl xs))

fun seconds (xs : (int * int) list) =
```

```
    if null xs
    then []
    else (#2 (hd xs))::(seconds(tl xs))

fun sum_pair_list2 (xs : (int * int) list) =
    (sum_list (firsts xs)) + (sum_list (seconds xs))
```

Let 表达式

Let-表达式是一种非常重要的特性，可以以一种简单、通用和灵活的方式实现局部变量。Let-表达式对于风格和效率都非常重要。 Let-表达式允许我们拥有局部变量。 实际上，它允许我们拥有任何类型的局部绑定，包括函数绑定。 因为它是一种表达式，所以它可以出现在任何表达式可以出现的地方。从语法上讲，let-表达式的形式是：

```
   let b1 b2 ... bn in e end
```

其中每个bi都是一个绑定，e是一个表达式。

let表达式的类型检查和语义与我们的ML程序中的顶层绑定的语义非常相似。 我们逐个评估每个绑定，为后续的绑定创建一个更大的环境。 因此，我们可以将之前的绑定用于后续的绑定，并且我们可以将它们全部用于 e。 我们将绑定的作用域称为“可以使用它的地方”，因此let表达式中绑定的作用域是该let表达式中的后续绑定和let表达式的“主体”（ e）。 表达式 eevaluates的值是整个let表达式的值，而 e的类型是整个let表达式的类型。

例如，这个表达式的值为7；请注意一个内部绑定 x遮蔽了外部绑定。

```
   let val x = 1
   in
       (let val x = 2 in x+1 end) + (let val y = x+2 in y+1 end)
   end
```

还要注意let表达式是表达式，所以它们可以作为加法中的子表达式出现（尽管这个例子很愚蠢且风格不好，因为很难阅读）。

let表达式也可以绑定函数，因为函数只是另一种绑定的方式。 如果一个辅助函数只被另一个函数使用，并且不太可能在其他地方有用，将其局部绑定是很好的风格。
例如，这里我们使用一个局部辅助函数来帮助生成列表 [1,2,...,x]：

```
fun countup_from1 (x:int) =
    let fun count (from:int, to:int) =
            if from=to
            then to::[]
            else from :: count(from+1,to)
    in
        count(1,x)
    end
```

然而，我们可以做得更好。 当我们评估对 count的调用时，我们在动态环境中评估 count的主体，该环境是 count被定义的环境，并扩展了 count参数的绑定。

上面的代码实际上没有使用这个： count的主体只使用 from, to和 count（用于递归）。 它也可以使用 x，因为在定义 count时，它在环境中。 然后我们根本不需要 to，因为在上面的代码中它总是与 x具有相同的值。所以这样更好：

```
fun countup_from1_better (x:int) =
    let fun count (from:int) =
            if from=x
            then x::[]
            else from :: count(from+1)
    in
        count 1
    end
```

这种技术 - 定义一个使用作用域中的其他变量的局部函数 - 是函数式编程中非常常见和方便的事情。 可惜许多非函数式语言几乎没有支持这样做的功能。

对于保持代码可读性来说，局部变量通常是很好的风格。 当它们绑定到可能昂贵的计算结果时，它们可能比这更重要。 例如，考虑这段不使用let表达式的代码：

```
fun bad_max (xs : int list) =
    if null xs
    then 0 (* 注意：不好的风格；见下文 *)
    else if null (tl xs)
    then hd xs
    else if hd xs > bad_max(tl xs)
    then hd xs
    else bad_max(tl xs)
```

如果你用countup_from1 30调用bad_max，它将大约进行2^{30}（超过十亿）次递归调用它自己。 原因是“指数爆炸” - 代码两次调用bad_max(tl xs)，每次调用都会再次调用bad_max两次（总共四次），依此类推。这种编程“错误”很难检测，因为它可能取决于您的测试数据（如果列表倒数，算法只进行30次递归调用而不是2^{30}次）。

我们可以使用let表达式避免重复计算。 这个版本计算列表的尾部的最大值，并将结果存储在tl_ans中。

```
fun good_max (xs : int list) =
    if null xs
    then 0 (* 注意：不好的风格；见下文 *)
    else if null (tl xs)
    then hd xs
    else
        (* 为了风格，也可以使用 let-binding 来获取 hd xs *)
        let val tl_ans = good_max(tl xs)
        in
            if hd xs > tl_ans
            then hd xs
            else tl_ans
        end
```

选项

前面的例子没有正确处理空列表 —— 它返回了 0。 这是不好的风格，因为0 实际上不是 0 个数字的最大值。 没有一个好的答案，但我们应该合理地处理这种情况。 一种可能性是引发一个异常；如果你对此感兴趣，可以在我们在课程中讨论它们之前自行了解 ML 异常。 相反，让我们将返回类型更改为要么返回最大数，要么指示输入列表为空，因此没有最大值。 根据我们拥有的构造，我们可以通过返回一个 int list 来“编码”，如果输入为空列表，则使用 []，如果输入列表不为空，则返回一个包含一个整数（最大值）的列表。

虽然这样可以工作，但是列表是“过度设计” - 我们总是返回一个包含0个或1个元素的列表。 所以列表并不是一个真正精确的描述我们返回的内容。 ML库有“选项”，它们是一个精确的描述：一个选项值要么是0个东西，要么是1个东西： NONE是一个“不携带任何东西”的选项值，而SOME e评估为一个值 v并成为携带这个值 v的选项。 NONE的类型是'a option，而SOME e的类型是t option，如果 e的类型是 t。

给定一个值，你如何使用它？ 就像我们有 null来判断一个列表是否为空，我们有 isSome来判断其参数是否为 NONE，如果是则返回 false。 就像我们有 hd和 tl来获取列表的部分（对于空列表会引发异常），我们有 valOf来获取 SOME携带的值（对于 NONE会引发异常）。

使用选项，这是一个带有返回类型int option的更好版本：

```
fun better_max (xs : int list) =
    如果 null xs
    则 NONE
    否则
        let val tl_ans = better_max(tl xs)
        如果 isSome tl_ans 并且 valOf tl_ans > hd xs
           则 tl_ans
           否则 SOME (hd xs)
        end
```

上面的版本工作得很好，是一个合理的递归函数，因为它不会重复任何可能昂贵的计算。 但是，每个递归调用（除了最后一个）都创建一个带有 SOME的选项，以便其调用者可以访问其下面的值，这既笨拙又低效。 这里是一种替代方法，我们在非空列表中使用一个本地辅助函数，然后只需让外部函数返回一个选项。 请注意，如果使用 []调用辅助函数，它将引发异常，但由于它是在本地定义的，我们可以确保这种情况永远不会发生。

```
fun better_max2 (xs : int list) =
    if null xs
    then NONE
    else let (* 假设参数非空因为它是局部的 *)
            fun max_nonempty (xs : int list) =
                if null (tl xs) (* xs 必须不是 [] *)
                then hd xs
                else let val tl_ans = max_nonempty(tl xs)
                     in
                         if hd xs > tl_ans
                         then hd xs
                         else tl_ans
                     end
```

```
in
    SOME (max_nonempty xs)
end
```

其他一些表达式和运算符

ML拥有你所需的所有算术和逻辑运算符，但语法有时与大多数语言不同。 这里是一些其他有用的表达式的简要列表：

- e1 andalso e2是逻辑与：只有当 e1评估为 true时，它才会评估 e2。 如果 e1和 e2都评估为true，则结果为true。 显然， e1和 e2都必须具有 bool类型，整个表达式也具有 bool类型。 在许多语言中，这样的表达式被写为e1 && e2，但这不是ML的语法，也不是and是我们稍后会遇到的用于不同目的的关键字。 使用e1 andalso e2通常比等价的if e1 then e2 else false更好的风格。

- e1 orelse e2是逻辑或：只有当e1为false时，才会计算e2。 如果e1或e2计算结果为true，则结果为true。 自然地，e1和e2都必须具有bool类型，并且整个表达式也具有bool类型。 在许多语言中，这样的表达式写作e1 || e2，但这不是ML语法，也不是e1 or e2。 使用e1 orelse e2通常比等价的if e1 then true else e2更好。

- not e是逻辑非。 not是一个类型为bool->bool的提供的函数，我们也可以自己定义为fun not x = if x then false else true。. 在许多语言中，这样的表达式写作!e，但在ML中，!运算符表示其他意思（与可变变量相关，我们不会使用）。

- 你可以比较许多值，包括整数，使用e1 = e2来判断它们是否相等。
- 与其写not (e1 = e2)来判断两个数字是否不同，更好的风格是e1 <> e2。 在许多语言中，语法是e1 != e2，而ML的 <>可以记作“小于或大于”。
- 其他算术比较与大多数语言的语法相同： >, <, >=, <=。
- 减法写作e1 - e2，但它必须有两个操作数，所以你不能只写- e来表示否定。 对于否定，正确的语法是~ e，特别是负数写作 ~7，而不是 -7。 对于整数来说，使用 ~e是更好的风格，与0 - e等效。

缺乏变异及其好处

在ML中，没有办法改变绑定、元组或列表的内容。 如果 x映射到某个值，比如在某个环境中的键值对列表 [(3,4),(7,9)]，那么 x将永远映射到该列表。
没有赋值语句可以将 x映射到不同的列表。 （你可以引入一个新的绑定来隐藏 x，但这不会影响任何查找“原始” x的代码在环境中的结果。）没有赋值语句可以改变列表的头部或尾部。 也没有赋值语句可以改变元组的内容。 因此，我们有用于构建复合数据和访问其部分的结构，但没有用于改变我们构建的数据的结构。

这是一个非常强大的特性！这可能会让你感到惊讶：一个语言没有某个特性怎么可能成为一个特性呢？ 因为如果没有这样的特性，那么当你编写你的代码时，你可以依赖于没有其他代码做一些可能使你的代码错误、不完整或难以使用的事情。拥有

不可变数据可能是一种语言最重要的“非特性”，也是函数式编程的主要贡献之一。

虽然不可变数据有各种优势，但在这里我们将重点关注一个重要的优势：它使共享和别名无关紧要。 在批评Java（以及其他任何默认使用可变数据和赋值语句的语言）之前，让我们重新考虑上面的两个例子。

```
fun sort_pair (pr : int*int) =
    if (#1 pr) < (#2 pr)
    then pr
    else ((#2 pr),(#1 pr))
```

在 sort_pair中，我们明显在else分支中构建并返回一个新的pair，但在then分支中，我们返回的是 pr所引用的pair的副本还是引用，像这样的调用者：

```
val x = (3,4)
val y = sort_pair x
```

现在 x和 y是对同一个pair的别名吗？ 答案是无法确定— ML中没有构造可以确定 x和 y是否是别名的，也没有理由担心它们可能是别名。如果有变异，情况就会不同。 假设我们可以说，“将pair x的第二部分更改为5而不是4。”那么我们就要想知道#2 y是4还是5。

如果你好奇的话，我们希望上面的代码会创建别名：通过返回 pr，
sort_pair函数将返回对其参数的别名。 这比这个版本更高效，它
会创建另一个具有完全相同内容的对：

```
fun sort_pair (pr : int*int) =
    if (#1 pr) < (#2 pr)
    then (#1 pr, #2 pr)
    else ((#2 pr),(#1 pr))
```

创建新的对(#1 pr, #2 pr)是不好的风格，因为 pr更简单，同样有效。 然而，在具有变异的语言中，程序员经常像这样制作副本，完全是为了防止像使用一个变量 x进行赋值会导致意外更改使用另一个变量 y。 在ML中， sort_pair的任何用户都无法知道我们是否返回一个新的对。

我们的第二个例子是我们优雅的列表追加函数：

```
fun append (xs : int list, ys : int list) =
    if null xs
    then ys
    else (hd xs) :: append(tl xs, ys)
```

我们可以提出一个类似的问题：返回的列表与参数共享任何元素吗？ 再次回答并不重要，因为没有调用者能够知道。 而且再次回答是肯定的：我们构建一个新的列表，它“重用”了 ys的所有元素。 这节省了空间，但如果后来有人可以改变 ys的话，会非常令人困惑。 节省空间是不可变数据的一个好处，但当编写优雅的算法时，不必担心是否存在别名也是一个好处。

事实上， tl本身幸运地引入了别名（尽管你无法知道）：它返回（一个别名）列表的尾部，这总是“廉价”的，而不是复制列表的尾部，对于长列表来说是“昂贵”的。

这个 append示例与 sort_pair示例非常相似，但它更加引人注目，因为如果你有许多可能很长的列表，很难跟踪潜在的别名。 如果我将[1,2]附加到 [3,4,5]，我将得到一些列表[1,2,3,4,5]，但如果后来有人可以更改 [3,4,5]列表为
[3,7,5]是附加的列表仍然[1,2,3,4,5]还是现在是[1,2,3,7,5]？

在类似的Java程序中，这是一个关键问题，这就是为什么Java程序员必须过分关注
何时使用对旧对象的引用以及何时创建新对象。 有时候过分关注
别名是正确的做法，有时候避免变异是正确的做法 - 函数式
编程将帮助您更好地掌握后者。

作为最后一个例子，以下Java代码是一个重要
（并随后修复的）Java库中一个实际安全漏洞的关键思想。 假设我们正在维护访问权限
类似于磁盘上的文件。 让每个人都可以看到谁有权限是可以的，但显然只有那些
具有权限的人才能实际使用资源。 考虑这个错误的代码（如果不是
相关的部分已省略）：

```
class ProtectedResource {
   private Resource theResource = ...;
   private String[] allowedUsers = ...;
   public String[] getAllowedUsers() {
      return allowedUsers;
   }
   public String currentUser() { ... }
   public void useTheResource() {
      for(int i=0; i < allowedUsers.length; i++) {
         if(currentUser().equals(allowedUsers[i])) {
             ... // access allowed: use it
             return;
         }
      }
      throw new IllegalAccessException();
   }
}
```

你能找到问题吗？ 这里是：getAllowedUsers返回allowedUsers数组的别名，所以任何用户都可以通过getAllowedUsers()[0] = currentUser()来获得访问权限。 糟糕！ 如果我们在Java中有一种不允许更新其内容的数组，这将是不可能的。 相反，在Java中，我们经常需要记住复制一份。 下面的修正显示了一个明确的循环，以详细说明必须做什么，但更好的风格是使用类似System.arraycopy或Arrays类中的类库方法 - 这些类库方法存在是因为数组复制是必然常见的，部分原因是由于变异。

```
public String[] getAllowedUsers() {
   String[] copy = new String[allowedUsers.length];
   for(int i=0; i < allowedUsers.length; i++)
      copy[i] = allowedUsers[i];
   return copy;
}
```

CSE341：程序设计语言 2016年春季
第2单元总结

标准描述：这个总结大致涵盖了课堂和复习部分的内容。 当回顾材料时，以叙述方式阅读材料并将整个单元的材料放在一个文档中可能会有所帮助。 请报告这些笔记中的错误，甚至是打字错误。 这个总结不能完全替代上课、阅读相关代码等。

目录

编程语言的组成部分

现在我们已经学习了足够的ML知识来编写一些简单的函数和程序，我们可以列举出定义和学习任何编程语言所需的基本“组成部分”：

- 语法：如何编写语言的各个部分？
- 语义：各种语言特性的含义是什么？ 例如，表达式如何求值？
- 习惯用法：使用语言特性表达计算的常见方法是什么？
- 库：已经为你编写好的内容是什么？ 如果没有库的支持，你如何做一些无法完成的事情（比如访问文件）？

- 工具：用于操作语言中的程序的可用工具（编译器，读取-求值-打印循环，调试器等）

虽然库和工具对于成为一个有效的程序员至关重要（避免重新发明已有的解决方案或不必要地手动完成任务），但本课程并不过多关注它们。 这可能会给人留下错误的印象，认为我们在使用“愚蠢”或“不切实际”的语言，但是库和工具在一门关于编程语言的概念相似性和差异性的课程中并不那么重要。

构建新类型的概念方法

编程语言具有基本类型，如整数、布尔值和单位，以及复合类型，这些类型在其定义中包含其他类型。 我们已经看到了在ML中创建复合类型的方法，即使用元组类型、列表类型和选项类型。 我们将很快学习到创建更加灵活的复合类型和为我们的新类型命名的新方法。 创建复合类型实际上只有三个基本构建块。 任何合理的编程语言都以某种方式提供这些构建块：[1]

- “Each-of”：一个复合类型 `t`描述包含 `t1`， `t2`，...，和 `tn`类型的值。
- “One-of”：一个复合类型 `t`描述包含 `t1`， `t2`，...，或 `tn`类型的值。
- “自引用”：一个复合类型 `t`可以在其定义中引用自身，以描述递归的数据结构，如列表和树。

对于大多数程序员来说，每个类型都是最熟悉的。 元组是一个例子： `int * bool`描述包含一个 `int`和一个 `bool`的值。 具有字段的Java类也是每个类型的一种。

每个类型也很常见，但在许多入门编程课程中往往没有强调。int option是一个简单的例子：这种类型的值包含一个 `int`或者不包含。对于在ML中包含一个 `int`或者一个 `bool`的类型，我们需要数据类型绑定，这是本课程的主要重点。 在具有类似Java的类的面向对象语言中，通过子类化实现了一种类型，但这是本课程的一个更高级的主题。

自引用允许类型描述递归数据结构。 这在与each-of和one-of类型结合使用时非常有用。 例如，int list描述了既不包含任何内容也不包含一个 `int`和另一个`int list`的值。 任何编程语言中的整数列表都可以用或,和自引用来描述，因为这就是整数列表的含义。

当然，由于复合类型可以嵌套，我们可以有任意嵌套的each-of、one-of和自引用。 例如，考虑类型`(int * bool) list list * (int option) list * bool`。

记录：另一种“each-of”类型的方法

记录类型是每个组件都是一个命名字段的“each-of”类型。 例如，类型 `{foo : int, bar : int*bool, baz : bool*int}`描述了具有三个名为 `foo`、 `bar`和baz的字段的记录。

[1]作为一个术语，你不需要知道，“each-of types”，“one-of types”，和“self-reference types”这些术语不是标准的 - 它们只是思考这些概念的好方法。 通常人们在谈论这些概念时只使用特定语言的结构，比如“元组”。 编程语言研究人员使用术语“乘积类型”，“和类型”和“递归类型”。为什么是乘积和和呢？ 这与布尔代数中的事实有关，其中0表示假，1表示真，和的工作方式类似于乘法和加法。

baz。 这只是一种新的类型，就像我们学习元组类型时一样。

一个记录表达式构建一个记录值。 例如，表达式
{bar = (1+2,true andalso true), foo = 3+4, baz = (false,9) }将求值为记录值
{bar = (3,true), foo = 7, baz = (false,9)}，它可以具有类型
{foo : int, bar : int*bool, baz : bool*int}因为字段的顺序并不重要（我们使用字段名）。 一般来说，记录表达式的语法是{f1 = e1, ..., fn = en}其中，如常，每个 ei可以是任何表达式。 这里的每个 f可以是任何字段名（尽管每个必须不同）。
字段名基本上是任何字母或数字的序列。

在ML中，我们不需要声明我们想要具有特定字段名和字段类型的记录类型 -我们只需写下一个记录表达式，类型检查器会给它正确的类型。 记录表达式的类型检查规则并不令人意外：对每个表达式进行类型检查以获得某个类型 ti，然后构建具有正确字段和类型的记录类型。 因为字段名的顺序并不重要，当打印时，REPL总是按字母顺序排列它们，以保持一致性。

记录表达式的评估规则是类似的：将每个表达式评估为一个值，并创建相应的记录值。

现在我们知道如何构建记录值，我们需要一种访问它们的部分的方法。 暂时，我们将使用
#foo e其中 foo是一个字段名。 类型检查要求 e具有一个带有名为 foo的字段的记录类型，并且如果该字段具有类型 t，则#foo e的类型就是它的类型。 评估将 e评估为一个记录值，然后生成 foo字段的内容。

按名称与按位置，语法糖和关于元组的真相

记录和元组非常相似。 它们都是“每个”构造，允许任意数量的组件。 唯一的真正区别是记录是“按名称”，元组是“按位置”。这意味着对于记录，我们通过使用字段名来构建它们和访问它们的部分，所以我们在记录表达式中写入字段的顺序并不重要。 但是元组没有字段名，所以我们使用位置（第一个，第二个，第三个，...）来区分组件。

在设计语言结构或选择使用哪种方式时，按名称与按位置是一个经典的决策，每种方式在特定情况下更方便。 作为一个大致的指南，按位置对于少量组件来说更简单，但对于较大的复合类型来说，记住每个位置是什么变得太困难。

Java方法参数（以及我们迄今为止描述的ML函数参数）实际上采用了一种混合方法：方法体使用变量名称来引用不同的参数，但调用者按位置传递参数。 还有其他语言，调用者按名称传递参数。

尽管存在“按名称与按位置”，记录和元组仍然非常相似，我们可以完全根据记录来定义元组。以下是如何做到的：

- 当你写(e1,...,en)时，它是写{1=e1,...,n=en}的另一种方式，即，元组表达式是一个具有字段名称 1、2、...、n的记录表达式。
- 类型t1 * ... * tn只是另一种写法{1:t1, ..., n:tn}.
- 注意#1 e,#2 e等现在已经表示正确的含义：获取名为1, 2等的字段的内容。

[2]短语“按名称调用” 实际上在函数参数方面意味着其他事情。这是一个不同的主题。

事实上，这就是ML实际上定义元组的方式：元组是一个记录。 也就是说，元组的所有语法只是一种方便的方式来书写和使用记录。 REPL总是在可能的情况下使用元组语法，因此如果你计算`{2=1+2, 1=3+4}`它会将结果打印为 `(7,3)`。 使用元组语法是更好的风格，但我们不需要为元组提供自己的语义：我们可以使用上面的“另一种写法”规则，然后重用记录的语义。

这是我们将看到的许多示例中的第一个***syntactic sugar***. 我们说，“元组只是记录的语法糖，字段名为$1, 2, ..., n$.” 这是 *syntactic*，因为我们可以用等效的记录语法来描述元组的一切。 这是因为它使语言更甜。 术语 *syntactic*sugar 被广泛使用。 语法糖是保持编程语言中关键思想的好方法（使其更容易实现），同时为程序员提供方便的编写方式。 事实上，在第一次作业中，我们在不知道记录存在的情况下使用了元组，尽管元组是记录。

数据类型绑定：我们自己的“一种”类型

我们现在介绍*datatype bindings*，它是变量绑定和函数绑定之后的第三种绑定方式。
我们从一个愚蠢但简单的例子开始，因为它将帮助我们看到数据类型绑定的许多不同方面。我们可以写：

```
datatype mytype = TwoInts of int * int
                | Str of string
                | Pizza
```

大致上，这定义了一个新的类型，其中的值可以是一个`int * int`或者一个字符串或者什么都没有。 任何值也会被“标记”上信息，以便我们知道它是哪个 *variant*：这些“标记”，我们称之为*constructors*，分别是 `TwoInts`， `Str`和 `Pizza`。 两个构造函数可以用来标记相同类型的底层数据；实际上，即使我们的示例对于每个变体使用了不同的类型，这种情况也很常见。

更准确地说，上面的示例向环境中添加了四个东西：

- 一个新的类型 `mytype`，我们现在可以像任何其他类型一样使用它
- 三个 *constructors* `TwoInts`, `Str`和 `Pizza`

一个*constructor*是两个不同的东西。 首先，它可以是一个用于创建新类型值的函数（如果变体具有某种类型 `t`的`of t`），否则它实际上是新类型的一个值。 在我们的示例中，`TwoInts`是一个类型为`int*int -> mytype`的函数， `Str`是一个类型为`string->mytype`的函数， `Pizza`是一个类型为 `mytype`的值。 其次，我们在case表达式中使用constructors，如下面所述。

所以我们知道如何构建类型为 `mytype`的值：使用正确类型的表达式调用构造函数（它们是函数）（或者只需使用 `Pizza`值）。 这些函数调用的结果是“知道它们是哪个变体”的值（它们存储了一个“标签”），并且将底层数据传递给构造函数。 REPL将这些值表示为`TwoInts(3,4)`或`Str "hi"`。

剩下的是一种检索这些部分的方法...

ML如何不提供对数据类型值的访问

给定一个类型为 `mytype`的值，我们如何访问其中存储的数据？ 首先，我们需要确定它是哪个变体，因为类型为 `mytype`的值可能是由 `TwoInts`、 `Str`或 `Pizza`创建的，这会影响可用的数据。

一旦我们知道我们有哪个变体，然后我们可以访问该变体携带的部分（如果有的话）。 一旦我们知道我们有哪个变体，然后我们可以访问该变体携带的部分（如果有的话）。

回想一下我们之前对于列表和选项的处理方式，它们也是一种类型：我们有用于测试我们所拥有的变体的函数（null或 isSome），以及用于获取各个部分的函数（hd， tl或 valOf），如果给定了错误的变体参数，则会引发异常。

ML可以采用相同的方法来处理数据类型绑定。 例如，它可以采用我们上面的数据类型定义，并向环境中添加函数 isTwoInts， isStr和 isPizza，它们的类型都是mytype -> bool。 它还可以添加类似 getTwoInts（类型为mytype -> int*int）和getStr（类型为mytype -> string）的函数，这些函数可能会引发异常。

但是ML不采用这种方法。 相反，它采用了更好的方法。 你可以使用更好的方法自己编写这些函数，尽管这样做通常不是一个好的风格。实际上，在学习了更好的方法之后，我们将不再像之前那样使用列表和选项的函数 - 我们只是为了逐步学习而开始使用这些函数。

ML如何提供对数据类型值的访问：Case表达式

更好的方法是*case*表达式。 这是一个关于我们示例数据类型绑定的基本示例：

```
fun f x = (* f的类型是mytype -> int *)
    case x of
        Pizza => 3
      | TwoInts(i1,i2) => i1 + i2
      | Str s => String.size s
```

从某种意义上说，case表达式类似于更强大的if-then-else表达式：像条件表达式一样，它评估两个子表达式：首先是在 case和 of关键字之间的表达式，然后是与之匹配的第一个分支中的表达式。 但是，与其有两个分支（一个为 true，一个为false），我们可以为我们的数据类型的每个变体都有一个分支（我们将在下面进一步推广）。与条件表达式一样，每个分支的表达式必须具有相同的类型（在上面的示例中为int），因为类型检查器无法知道将使用哪个分支。

每个分支的形式为p => e其中 p是一个 *pattern*， e是一个表达式，我们用 |字符分隔分支。 模式看起来像表达式，但不要把它们当作表达式。 相反，它们用于匹配对评估案例的第一个表达式的结果（即 case之后的部分）。 这就是为什么评估案例表达式被称为模式匹配的原因。

目前（很快将被显著推广），我们保持模式匹配简单：每个模式使用不同的构造函数，模式匹配选择“正确的”分支，给定单词case之后的表达式。 评估该分支的结果是整体答案；不会评估其他分支。 例如，如果TwoInts(7,9)传递给 f，则选择第二个分支。

这解决了使用one-of类型的“检查变体”部分，但模式匹配还处理了“获取底层数据”的部分。 由于 TwoInts有两个值，它“携带”，它的模式可以（现在必须）使用两个变量（ (i1,i2)）。 作为匹配的一部分，值的相应部分（继续我们的例子，7和9）绑定到用于评估相应右侧的环境中（(i1+i2)）。 从这个意义上说，模式匹配就像一个*let*表达式：它在局部范围内绑定变量。 类型检查器知道这些变量具有什么类型，因为它们在创建用于模式中的构造函数的数据类型绑定中指定。

为什么case表达式比函数更适合测试变体和提取部分？

- 我们永远不会“搞砸”并尝试从错误的变体中提取东西。 也就是说，我们不会像使用hd []那样得到异常。
- 如果case表达式忘记了一个变体，那么类型检查器会给出一个警告信息。 这表示评估case表达式找不到匹配的分支，此时它会引发一个异常。 如果没有这样的警告，那么你就知道这种情况不会发生。
- 如果case表达式使用一个变体两次，那么类型检查器会给出一个错误信息，因为其中一个分支永远不可能被使用。
- 如果你仍然想要像 null和 hd这样的函数，你可以很容易地自己编写它们（但不要在作业中这样做）。
- 模式匹配比我们目前所示的更加通用和强大。我们在下面给出“完整的真相”。

“One-of”类型的有用示例

现在让我们考虑几个例子，其中“one-of”类型很有用，因为到目前为止，我们只考虑了一个愚蠢的例子。

首先，它们非常适合列举一组固定的选项 - 比使用小整数要好得多。例如：

```
datatype suit = 梅花 | 方块 | 红心 | 黑桃
```

许多语言都支持这种枚举类型，包括Java和C，但是ML采取了下一步，允许变体携带数据，因此我们可以做这样的事情：

```
datatype rank = 杰克 | 女王 | 国王 | A | 数字 of int
```

然后，我们可以使用each-of类型将这两个部分组合起来：梅花 * rank

当您在不同的情况下具有不同的数据时，one-of类型也很有用。 例如，假设您想通过学生的学号来识别他们，但是如果有些学生没有学号（也许他们是新来的大学生），那么您将使用他们的全名（包括名字、可选的中间名和姓氏）。 这个数据类型绑定直接捕捉到了这个想法：

```
数据类型 id = 学生编号 of int
                        | 字符串名称 * (字符串选项) * 字符串
```

不幸的是，这种类型的例子经常是程序员对一种类型的深刻缺乏理解，并坚持使用每种类型的例子，这就像用锯子当锤子一样（它能工作，但你在做错事）。 考虑这样的糟糕代码：

```
(* 如果 student_num 是 -1，则使用其他字段，否则忽略其他字段 *)
{student_num : int, first : string, middle : string option, last : string}
```

这种方法要求所有的代码都遵循注释中的规则，没有类型检查器的帮助。
它还浪费空间，每个记录中都有不应该使用的字段。

另一方面，如果我们想为每个学生存储他们的id号码（如果他们有的话）和他们的全名，每种类型都是完全正确的方法：

```
{ student_num : int option,
  first       : 字符串,
  中间        : 字符串选项,
  最后        : 字符串 }
```

我们的最后一个例子是一个包含常量、否定、加法和乘法的算术表达式的数据定义。

```
数据类型 exp = 常量 int
             | 取反 exp
             | 加法 exp * exp
             | 乘法 exp * exp
```

由于自引用，这个数据定义实际上描述的是树，其中叶子是整数，内部节点要么是带有一个子节点的否定，要么是带有两个子节点的加法，要么是带有两个子节点的乘法。 我们可以编写一个函数，它接受一个表达式并对其进行求值：

```
fun eval e =
    情况 e of
        常量 i => i
      | 取反 e2 => ~ (eval e2)
      | 加法(e1,e2) => (eval e1) + (eval e2)
      | 乘法(e1,e2) => (eval e1) * (eval e2)
```

因此，这个函数调用的求值结果为15：

```
eval (加法 (常量 19, 取反 (常量 4)))
```

注意构造函数只是我们用其他表达式（通常是从构造函数构建的其他值）调用的函数。

我们可能会对类型为 exp的值编写许多函数，其中大多数函数将以类似的方式使用模式匹配和递归。以下是您可以编写的其他处理 exp参数的函数：

- 表达式中最大的常数
- 表达式中所有常数的列表（使用列表附加）
- 指示表达式中是否至少有一个乘法的 bool值
- 表达式中加法表达式的数量

这是最后一个：

```
fun number_of_adds e =
    案例 e of
        常数 i            => 0
      | 取反 e2           => number_of_adds e2
      | 加 (e1, e2)       => 1 + number_of_adds e1 + number_of_adds e2
      | 乘 (e1, e2) => number_of_adds e1 + number_of_adds e2
```

数据类型绑定和案例表达式到目前为止，准确地说

我们可以总结我们对数据类型和模式匹配的了解如下：绑定

```
数据类型t = C1 of t1 | C2 of t2 | ... | Cn of tn
```

引入一个新类型 t和每个构造函数 Ci是一个类型为 ti->t的函数。 对于一个不携带任何内容的变体，可以省略“of ti”，这样的构造函数只有类型 t。 为了“获取” t的各个部分，我们使用一个案例表达式：

```
情况 e of p1 => e1 | p2 => e2 | ... | pn => en
```

一个案例表达式将 e评估为一个值 v，找到第一个模式 pi与 v匹配，并评估 ei以产生整个案例表达式的结果。 到目前为止，模式看起来像Ci(x1,...,xn)其中 Ci是类型为t1 * ... * tn -> t（或者只是 Ci如果 Ci不携带任何内容）。 这样的模式匹配一个形式为Ci(v1,...,vn)的值，并将每个 xi绑定到 vi以评估相应的 ei。

类型同义词

在继续讨论数据类型之前，让我们将其与另一种有用的绑定方式进行对比，该方式还引入了一个新的类型名称。 类型同义词只是为现有类型创建另一个名称，该名称与现有类型完全可互换。

例如，如果我们写入：

```
类型 foo = int
```

那么我们可以在任何写入int的地方写入foo，反之亦然。 因此，给定一个类型为foo->foo的函数，我们可以用3调用该函数，并将结果加上4。 REPL有时会打印foo，有时会打印int，这取决于情况；细节并不重要，由语言实现决定。 对于像int这样的类型，这样的同义词并不是很有用（尽管稍后在学习ML的模块系统时，我们将在此功能的基础上构建）。

但对于更复杂的类型，创建类型同义词可能很方便。 以下是我们上面创建的一些类型的示例：

```
类型 card = suit * rank

类型 name_record = { student_num : int option,
                     first       : 字符串,
                     中间        : 字符串选项,
                     最后        : 字符串 }
```

只需记住这些同义词是完全可互换的。 例如，如果一个作业问题需要一个类型为card -> int的函数，并且REPL报告你的解决方案的类型为suit * rank -> int，这是可以的，因为类型是“相同的”。

相比之下，数据类型绑定引入了一个与任何现有类型都不相同的类型。 它创建了一个产生此新类型值的构造函数。 因此，例如，与 suit相同的唯一类型是 suit，除非我们后来为其引入一个同义词。

列表和选项是数据类型

因为数据类型定义可以是递归的，所以我们可以使用它们来为列表创建自己的类型。 例如，
这个绑定对于整数的链表非常有效：

```
数据类型 my_int_list = Empty
                     | Cons int * my_int_list
```

我们可以使用构造函数 Empty和 Cons来创建my_int_list的值，并且我们可以使用case表达式来使用这些值
：[3]

```
val one_two_three = Cons(1,Cons(2,Cons(3,Empty)))

fun append_mylist (xs,ys) =
    情况 xs of
        Empty => ys
      | Cons(x,xs') => Cons(x, append_mylist(xs',ys))
```

事实证明，列表和选项“内置”（即，预定义了一些特殊的语法支持）只是数据类型。 从风格上讲，使用内置的广为人知的功能比发明自己的功能更好。

更重要的是，使用模式匹配来访问列表和选项值比使用之前看到的函数null，hd，tl，isSome和valOf更好的风格。 （我们之前使用它们是因为我们还没有学习到模式匹配，并且我们不想延迟练习我们的函数式编程技能。）

对于选项，你只需要知道SOME和NONE是构造函数，我们用它们来创建值（就像以前一样）并在模式中访问这些值。 下面是一个简短的例子：

```
fun inc_or_zero intoption =
    情况 intoption of
          无 => 0
      | 一些 i => i+1
```

列表的故事与一些方便的语法特点类似： []真的是一个不携带任何东西的构造函数，而 ::真的是一个携带两个东西的构造函数，但 ::是不寻常的，因为它是一个中缀运算符（它被放置在其两个操作数之间），在创建事物和模式中都是如此：

```
fun sum_list xs =
    情况 xs of
        [] => 0
      | x::xs' => x + sum_list xs'

fun append (xs,ys) =
    情况 xs of
        [] => ys
      | x::xs' => x :: append(xs',ys)
```

[3]在这个例子中，我们使用一个变量 xs'。 许多语言不允许在变量名中使用字符 '，但是ML允许，并且在数学中使用它并发音为“exes prime”是常见的。

注意这里 x和 xs'只是通过模式匹配引入的局部变量。 我们可以使用任何我们想要的变量名。 我们甚至可以使用 hd和 tl- 这样做只会遮蔽外部环境中预定义的函数。

通常情况下，你应该优先选择模式匹配来访问列表和选项，而不是像 null和 hd这样的函数 - 原因与一般情况下的数据类型绑定相同：你不能忘记情况，你不能应用错误的函数等等。那么为什么ML环境会预定义这些函数，如果这种方法是次优的呢？ 部分原因是因为它们对于作为参数传递给其他函数非常有用，这是本课程下一节的主要内容。

多态数据类型

除了奇怪的 []和 ::的语法之外，内置的列表和选项与我们的示例数据类型绑定唯一的区别是内置的列表和选项是多态的 - 它们可以用于携带任何类型的值，正如我们在int list，int list list，(bool * int) list等中所见。你也可以对自己的数据类型绑定做到这一点，事实上，这对于构建"通用"数据结构非常有用。

虽然我们在这里不会专注于使用这个功能（即，你不需要知道如何做），但这并不是非常复杂的。例如，这正是选项在环境中预定义的方式：

```
datatype 'a option = NONE | SOME of 'a
```

这样的绑定并不引入一个 *type* option。 相反，它使得如果 t是一个类型，那么t option就是一个类型。 你也可以定义接受多个类型的多态数据类型。 例如，这是一个二叉树，其中内部节点保存类型为 'a的值，叶子节点保存类型为 'b的值。

```
datatype ('a,'b) tree = Node of 'a * ('a,'b) tree * ('a,'b) tree
                      | Leaf of 'b
```

然后我们有像(int,int) tree（其中每个节点和叶子节点都保存一个 int）和(string,bool) tree（其中每个节点保存一个 string，每个叶子节点保存一个 bool）这样的类型。 你使用构造函数和模式匹配的方式对待常规数据类型和多态数据类型是相同的。

模式匹配用于每个类型：关于值绑定的真相

到目前为止，我们已经使用模式匹配来处理一种类型，但我们也可以用它来处理每种类型。 给定一个记录值{f1=v1,...,fn=vn}，模式{f1=x1,...,fn=xn}匹配并绑定 xi到 vi。 正如你可能期望的那样，模式中字段的顺序并不重要。 与以前一样，元组是记录的语法糖：模式(x1,...,xn)与{1=x1,...,n=xn}相同，并匹配元组值(v1,...,vn)，这与{1=v1,...,n=vn}相同。 因此，我们可以为求和三个部分的int * int * int编写这个函数：

```
fun sum_triple (triple : int * int * int) =
    case triple of
      (x,y,z) => z + y + x
```

而且，一个类似的记录示例（和ML的字符串连接运算符）可能是这样的：

```
fun full_name (r : {first:string,middle:string,last:string}) =
    case r of
        {first=x,middle=y,last=z} => x ^ " " ^ y ^ " " ^z
```

然而，只有一个分支的case表达式是很糟糕的风格 - 它看起来很奇怪，因为这样的表达式的目的是区分 *cases* ，复数。那么，当我们知道单个模式肯定匹配时，我们应该如何使用模式匹配来提取值呢？ 事实证明，你也可以在val绑定中使用模式！所以这种方法更好：

```
fun full_name (r : {first:string,middle:string,last:string}) =
    let val {first=x,middle=y,last=z} = r
    in
        x ^ " " ^ y ^ " " ^z
    end
fun sum_triple (triple : int*int*int) =
    let val (x,y,z) = triple
    in
        x + y + z
    end
```

实际上，我们可以做得更好：就像模式可以在val绑定中用于绑定变量（例如， `x`， `y`，和 `z`）到表达式的各个部分（例如， `triple`），我们可以在定义函数绑定时使用模式，并且该模式将用于通过匹配传递给函数的值来引入绑定。 所以这是我们示例函数的第三种也是最好的方法：

```
fun full_name {first=x,middle=y,last=z} =
    x ^ " " ^ y ^ " " ^z
fun sum_triple (x,y,z) =
    x + y + z
```

这个版本的 `sum_triple`应该引起你的兴趣：它以一个三元组作为参数，并使用模式匹配将三个变量绑定到函数体中的三个部分以供使用。 但它看起来就像一个接受 `int`类型的三个参数的函数。 确实，对于三个参数的函数来说，它的类型是`int*int*int->int`，还是对于接受三元组的单参数函数来说？

事实证明，我们基本上是在撒谎：在ML中没有多参数函数的概念：ML中的每个函数都只接受一个参数！每次我们编写一个多参数函数时，实际上是在编写一个以元组作为参数并使用模式匹配来提取元素的一参数函数。 这是一个非常常见的习惯用法，很容易被忽视，而且在与朋友讨论ML代码时谈论“多参数函数”是完全可以的。 但从实际的语言定义来看，它确实是一个一参数函数：是将其展开为第一个版本的 `sum_tripl e`的语法糖，其中包含一个单臂的case表达式。

这种灵活性有时很有用。 在C和Java等语言中，你不能让一个函数/方法计算结果并立即传递给另一个多参数函数/方法。 但是对于作为元组的一参数函数，这是可以的。 下面是一个愚蠢的例子，我们通过“向左旋转两次”来“向右旋转三元组”：

```
fun rotate_left (x,y,z) = (y,z,x)
fun rotate_right triple = rotate_left(rotate_left triple)
```

更一般地，即使函数的编写者
是按照多个参数的方式思考的，你也可以计算元组并将它们传递给函数。

那么零参数函数呢？ 它们也不存在。 绑定fun f () = e使用了
单元模式 ()来匹配传递单元值 ()的调用，这是类型为 unit的唯一值。
类型unit只是一个只有一个构造函数的数据类型，它不带任何参数并使用不寻常的
语法 ()。 基本上， 数据类型unit = () 是预定义的。

插曲：类型推断

通过使用模式来访问元组和记录的值，而不是 #foo，你会发现在函数参数上不再需要写类型。 事实上，在ML中，通常会省略它们- 你总是可以使用REPL来查找函数的类型。 之前我们需要它们的原因是因为#foo不能提供足够的信息来对函数进行类型检查，因为类型检查器不知道记录应该具有哪些其他字段，但是上面介绍的记录/元组模式提供了这些信息。 在ML中，每个变量和函数都有一个类型（否则你的程序将无法通过类型检查） -类型推断只是指你不需要写下类型。

因此，我们上面的例子中使用了模式匹配而不是 #middle或 #2需要参数类型。
通常更好的风格是编写这些更简洁的版本，其中最后一个是最好的：

```
fun sum_triple triple =
    案例 triple of
      (x,y,z) => z + y + x
fun sum_triple triple =
    让 val (x,y,z) = triple
    在
        x + y + z
    结束
fun sum_triple (x,y,z) =
    x + y + z
```

这个版本需要在参数上明确指定类型：

```
fun sum_triple (triple : int * int * int) =
    #1 triple + #2 triple + #3 triple
```

原因是类型检查器无法进行推断

```
fun sum_triple triple =
    #1 triple + #2 triple + #3 triple
```

并推断出参数必须具有类型int*int*int，因为它也可以具有类型int*int*int*int或int*int*int*string或int*int*int*bool*string或无限多种其他类型。 如果你不使用 #，ML几乎不需要显式类型注释，这要归功于类型推断的便利性。

事实上，类型推断有时会揭示函数比你想象的更通用。
考虑一下这段代码，它使用了元组/记录的一部分：

```
fun partial_sum (x,y,z) = x + z
fun partial_name {first=x, middle=y, last=z} = x ^ " " ^ z
```

在这两种情况下，推断出的函数类型表明 y 的类型可以是 任何 类型，所以我们可以调用
partial_sum (3,4,5)或者partial_sum (3,false,5)。

我们将在未来的章节中讨论这些多态函数以及类型推断的工作原理，因为它们是课程的重要主题。 暂时停止使用 #，停止写入参数类型，如果你看到偶尔出现的类型如 'a或 'b，不要感到困惑，这是由于类型推断，稍后会更详细地讨论。

插曲：多态类型和相等类型

我们现在鼓励您在程序中不要使用显式类型注释，但正如上面所示，这可能导致意外的通用类型。 假设您被要求编写一个类型为int*int*int -> int的函数，它的行为类似于上面的partial_sum，但REPL正确地指示partial_sum的类型为int*'a*int->int。 这是可以的，因为多态性表明partial_sum具有更一般的类型。. 如果您可以获取包含'a，'b，'c等的类型，并一致地替换每个类型变量以获得您“想要”的类型，那么您拥有比所需类型更一般的类型。

作为另一个例子，我们编写的 append的类型为'a list * 'a list -> 'a list，因此通过一致地将'a替换为 string，我们可以像string list * string list -> string list那样使用 append。 我们可以对任何类型进行这样的操作，不仅仅是 string。 实际上，我们并没有做任何事情：这只是一种心理锻炼，用于检查一个类型是否比我们需要的类型更一般。 请注意，像'a这样的类型变量必须一致地替换，这意味着 append的类型不比string list * int list -> string list更一般。

你也可以看到带有两个前导撇号的类型变量，比如 ''a。 这些被称为等式类型，并且它们是ML的一个相当奇怪的特性，与我们目前的学习无关。 基本上，ML中的 =运算符（用于比较事物）适用于许多类型，不仅仅是 int，但它的两个操作数必须具有相同的类型。
例如，它适用于 string以及元组类型，其中元组中的所有类型都支持相等性（例如，int * (string * bool)。但它并不适用于每种类型。[4]像 ''a这样的类型只能用“等式类型”替代。

fun same_thing(x,y) = if x=y then "yes" else "no"（具有类型 ''a * ''a -> string）
fun is_three x = if x=3 then "yes" else "no"（具有类型 int -> string）

再次强调，我们将在以后讨论多态类型和类型推断，但这个离题对于避免在作业2中产生困惑是有帮助的：如果你编写的函数比REPL给出的类型更一般，那是可以的。 还记得，如上所述，如果REPL使用的类型同义词与你期望的不同，也是可以的。

嵌套模式

事实证明，模式的定义是递归的：在我们的模式中放置一个变量的任何地方，我们都可以放置另一个模式。 粗略地说，模式匹配的语义是被匹配的值必须与模式具有相同的“形状”，并且变量绑定到“正确的部分”。（这是一个非常模糊的解释，所以下面描述了一个精确的定义。）例如，模式a::(b::(c::d))将匹配至少有3个元素的任何列表，并且它将把 a绑定到第一个元素， b绑定到第二个元素， c绑定到第三个元素， d绑定到包含所有其他元素的列表（如果有的话）。 另一方面，模式a::(b::(c::[]))只匹配具有三个元素的列表。 另一个嵌套模式是 (a,b,c)::d，它匹配任何非空的三元组列表，并将 a绑定到第一个组件

[4]它对于函数不起作用，因为无法确定两个函数是否总是做相同的事情。 它也不适用于类型 real以强制执行规则，由于浮点值的舍入，比较它们几乎总是错误的算法。

头部， b到头部的第二个组件， c到头部的第三个组件，和 d到列表的尾部。

一般来说，模式匹配是关于取一个值和一个模式，并且（1）决定模式是否与值匹配，（2）如果匹配，则将变量绑定到值的正确部分。 以下是模式匹配优雅递归定义的一些关键部分：

- 变量模式（x）匹配任何值 v并引入一个绑定（从 x到 v）。
- 模式 C匹配值 C，如果 C是不携带数据的构造函数。
- 模式C p其中 C是构造函数， p是模式，如果C v（注意构造函数相同）匹配值，则匹配形式C v。 它引入了与 v匹配的 p匹配引入的绑定。

- 模式(p1,p2,...,pn)匹配元组值(v1,v2,...,vn)如果 p1匹配 v1并且 p2匹配v2，...，并且 pn匹配 vn。 它介绍了递归匹配引入的所有绑定。
- （类似的情况是形如{f1=p1,...,fn=pn}的记录模式 ...）

这个递归定义以两种有趣的方式扩展了我们之前的理解。 首先，对于带有多个参数的构造函数C，我们不必像C(x1,...,xn)那样编写模式，尽管我们经常这样做。我们也可以写C x；这将将 x绑定到值C(v1,...,vn)所携带的元组。 实际上，所有构造函数都接受0个或1个参数，但是这1个参数本身可以是一个元组。 所以C(x1,...,xn)实际上是一个嵌套模式，其中(x1,...,xn)部分只是匹配所有具有 n个部分的元组的模式。 其次，更重要的是，当我们想要匹配具有特定“形状”的值时，我们可以使用嵌套模式而不是嵌套case表达式。

还有其他类型的模式。 有时我们不需要将变量绑定到值的一部分。 例如，考虑计算列表长度的函数：

```
fun len xs =
    case xs of
       [] => 0
     | x::xs' => 1 + len xs'
```

我们不使用变量 x。 在这种情况下，最好不要引入变量。 相反，通配符模式 _匹配任何值（就像变量模式匹配任何值一样），但不引入绑定。所以我们应该写：

```
fun len xs =
    case xs of
       [] => 0
      | _::xs' => 1 + len xs'
```

根据我们的一般定义，通配符模式很简单：

- 通配符模式（_）匹配任何值 v并且不引入绑定。

最后，您可以在模式中使用整数常量。 例如，模式 37匹配值 37并且不引入绑定。

嵌套模式的有用示例

使用嵌套模式的优雅示例，而不是嵌套的case表达式的混乱，是“压缩”或“解压缩”列表（在此示例中有三个）：[5]

```
异常 BadTriple

fun zip3 list_triple =
    案例 list_triple of
        ([],[],[]) => []
      | (hd1::tl1,hd2::tl2,hd3::tl3) => (hd1,hd2,hd3)::zip3(tl1,tl2,tl3)
      | _ => raise BadTriple

fun unzip3 lst =
    情况 lst of
        [] => ([],[],[])
      | (a,b,c)::tl => 让 val (l1,l2,l3) = unzip3 tl
                       在
                           (a::l1,b::l2,c::l3)
                       结束
```

这个例子检查一个整数列表是否已排序：

```
fun nondecreasing intlist =
    情况 intlist of
          [] => 真
        | _::[] => 真
      | head::(neck::rest) => (head <= neck 并且 nondecreasing (neck::rest))
```

有时候，通过匹配两个值来比较它们也是一种优雅的方式。 这个例子用于确定在不执行乘法的情况下乘法的符号，有点愚蠢，但是它演示了这个思想：

```
数据类型 sgn = P | N | Z

fun multsign (x1,x2) =
  让 fun sign x = 如果 x=0 那么 Z 否则 如果 x>0 那么 P 否则 N
  在
      情况 (sign x1,sign x2) of
          (Z,_) => Z
        | (_,Z) => Z
        | (P,P) => P
        | (N,N) => P
        | _     => N (* 很多人说这是不好的风格；我对此表示赞同 *)
  end
```

最后一个case的风格值得讨论：当你在底部包含一个“捕获所有”情况时，你放弃了对你是否忘记了任何情况的检查：毕竟，它匹配了之前的所有情况没有匹配的任何东西，所以类型检查器肯定不会认为你忘记了任何情况。所以你需要额外小心

[5] 下面讨论了异常，但它们不是这个例子的重要部分。

如果使用这种技术要小心，枚举剩余的情况可能更不容易出错（在这种情况下（N，P）和（P，N））。 类型检查器仍然会确定没有遗漏的情况是有用的和非平凡的，因为它必须推理出使用（Z，_）和（_，Z）来确定没有遗漏的类型sgn * sgn。

可选：函数绑定中的多个情况

到目前为止，我们已经在case表达式中看到了对一种类型的模式匹配。 我们还看到了在val或函数绑定中对每种类型进行模式匹配的良好风格，这就是“多参数函数”的真正含义。但是在val/function绑定中有没有一种方式可以匹配一种类型？ 这似乎是一个坏主意，因为我们需要多个可能性。 但是事实证明，ML在函数定义中有特殊的语法来实现这一点。 这里有两个例子，一个是我们自己的数据类型，一个是列表：

```
数据类型 exp = 常量 of int | 取反 of exp | 加 of exp * exp | 乘 of exp * exp

fun eval (Constant i) = i
  | eval (Negate e2) = ~ (eval e2)
  | eval (Add(e1,e2)) = (eval e1) + (eval e2)
  | eval (Multiply(e1,e2)) = (eval e1) * (eval e2)

fun append ([],ys) = ys
  | append (x::xs',ys) = x :: append(xs',ys)
```

作为一个品味问题, 你的讲师从来不太喜欢这种风格, 你必须把括号放在正确的位置。 但这在ML程序员中很常见，所以你也可以使用。 作为一个语义问题, 它只是一个语法糖，对应一个单一的函数体，该函数体是一个case表达式:

```
fun eval e =
    情况 e of
              常量 i => i
      | 取反 e2 => ~ (eval e2)
      | 加法(e1,e2) => (eval e1) + (eval e2)
      | 乘法(e1,e2) => (eval e1) * (eval e2)

fun append e =
    案例 e of
       ([],ys) => ys
     | (x::xs',ys) => x :: append(xs',ys)
```

一般来说，语法

```
fun f p1 = e1
|   f p2 = e2
...
|   f pn = en
```

只是语法糖：[6]

[6]作为技术性问题， x必须是外部环境中尚未定义并且被函数中的一个表达式使用的某个变量。

```
fun f x =
   case x of
     p1 => e1
   | p2 => e2
...
   | pn => en
```

注意 append示例使用了嵌套模式：每个分支都匹配一对列表，通过将模式
（例如 []或 x::xs'）放在其他模式中。

异常

ML内置了异常的概念。 你可以使用 *raise*（也称为 *throw*）原语引发异常。 例如，标准库中的 hd函数在使用 []调用时会引发 List.Empty异常：

```
fun hd xs =
    情况 xs of
        []      => 引发 List.Empty
      | x::_ => x
```

您可以使用异常绑定创建自己的异常类型。 异常可以选择携带值，这样引发异常的代码可以提供更多信息：

```
异常 MyUndesirableCondition
异常 MyOtherException of int * int
```

异常类型很像数据类型绑定的构造函数。 实际上，它们是函数（如果它们携带值）或值（如果它们不携带值），用于创建类型为 exn 而不是数据类型的值。因此，Empty、MyUndesirable*e*Condition 和 MyOtherException(3,9) 都是类型为 exn 的值，而 MyOtherException 具有类型 int*int->exn。

通常，我们只是将异常构造函数作为参数传递给 raise，例如 raise MyOtherException(3,9)，但我们也可以更一般地使用它们来创建类型为 exn 的值。 例如，这是一个返回整数列表中最大元素的函数的版本。 与其返回一个选项或引发特定的异常（如 List.Empty，如果调用时传入 []），它接受一个类型为 exn 的参数并引发它。因此，调用者可以传入其选择的异常。 （类型检查器可以推断出 ex 必须具有类型 exn，因为这是 raise 对其参数期望的类型。）

```
fun maxlist (xs,ex) =
    case xs of
        [] => raise ex
      | x::[] => x
      | x::xs' => Int.max(x,maxlist(xs',ex))
```

注意调用maxlist([3,4,0],List.Empty)不会引发异常；这个调用将一个异常值传递给函数，然后函数不会引发异常。

与异常相关的另一个特性是处理（也称为捕获）它们。 为此，ML有处理表达式，它们看起来像e1 handle p => e2其中 e1和 e2是表达式， p是匹配异常的模式

。 语义是评估 e1并将结果作为答案。 但是，如果 e1引发与 p匹配的异常，则评估 e2并将其作为整个表达式的答案。 如果 e1引发与 p不匹配的异常，则整个处理表达式也引发该异常。 类似地，如果 e2引发异常，则整个表达式也引发异常。

与case表达式一样，处理表达式也可以有多个分支，每个分支都有一个模式和表达式，语法上用 |分隔。

尾递归和累加器

这个主题涉及新的编程习惯，但没有新的语言结构。 它定义了尾递归，描述了它与在函数式语言（如ML）中编写高效递归函数的关系，并介绍了如何使用累加器作为一种使某些函数尾递归的技术。

为了理解尾递归和累加器，考虑以下用于对列表元素求和的函数：

```
fun sum1 xs =
    case xs of
        [] => 0
      | i::xs' => i + sum1 xs'


fun sum2 xs =
    let fun f (xs,acc) =
            case xs of
                [] => acc
              | i::xs' => f(xs',i+acc)
    in
        f(xs,0)
    end
```

这两个函数计算相同的结果，但是sum2更复杂，使用了一个局部辅助函数，它接受一个额外的参数，称为acc（累加器）。在f的基本情况中，我们返回acc，而外部调用传递的值为0，与sum1的基本情况中使用的值相同。 这种模式很常见：非累加器风格中的基本情况变成了初始累加器，而累加器风格中的基本情况只返回累加器。

为什么在明显更复杂的情况下可能更喜欢 sum2？ 为了回答这个问题，我们需要了解一点关于函数调用的实现方式。 从概念上讲，有一个调用栈，它是一个栈（具有推入和弹出操作的数据结构），每个已经开始但尚未完成的函数调用都有一个元素。 每个元素存储诸如局部变量的值以及尚未计算的函数的哪个部分等信息。当一个函数体的求值调用另一个函数时，会在调用栈上推入一个新元素，并在被调用的函数完成时弹出。

因此，对于 sum1，每个递归调用 sum1都会有一个调用栈元素（有时只称为“堆栈帧”），即堆栈的大小将与列表一样大。 这是必要的，因为在每个堆栈帧弹出后，调用者必须“执行剩余的部分”-即将 i添加到递归结果并返回。

根据目前的描述，sum2并没有更好：sum2调用了f，然后对每个列表元素进行了一次递归调用。 然而，当f对f进行递归调用时，调用者在被调用者返回后没有其他事情可做，除了返回被调用者的结果。 这种情况被称为尾调用（让我们不要试图弄清楚为什么叫这个名字），像ML这样的函数式语言通常承诺一种重要的优化：当

如果一个调用是尾调用，调用者的栈帧会在调用之前弹出，被调用者的栈帧会替换掉
调用者的栈帧。 这是有道理的：调用者只是要返回被调用者的结果。 因此，对于sum2的调用
to从不使用超过1个栈帧。

为什么函数式语言的实现会包括这种优化？ 通过这样做，递归有时可以像while循环一样高效，而且不会使调用栈变大。 “有时候”指的就是尾调用，你作为程序员可以通过查看代码并确定哪些调用是尾调用来进行推理。

尾调用不需要调用相同的函数（f可以调用 g），因此它们比总是必须“调用”相同循环的while循环更灵活。 使用累加器是将递归函数转换为“尾递归函数”（所有递归调用都是尾调用）的常见方法，但并不总是如此。 例如，处理树（而不是列表）的函数通常具有与树的深度一样大的调用堆栈，但这在任何语言中都是正确的：while循环对于处理树不是很有用。

尾递归的更多示例

尾递归在处理列表的函数中很常见，但这个概念更加通用。例如，这里有两个阶乘函数的实现，第二个使用了一个尾递归的辅助函数，因此它只需要一个小的常量量的调用堆栈空间：

```
fun fact1 n = if n=0 then 1 else n * fact1(n-1)

fun fact2 n =
    let fun aux(n,acc) = if n=0 then acc else aux(n-1,acc*n)
    in
        aux(n,1)
    end
```

值得注意的是，fact1 4和fact2 4产生相同的答案，尽管前者执行
$4*(3*(2*(1*1)))$和后者执行$(((1*4)*3)*2)*1$。 我们依赖于乘法是结合的事实（$a*(b*c) = (a*b)*c$），并且乘以1是恒等函数（$1*x = x*1 = x$）。 早期的求和示例对加法做出了类似的假设。 一般来说，将非尾递归函数转换为尾递归函数通常需要结合性，但许多函数都是结合的。

一个更有趣的例子是这个低效的反转列表的函数：

```
fun rev1 lst =
   case lst of
       [] => []
     | x::xs => (rev1 xs) @ [x]
```

我们可以立即认识到这个函数不是尾递归的，因为在递归调用之后，它仍然需要将结果附加到保存列表头部的单元素列表上。 虽然这是递归反转列表的最自然的方式，但低效性不仅仅是由于创建一个深度等于参数长度的调用栈，我们将其称为n。 更糟糕的问题是执行的总工作量与n的平方成正比，即这是一个二次算法。 原因是连接两个列表的时间与第一个列表的长度成正比：它必须遍历第一个列表 - 参见我们之前讨论的append的实现。 在所有对rev1的递归调用中，我们使用长度为n-1，n-^{2}，...，1的第一个参数调用@，而1到n-1的整数之和是n * (n-1) / ^{2}。

正如你在数据结构和算法课程中学到的那样，对于足够大的n，像这样的二次算法比线性算法要慢得多。 话虽如此，如果你预计n始终很小，那么可能值得考虑程序员的时间，并坚持使用简单的递归算法。 否则，幸运的是，使用累加器习惯用法可以得到一个几乎和简单线性算法一样简单的算法。

```
fun rev2 lst =
    let fun aux(lst,acc) =
            case lst of
                [] => acc
              | x::xs => aux(xs, x::acc)
    in
        aux(lst,[])
    end
```

关键的区别在于（1）尾递归和（2）我们对每个递归调用只做了固定数量的工作，因为 ::不需要遍历其任何参数。

尾位置的精确定义

虽然大多数人依靠直觉来判断“哪些调用是尾调用”，但我们可以通过递归地定义尾位置并说一个调用在尾位置上时它是尾调用来更加精确。 定义中有每种表达式的一部分；这里是几个部分：

- 在`fun f(x) = e`, `e`在尾部位置。
- 如果一个表达式不在尾部位置，那么它的子表达式也不在尾部位置。
- 如果`if e1 then e2 else e3`在尾部位置，那么 `e2`和 `e3`也在尾部位置（但不包括 `e1`）。 （Case-表达式类似。）
- 如果`let b1 ... bn in e end`在尾部位置，那么 `e`也在尾部位置（但绑定中的表达式不在尾部位置）。
- 函数调用参数不在尾部位置。
- ...

CSE341：程序设计语言 2016年春季
第3单元总结

标准描述：这个总结大致涵盖了课堂和复习部分的内容。 当回顾材料时，以叙述方式阅读材料并将整个单元的材料放在一个文档中可能会有所帮助。 请报告这些笔记中的错误，甚至是打字错误。 这个总结不能完全替代上课、阅读相关代码等。

目录

介绍和一些术语

本单元重点介绍一级函数和函数闭包。 通过“一级”我们指的是函数可以在计算、传递、存储等方面与其他值一样。例如，我们可以将它们传递给函数，从函数中返回它们，将它们放入对中，使它们成为数据的一部分，等等。 “函数闭包”指的是使用在其外部定义的变量的函数，这使得一级函数更加强大，我们将在开始使用不使用此功能的更简单的一级函数后看到。 术语高阶函数只是指接受或返回其他函数的函数。

诸如“一等函数”、“函数闭包”和“高阶函数”等术语经常被混淆或被视为同义词。 因为世界上很多人对这些术语不够谨慎，所以我们也不会过于担心它们。 但是“一等函数”的概念和“函数闭包”的概念确实是不同的概念，我们经常一起使用它们来编写优雅、可重用的代码。 因此，我们将推迟闭包的概念，以便将其作为一个独立的概念引入。

还有一个更一般的术语，即“函数式编程”。 这个术语也经常被不准确地用来指代几个不同的概念。 最重要和最常见的两个概念是：

- 在大多数或所有情况下不使用可变数据：到目前为止，我们在课程中一直避免使用变异，并且在很大程度上将继续如此。
- 将函数作为值使用，这就是本单元所讨论的内容

还有其他一些与函数式编程相关的事物：

- 一种鼓励递归和递归数据结构的编程风格
- 使用更接近传统数学函数定义的语法或风格进行编程
- 任何不是面向对象编程的东西（这个真的是不正确的）
- 使用与懒惰相关的某种编程习惯，这是一种我们稍后会简要学习的编程构造/习惯的技术术语

一个明显相关的问题是“什么使得一种编程语言成为函数式语言？”你的讲师得出的结论是这不是一个有确切答案的问题，而且几乎没有意义作为一个问题。 但是可以说，函数式语言是一种在函数式风格（如上所述）中编写更方便、更自然、更常见的语言。 至少，你需要对不可变数据、一级函数和函数闭包有良好的支持。 越来越多的新语言提供了这样的支持，同时也为其他风格（如面向对象编程）提供了良好的支持，我们将在课程末尾学习一些。

将函数作为参数

最常见的使用第一类函数的方式是将它们作为参数传递给其他函数，因此我们首先解释这种用法。

下面是一个接受另一个函数作为参数的函数的第一个示例：

```
fun n_times (f,n,x) =
    if n=0
    then x
    else f (n_times(f,n-1,x))
```

我们可以通过最后一行调用带有参数的f来判断参数f是一个函数。 n_times的作用是计算f(f(...(f(x)))，其中对f的调用次数为n。 这是一个非常有用的辅助函数。 例如，下面是它的三个不同用法：

```
fun double x = x+x
val x1 = n_times(double,4,7) (* 答案：112 *)
```

```
fun increment x = x+1
val x2 = n_times(increment,4,7) (* 答案：11 *)

val x3 = n_times(tl,2,[4,8,12,16]) (* 答案：[12,16] *)
```

像任何辅助函数一样， n_times让我们抽象出多个计算的共同部分，这样我们可以通过传入不同的参数以不同的方式重用一些代码。 主要的创新之处在于将其中一个参数设置为函数，这是一种强大而灵活的编程习惯。 这也是完全合理的- 我们在这里没有引入任何新的语言结构，只是在使用我们已经知道的方式中的一些你可能没有想到的方式。

一旦我们定义了这样的抽象，我们可以找到它们的其他用途。 例如，即使我们的程序今天不需要将任何值乘以三次，也许明天会需要，在这种情况下，我们只需定义函数triple_n_times使用 n_times：

```
fun triple x = 3*x

fun triple_n_times (n,x) = n_times(triple,n,x)
```

多态类型和函数作为参数

现在让我们考虑 n_times的类型，它是('a -> 'a) * int * 'a -> 'a。 起初，考虑类型(int -> int) * int * int -> int可能更简单，这是 n_times在上面用于 x1和 x2时的情况：它接受3个参数，其中第一个参数本身是一个接受并返回 int的函数。 类似地，对于x3，我们使用 n_times，就好像它具有(int list -> int list) * int * int list -> int list的类型。 但是，选择这两种类型之一作为 n_times的类型会使它变得不太有用，因为只有一些我们的示例使用会通过类型检查。 类型('a -> 'a) * int * 'a -> 'a表示第三个参数和结果可以是任何类型，但它们必须是相同的类型，就像第一个参数的参数和返回类型一样。

当类型可以是任何类型，并且不必与其他类型相同时，我们使用不同的字母（'b'，'c'，等等）

这被称为参数多态性，或者更常见的是泛型类型。 它允许函数接受任何类型的参数。 这是与一级函数无关的一个单独问题：

- 有些函数接受函数并且没有多态类型
- 有些具有多态类型的函数不接受函数。

然而，我们的许多一级函数示例将具有多态类型。 这是一件好事，因为它使我们的代码更可重用。

如果没有参数多态性，我们将不得不为列表的每种可能元素类型重新定义列表。 相反，我们可以拥有适用于任何类型列表的函数，例如 length，它的类型是'a list -> int即使它不使用任何函数参数。 相反，这是一个不具有多态性的高阶函数：它的类型是(int->int) * int -> int：[1]

```
fun times_until_zero (f,x) =
    如果 x = 0 则返回 0，否则返回 1 + times_until_zero(f, f x)
```

[1]最好使用累加器将此函数改为尾递归。

匿名函数

没有理由像 triple这样的函数被传递给像 n_times这样的另一个函数时需要在顶层定义。 通常情况下，如果这些函数只在局部范围内使用，最好在局部定义它们。
所以我们可以这样写：

```
fun triple_n_times (n,x) =
  let fun triple x = 3*x in n_times(triple,n,x) end
```

实际上，我们可以给 triple函数一个更小的作用域：我们只需要它作为
n_times的第一个参数，所以我们可以在那里使用一个 let 表达式来求值 triple 函数：

```
fun triple_n_times (n,x) = n_times((let fun triple y = 3*y in triple end), n, x)
```

请注意，在这个例子中，这实际上是一种不好的风格，我们需要在let表达式中使用“return” triple因为，像往常一样，let表达式产生在 in和 end之间的表达式的结果。 在这种情况下，我们只需在环境中查找 trip le，并且得到的函数就是我们作为第一个参数传递给 n_times的值。

ML有一种更简洁的方式，在你使用它们的地方定义函数，就像这个最终的、最好的版本中一样：

```
fun triple_n_times (n,x) = n_times((fn y => 3*y), n, x)
```

这段代码定义了一个匿名函数fn y => 3*y。 它是一个接受参数 y并具有
体 3*y的函数。 fn是一个关键字， => （不是 =）也是语法的一部分。 我们从未给这个函数一个
名字（它是匿名的，明白吗？），这很方便，因为我们不需要一个。我们只是想把一个
函数传递给 n_times，在 n_times的主体中，这个函数被绑定到 f。

通常使用匿名函数作为其他函数的参数。 此外，你可以将匿名函数放在任何可以放置表达式的地方 - 它只是一个值，即函数本身。 唯一不能使用匿名函数的事情是递归，因为你没有名称来进行递归调用。 在这种情况下，你需要像以前一样使用 fun绑定，并且 fun绑定必须在let表达式或顶层中。

对于非递归函数，你可以使用带有 val绑定的匿名函数，而不是 fun绑定。
例如，这两个绑定完全相同：

```
fun increment x = x + 1
val increment = fn x => x+1
```

它们都将 increment绑定到一个返回其参数加1的函数值。 因此，函数绑定几乎是语法糖，但它们支持递归，这是必不可少的。

不必要的函数包装

虽然匿名函数非常方便，但有一种不好的习惯是没有任何好的理由使用它们。考虑一下：

```
fun nth_tail_poor (n,x) = n_times((fn y => tl y), n, x)
```

什么是 fn y => tl y？ 它是一个返回其参数列表尾部的函数。 但是已经有一个绑定到执行完全相同操作的函数的变量了： tl！一般来说，没有理由写fn x => f x当我们可以直接使用 f。 这类似于初学者习惯于写if x then true else false而不是 x。只需这样做：

```
fun nth_tail (n,x) = n_times(tl, n, x)
```

映射和过滤

现在我们考虑一个非常有用的高阶函数：

```
fun map (f,xs) =
    情况 xs of
        [] => []
      | x::xs' => (f x)::(map(f,xs'))
```

map函数接受一个列表和一个函数 f，并通过将 f应用于列表的每个元素来生成一个新列表。 这里有两个示例用法：

```
val x1 = map (increment, [4,8,12,16]) (* 答案：[5,9,13,17] *)
val x2 = map (hd, [[1,2],[3,4],[5,6,7]]) (* 答案：[1,3,5] *)
```

map的类型很有启发性：('a -> 'b) * 'a list -> 'b list。 你可以传递任何类型的列表给map，但是函数 f的参数类型必须是列表的元素类型（它们都是 'a）。 但是函数 f的返回类型可以是不同的类型 'b。 生成的列表是一个'b list。 对于 x1， 'a和 'b都被实例化为 int。 对于 x2， 'a是int list， 'b是 int。

ML标准库提供了一个非常类似的函数 List.map，但它是以柯里化的形式定义的，这是我们将在本单元中讨论的一个主题。

尽管我们的特定示例很简单，但map的定义和使用是一个非常重要的习语。
我们本可以很容易地编写一个递归函数，对整数列表的所有元素进行递增，但我们将工作分为两部分：map的实现者知道如何遍历递归数据结构，这种情况下是一个列表。 map的客户端知道如何处理数据，在这种情况下是递增每个数字。 你可以想象这两个任务中的任何一个——遍历一个复杂的数据结构或为每个部分进行一些计算——都可能非常复杂，并且最好由不同的开发人员完成，而不对另一个任务做出假设。 这正是将map作为一个接受函数的辅助函数来编写的原因。

这是第二个非常有用的高阶函数，用于列表。 它接受一个类型为'a -> bool的函数和一个'a list，并返回一个只包含满足函数返回true的输入列表元素的'a list：

```
fun filter (f,xs) =
    情况 xs of
        [] => []
      | x::xs' => 如果 f x
                  那么 x::(filter (f,xs'))
                  否则 filter (f,xs')
```

这是一个示例用法，假设列表元素是具有类型 int的二元组；它返回第二个组件为偶数的列表元素：

```
fun get_all_even_snd xs = 过滤器((fn (_,v) => v mod 2 = 0), xs)
```

（注意我们如何使用模式作为匿名函数的参数。）

返回函数

函数也可以返回函数。这是一个例子：

```
fun double_or_triple f =
    如果 f 7
    那么 fn x => 2*x
    否则 fn x => 3*x
```

double_or_triple的类型是(int -> bool) -> (int -> int)：if-测试使 f的类型清晰而且通常情况下，if的两个分支必须具有相同的类型，这种情况下是 int->int。 然而，ML会将类型打印为(int -> bool) -> int -> int，这是一样的。 括号是不必要的，因为 ->“从右边关联”，即t1 -> t2 -> t3 -> t4等同于t1 -> (t2 -> (t3 -> t4))。

不仅仅适用于数字和列表

因为ML程序经常使用列表，你可能会忘记高阶函数对于其他数据结构也很有用。 我们的一些最初的例子只使用了整数。 但是高阶函数也非常适用于我们自己的数据结构。 在这里，我们使用 is_even函数来检查算术表达式中的所有常量是否都是偶数。 我们可以很容易地重用true_of_all_constants来检查任何其他属性。

```
数据类型 exp = 常量 of int | 取反 of exp | 加 of exp * exp | 乘 of exp * exp

fun is_even v =
    (v mod 2 = 0)

fun true_of_all_constants(f,e) =
    情况 e of
            常数 i       => f i
      | 否定 e1          => true_of_all_constants(f,e1)
      | 加(e1,e2)        => true_of_all_constants(f,e1) 并且 true_of_all_constants(f,e2)
      | 乘(e1,e2) => true_of_all_constants(f,e1) 并且 true_of_all_constants(f,e2)

fun all_even e = true_of_all_constants(is_even,e)
```

词法作用域

到目前为止，我们传递给其他函数或从其他函数返回的函数都是闭包：函数体只使用函数的参数和任何局部定义的变量。 但我们知道函数可以做更多的事情：它们可以使用任何在作用域内的绑定。 与高阶函数结合使用，这种技术非常强大，因此学习使用这种技术的有效习惯非常重要。但首先需要

更加重要的是要正确理解语义。 这可能是整个课程中最微妙和最重要的概念，所以要慢慢来，仔细阅读。

函数的主体在定义函数的环境中进行评估，而不是调用函数的环境中。这里有一个非常简单的例子来演示差异：

```
val x = 1
fun f y = x + y
val x = 2
val y = 3
val z = f (x+y)
```

在这个例子中， f被绑定到一个接受参数 y的函数。 它的主体还在 f被定义的环境中查找 x。 因此，这个函数总是增加它的参数，因为定义时的环境将 x映射到1。 后来，我们有一个不同的环境，其中 f映射到这个函数， x映射到2， y映射到3，然后我们调用f x。 以下是评估的步骤：

- 查找 f 以获取先前描述的函数。
- 通过查找 x和 y在当前环境中，计算参数 x+y的值，得到5。
- 使用参数5调用函数，这意味着在 "旧 "环境中计算函数体 x+y
  其中 x映射到1， y映射到5。因此结果为6。

注意参数在当前环境中计算（得到5），但函数体在"旧"环境中计算。 我们将在下面讨论为什么这种语义是可取的，但首先我们更准确地定义这种语义，并通过使用高阶函数的其他愚蠢示例来理解语义。

这种语义被称为词法作用域。 另一种较差的语义是使用当前环境（在上面的示例中会产生7），被称为动态作用域。

环境和闭包

我们已经说过函数是值，但我们没有明确说明这个值到底是什么。我们现在解释一下，函数值有两个部分，函数的代码（显然）和创建函数时的环境。 这两个部分确实形成了一个"对"，但我们在引号中加上了"对"这个词，因为它不是ML中的对，只是一个有两个部分的东西。 你不能单独访问"对"的各个部分；你只能调用函数。 这个调用使用了两个部分，因为它使用环境部分来评估代码部分。

这个"对"被称为函数闭包或者只是闭包。 原因是，虽然代码本身可以有自由变量（在代码内部没有绑定的变量，因此需要由某个外部环境绑定），但闭包携带了一个环境，提供了所有这些绑定。 因此，闭包整体上是"封闭"的-它拥有了产生函数结果所需的一切，给定一个函数参数。

在上面的例子中，绑定fun f y = x + y绑定 f到一个闭包。 代码部分是函数
fn y => x + y环境部分将 x映射到1。 因此，对这个闭包的任何调用都将返回 y+1。

（愚蠢的）包含高阶函数的例子

当我们有高阶函数时，词法作用域和闭包变得更有趣，但已经描述的语义将引导我们得到正确的答案。

例子1：

```
val x = 1
fun f y =
    让
        val x = y+1
    在
        fn z => x + y    + z
    结束
val x = 3
val g = f 4
val y = 5
val z = g 6
```

在这里， f绑定到一个闭包，其中环境部分将 x映射到1。 因此，当我们稍后评估f 4时，我们在一个环境中评估let val x = y + 1 in fn z => x + y + z end其中 x映射到1扩展到映射 y到4。 但是由于let绑定，我们遮蔽了 x，所以我们在一个环境中评估fn z => x + y + z其中 x映射到 5和 y映射到4。 我们如何评估类似fn z => x + y + z的函数?
我们使用当前环境创建一个闭包。 因此，f 4 返回一个闭包，当调用时，无论调用点的环境如何，都会将其参数加上9。 因此，在示例的最后一行中， z将绑定到15。

示例2：

```
fun f g =
    让
        val x = 3
    在
        g 2
    结束
val x = 4
fun h y = x + y
val z = f h
```

在这个示例中， f绑定到一个闭包，它接受另一个函数 g作为参数，并返回g 2的结果。 绑定到 h的闭包始终将其参数加上4，因为参数是 y，主体是 x+y，而函数是在一个环境中定义的，其中 x映射到4。 因此，在最后一行中， z将绑定到6。 绑定val x = 3是完全无关紧要的：调用g 2通过查找 g以获取传入的闭包，然后使用该闭包和其环境（其中 x映射到4）以2作为参数进行评估。

为什么词法作用域

虽然词法作用域和高阶函数需要一些时间来适应，但几十年的经验表明，这种语义是我们想要的。 本节的大部分内容将描述各种广泛使用的强大的习惯用法，并且依赖于词法作用域。

但首先，我们还可以通过展示动态作用域（在其中只有一个当前环境并用它来评估函数体）导致一些根本性问题来激发词法作用域。

首先，假设在上面的示例1中， f的主体被更改为let val q = y+1 in fn z => q + y + z。在词法作用域下，这是可以的：我们可以随时更改局部变量的名称及其使用而不会影响任何内容。 在动态作用域下，现在对g 6的调用将没有意义：我们将尝试查找 q，但在调用点的环境中没有 q。

其次，再次考虑示例1的原始版本，但现在将行val x = 3更改为val x = "hi"。
在词法作用域下，这是可以的：该绑定实际上从未被使用过。 在动态作用域下，对于调用
g 6将查找 x，获取一个字符串，并尝试将其添加，这在一个进行类型检查的程序中是不应该发生的。

类似的问题也出现在示例2中：在这个示例中， f的主体很糟糕：我们有一个本地绑定，但我们从未使用过。在词法作用域下，我们可以将其删除，将主体更改为g 2，并且知道这对程序的其余部分没有影响。 在动态作用域下，它将产生影响。 此外，在词法作用域下，我们知道对于绑定到 h的闭包的任何使用都会将其参数增加4，而不管其他函数如 g如何实现以及它们使用的变量名是什么。这是仅词法作用域提供的关注点分离的关键。

对于程序中的“常规”变量，词法作用域是最好的选择。对于某些习惯用法，动态作用域有一些引人注目的用途，但是很少有语言专门支持这些用法（Racket是例外），而且几乎没有现代语言将动态作用域作为默认选项。 但是你已经看到了一个更像动态作用域而不是词法作用域的特性：异常处理。 当抛出异常时，评估过程必须“查找”应该评估的处理表达式。 这个“查找”是使用动态调用栈完成的，不考虑程序的词法结构。

将闭包传递给迭代器，如过滤器

上面的例子很愚蠢，所以我们需要展示依赖于词法作用域的有用程序。 我们将展示的第一个习惯用法是将函数传递给像map和filter这样的迭代器。 我们之前传递的函数没有使用它们的环境（只使用它们的参数和可能的局部变量），但是能够传递闭包使得高阶函数更加广泛地可用。考虑以下例子：

```
fun filter (f,xs) =
    情况 xs of
        [] => []
      | x::xs' => 如果 f x 那么 x::(filter(f,xs')) 否则 filter(f,xs')

fun allGreaterThanSeven xs = filter (fn x => x > 7, xs)

fun allGreaterThan (xs,n) = filter (fn x => x > n, xs)
```

在这里，allGreaterThanSeven 是“旧闻”-我们传入一个函数，从结果中删除列表中小于等于7的任何数字。但更有可能的是，您想要一个像allGreaterThan这样的函数，将“限制”作为参数，并使用函数fn x => x > n。 请注意，这需要一个闭包和词法范围！当filter的实现调用此函数时，我们需要在定义fn x => x > n的环境中查找n。

这里有两个额外的例子：

```
fun allShorterThan1 (xs,s) = filter (fn x => String.size x < String.size s, xs)
```

```
fun allShorterThan2 (xs,s) =
    let
        val i = String.size s
    in
        filter(fn x => String.size x < i, xs)
    end
```

这两个函数都接受一个字符串列表 xs和一个字符串 s，并返回一个只包含长度小于 s的字符串的列表。 它们都使用闭包，在匿名函数被调用时查找 s或 i。 第二个函数更复杂，但更高效：第一个函数在xs中的每个元素上重新计算String.size s（因为 filter调用其函数参数这么多次，并且每次都计算String.size s）。 第二个函数“预先计算”String.size s并将其绑定到一个变量 i，使其在函数fn x => String.size x < i 中可用。

折叠和更多闭包示例

除了map和filter之外，第三个非常有用的高阶函数是 *fold*，它可以有几种稍微不同的定义，并且也被称为 *reduce*和 *inject*。 这是一个常见的定义：

```
fun fold (f,acc,xs) =
  案例 xs of
    []      => acc
  | x::xs' => fold (f, f(acc,x), xs')
```

fold接受一个“初始答案” acc并使用 f将 acc和列表的第一个元素“组合”起来，将其作为“折叠”列表其余部分的新“初始答案”。 我们可以使用 fold来处理在我们提供一些表达如何组合元素的函数的同时遍历列表的问题。 例如，要对列表中的元素求和 foo，我们可以这样做：

```
fold ((fn (x,y) => x+y), 0, foo)
```

与 map和 filter一样， fold的很多功能来自于客户端传递闭包，这些闭包可以有“私有字段”（以变量绑定的形式）来保存他们想要查询的数据。以下是两个示例。
第一个示例计算某个整数范围中有多少个元素。 第二个示例检查所有元素是否都是长度小于某个其他字符串的长度的字符串。

```
fun numberInRange (xs,lo,hi) =
  fold ((fn (x,y) =>
            x + (if y >= lo andalso y <= hi then 1 else 0)),
         0, xs)

fun areAllShorter (xs,s) =
    let
        val i = String.size s
    in
        fold((fn (x,y) => x andalso String.size y < i), true, xs)
    end
```

这种将递归遍历（fold或 map）与对元素进行的数据处理（传递的闭包）分离的模式是基本的。在我们的例子中，这两个部分都很简单，我们可以直接完成整个

将事物组合在几行简单的代码中。 更一般地说，我们可能有一组非常复杂的数据结构要遍历，或者我们可能有非常复杂的数据处理要做。将这些问题分开处理是很好的，这样可以分别解决编程问题。

另一个闭包习惯用法：组合函数

函数组合

当我们使用大量函数进行编程时，创建仅是其他函数组合的新函数非常有用。 你可能在数学中做过类似的事情，比如组合两个函数。 例如，这是一个完全进行函数组合的函数：

```
fun compose (f,g) = fn x => f (g x)
```

它接受两个函数 f和 g并返回一个将其参数应用于 g并将其作为参数传递给 f的函数。 关键是，代码fn x => f (g x)在其定义的环境中使用 f和 g。 注意 compose的类型被推断为('a -> 'b) * ('c -> 'a) -> 'c -> 'b，这等同于你可能写的：('b -> 'c) * ('a -> 'b) -> ('a -> 'c)，因为这两种类型只是一致地使用不同的类型变量名。

作为一个可爱和方便的库函数，ML库定义了中缀运算符 o作为函数组合，就像数学中一样。所以不用写：

```
fun sqrt_of_abs i = Math.sqrt(Real.fromInt (abs i))
```

你可以写：

```
fun sqrt_of_abs i = (Math.sqrt o Real.fromInt o abs) i
```

但是这个第二个版本更清楚地表明我们可以使用函数组合来创建一个函数，然后将其绑定到一个变量上，就像这个第三个版本中那样：

```
val sqrt_of_abs = Math.sqrt o Real.fromInt o abs
```

虽然这三个版本都相当易读，但第一个版本并不立即向读者表明sqrt_of_abs只是其他函数的组合。

管道运算符

在函数式编程中，将其他函数组合起来创建更大的函数非常常见，因此定义方便的语法是有意义的。虽然上面的第三个版本很简洁，但它和数学中的函数组合一样，具有从右到左的计算顺序：“取绝对值，将其转换为实数，然后计算平方根”可能比“取绝对值的实数转换的平方根”更容易理解。

我们还可以定义方便的从左到右的语法。 让我们首先定义一个中缀运算符，使我们能够将函数放在我们调用它的参数的右边：

```
中缀 |>（* 告诉解析器 |> 是一个出现在其两个参数之间的函数 *）
fun x |> f = f x
```

现在我们可以写：

```
fun sqrt_of_abs i = i |> abs |> Real.fromInt |> Math.sqrt
```

这个运算符，在F#编程中被称为管道运算符，非常受欢迎。（F#是运行在.Net上并与其他.Net语言编写的库良好交互的ML方言。）正如我们所见，管道运算符的语义并不复杂。

另一个闭包习惯用法：柯里化和部分应用

我们考虑的下一个习惯用法在一般情况下非常方便，并且在定义和使用高阶函数（如map，filter和fold）时经常使用。 我们已经看到，在ML中，每个函数只接受一个参数，所以你必须使用一个习惯用法来获得多个参数的效果。 我们之前的方法将一个元组作为一个参数传递，所以元组的每个部分在概念上都是多个参数之一。
另一种更聪明、更方便的方法是让一个函数接受第一个概念参数，并返回另一个函数，该函数接受第二个概念参数，依此类推。词法作用域对于这种技术的正确工作是必不可少的。

这种技术被称为柯里化，以一个名叫Haskell Curry的逻辑学家命名，他研究了相关的思想（所以如果你不知道这个，那么柯里化这个术语就没有太多意义）。

定义和使用柯里化函数

这是一个使用柯里化的“三个参数”函数的例子：

```
val sorted3 = fn x => fn y => fn z => z >= y andalso y >= x
```

如果我们调用sorted3 4，我们将得到一个闭包，它的环境中有x。 如果我们然后用5调用这个闭包，我们将得到一个闭包，它的环境中有x和y。 如果我们然后用6调用这个闭包，我们将得到true，因为6大于5且5大于4。 这就是闭包的工作原理。

所以((sorted3 4) 5) 6计算出我们想要的结果，并且感觉非常接近调用 sorted3带有3个参数的情况。 更好的是，括号是可选的，所以我们可以写成sorted3 4 5 6，这实际上比我们以前的元组方法更少字符。

```
fun sorted3_tupled (x,y,z) = z >= y andalso y >= x
val someClient = sorted3_tupled(4,5,6)
```

一般来说，语法e1 e2 e3 e4隐式地表示嵌套的函数调用(((e1 e2) e3) e4)这个选择是因为它使得使用柯里化函数非常愉快。

部分应用

尽管我们可能期望大多数客户端使用我们的柯里化 sorted3提供所有3个概念参数，但他们可能提供较少的参数，并在以后使用生成的闭包。 这被称为“部分应用”，因为我们提供了概念参数的子集（更准确地说，是前缀）。 作为一个愚蠢的例子，sorted3 0 0返回一个函数，如果其参数是非负数，则返回true。

部分应用和高阶函数

柯里化对于创建具有迭代器的相似函数特别方便。 例如，这是一个列表的柯里化版本的fold函数：

```
fun fold f = fn acc => fn xs =>
  case xs of
    []      => acc
  | x::xs' => fold f (f(acc,x)) xs'
```

现在我们可以使用这个fold来定义一个求和列表元素的函数，如下所示：

```
fun sum1 xs = fold (fn (x,y) => x+y) 0 xs
```

但与仅使用部分应用相比，这是不必要的复杂：

```
val sum2 = fold (fn (x,y) => x+y) 0
```

部分应用的方便之处是为什么ML标准库中的许多迭代器使用柯里化作为它们接受的函数的第一个参数。 例如，所有这些函数的类型都使用了柯里化：

```
val List.map = fn : ('a -> 'b) -> 'a list -> 'b list
val List.filter = fn : ('a -> bool) -> 'a list -> 'a list
val List.foldl = fn : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
```

作为一个例子，`List.foldl((fn (x,y) => x+y), 0, [3,4,5])`不能通过类型检查，因为 `List.foldl` 期望一个 `'a * 'b -> 'b` 函数，而不是一个三元组。 正确的调用是 `List.foldl (fn (x,y) => x+y) 0 [3,4,5]` ，它调用 `List.foldl` 使用一个函数，该函数返回一个闭包，依此类推。

有一种语法糖可以用于定义柯里化函数；你可以只用空格分隔概念上的参数，而不是使用匿名函数。 所以我们的fold函数的更好的风格是：

```
fun fold f acc xs =
  情况 xs of
    []      => acc
  | x::xs' => fold f (f(acc,x)) xs'
```

另一个有用的柯里化函数是`List.exists`，在下面的回调示例中使用。 这些库函数很容易自己实现，所以我们应该明白它们并不花哨：

```
fun exists predicate xs =
    情况 xs of
      [] => false
    | x::xs' => 谓词 x 或者存在谓词 xs'
```

柯里化的一般性

虽然柯里化和部分应用在高阶函数中非常好用，但它们在一般情况下也非常好用。它们适用于任何多参数函数，而部分应用也可以非常方便。 在这个例子中，zip和range都是通过柯里化定义的，而countup部分应用了range。 add_numbers函数将列表`[v1,v2,...,vn]`转换为`[(1,v1),(2,v2),...,(n,vn)]`。

```
fun zip xs ys =
    (case (xs,ys) of
        ([],[]) => [])
      | (x::xs',y::ys') => (x,y) :: (zip xs' ys')
      | _ => raise Empty
```

```
fun range i j = if i > j then [] else i :: range (i+1) j

val countup   = range 1

fun add_numbers xs = zip (countup (length xs)) xs
```

将函数组合以柯里化和非柯里化其他函数

有时函数是柯里化的，但参数的顺序不是你想要的部分应用。
或者有时候一个函数是柯里化的，但你希望它使用元组或者反之亦然。 幸运的是，我们之前的组合函数的习惯可以使用一种方法来接受函数，并产生使用另一种方法的函数：

```
fun other_curry1 f = fn x => fn y => f y x
fun other_curry2 f x y = f y x
fun curry f x y = f (x,y)
fun uncurry f (x,y) = f x y
```

查看这些函数的类型可以帮助你理解它们的功能。顺便说一下，这些类型也很有趣，因为如果你将 ->发音为“蕴含”，将 *发音为“与”，所有这些函数的类型都是逻辑重言式。

效率

最后，你可能想知道哪个更快，柯里化还是元组化。 几乎从不重要；它们都按照概念参数的数量进行工作，而这通常非常小。 对于性能关键的软件函数，选择更快的方式可能很重要。在我们使用的ML编译器版本中，元组化恰好更快。 在OCaml、Haskell和F#的广泛使用实现中，柯里化函数更快，因此它们是这些语言中定义多参数函数的标准方式。

值限制

一旦你学会了柯里化和部分应用，你可以尝试使用它来创建一个多态函数。 不幸的是，某些用法，比如这些，在ML中不起作用：

```
val mapSome = List.map SOME （将[v1,v2,...,vn]转换为[SOME v1, SOME v2, ..., SOME vn]）
val pairIt = List.map (fn x => (x,x)) （将[v1,v2,...,vn]转换为[(v1,v1),(v2,v2),...,(vn,vn)]）
```

根据我们目前所学的知识，没有理由认为这不会起作用，特别是因为所有这些函数都起作用：

```
fun mapSome xs = List.map SOME xs
val mapSome = fn xs => List.map SOME xs
val pairIt : int list -> (int * int) list = List.map (fn x => (x,x))
val incrementIt = List.map (fn x => x+1)
```

原因被称为值限制有时很烦人。 它在语言中有很好的原因：没有它，类型检查器可能会允许一些代码破坏类型系统。 这只能发生在使用突变的代码中，而上面的代码没有，但类型检查器不知道这一点。

最简单的方法是忽略这个问题，直到你收到关于值限制的警告/错误。 当
你这样做时，将val绑定转换回像上面第一个示例中有效的绑定。

当我们在下一个单元中学习类型推断时，我们将更详细地讨论值限制。

通过ML引用进行突变

我们现在终于介绍了ML对突变的支持。 在某些情况下，突变是可以接受的。 函数式编程的一个关键方法是仅在“更新某个状态以便所有使用该状态的用户都能看到发生的更改”是模拟计算的自然方式时使用它。 此外，我们希望将突变的特性保持分离，以便我们知道何时没有使用突变。

在ML中，大多数东西真的不能被改变。 相反，你必须创建一个引用，它是一个容器，其内容可以被改变。 你可以使用表达式`ref e`创建一个新的引用（初始内容是评估 `e`的结果）。 你可以使用 `!r`来获取引用 `r`的当前内容（不要与Java或C中的否定混淆），并使用`r := e`来更改引用 `r`的内容。 包含类型为 `t`的值的引用的类型被写作`t ref`。

一个关于引用的好方法是将其视为一个具有一个字段的记录，该字段可以使用 `:=`运算符进行更新。

下面是一个简短的例子：

```
val x = ref 0
val x2 = x （x和x2都指向同一个引用）
val x3 = ref 0
（* val y = x + 1*） （* 错误：x不是一个整数*）
val y = (!x) + 1 （* y是1*）
val _ = x := (!x) + 7 (* 引用x的内容现在是7 *)
val z1 = !x (* z1是7 *)
val z2 = !x2 (* z2也是7 -- 使用变异时，别名很重要 *)
val z3 = !x3 (* z3是0 *)
```

另一个闭包习惯用法：回调函数

我们考虑的下一个常见习惯用法是实现一个库，该库可以检测“事件”何时发生，并通知之前“注册”了对事件感兴趣的客户端。 客户端可以通过提供一个“回调函数”来注册他们的兴趣 —— 当事件发生时，该函数会被调用。 你可能希望使用这种类型的库来处理诸如用户移动鼠标或按键等事件。从网络接口接收数据也是一个例子。 在游戏中，计算机玩家的回合是另一个事件。

这些库的目的是允许多个客户端注册回调函数。 库的实现者不知道客户端在事件发生时需要计算什么，而且客户端可能需要“额外的数据”来进行计算。 因此，库的实现者不应限制每个客户端使用的“额外数据”。 闭包非常适合这个任务，因为函数的类型`t1 -> t2`不指定闭包使用的任何其他变量的类型，所以我们可以将“额外数据”放在闭包的环境中。

如果你在Java的Swing库中使用了“事件监听器”，那么你在面向对象的设置中使用了这种习惯用法。 在Java中，你可以通过定义一个带有额外字段的子类来获得“额外数据”。 这对于一个简单的监听器来说需要非常多的按键，这也是Java语言添加匿名内部类的一个主要原因（这门课程不需要了解，但我们稍后会展示一个例子），这更接近闭包的便利性。

在ML中，我们将使用变异来展示回调习惯用法。 这是合理的，因为我们确实希望在注册回调时“改变世界的状态”——当事件发生时，现在有更多的回调要调用。

我们的例子使用了回调应该在按下键盘上的一个键时被调用的想法。 我们将把一个编码了哪个键的整数传递给回调函数。我们的接口只需要一种注册回调的方式。
（在一个真正的库中，你可能还想要一种取消注册的方式。）

```
val onKeyEvent : (int -> unit) -> unit
```

客户端将传递一个类型为int -> unit的函数，当以 int调用时，它将执行他们想要的任何操作。 要实现这个函数，我们只需使用一个保存回调函数列表的引用。 然后当事件实际发生时，我们假设函数 onEvent被调用，并调用列表中的每个回调函数：

```
val cbs: (int -> unit) 列表引用 = ref []
fun onKeyEvent f = cbs := f::(!cbs)（唯一的"公共"绑定）
fun onEvent i =
   let fun loop fs =
         case fs of
           [] => ()
         | f::fs' => (f i; loop fs')
    in loop (!cbs) end
```

最重要的是， onKeyEvent的类型对回调函数在调用时可以访问的额外数据没有限制。 这里有不同的客户端（对 onKeyEvent的调用），它们在环境中使用不同类型的绑定。 （val _ = e的用法常用于仅为了其副作用而执行表达式，在这种情况下是注册回调函数。）

```
val timesPressed = ref 0
val _ = onKeyEvent (fn _ => timesPressed := (!timesPressed) + 1)

fun printIfPressed i =
    onKeyEvent (fn j => if i=j
                        then print ("你按了 " ^ Int.toString i ^ "")else ())


val _ = printIfPressed 4
val _ = printIfPressed 11
val _ = printIfPressed 23
```

可选：另一个闭包习惯用法：抽象数据类型

我们将考虑的最后一个闭包习惯用法是最花哨和最微妙的。 这不是程序员通常做的事情 - 在现代编程语言中通常有更简单的方法。 它作为一个高级示例包含在内，以演示具有相同环境的闭包记录与面向对象编程中的对象非常相似：函数是方法，环境中的绑定是私有字段和方法。 这里没有新的语言特性，只有词法作用域。 它表明（正确地）函数式编程和面向对象编程比它们看起来更相似（这是我们在课程中稍后会重新讨论的一个主题；也有重要的区别）。

抽象数据类型（ADT）的关键是要求客户端通过一系列函数来使用它，而不是直接访问其私有实现。 由于这种抽象，我们可以在不改变客户端行为的情况下，稍后更改数据类型的实现方式。 在面向对象的语言中，您可以通过定义一个具有所有私有字段（对客户端不可访问）和一些公共方法（与客户端的接口）的类来实现ADT。

在ML中，我们可以通过闭包的记录来做同样的事情；闭包从环境中使用的变量对应于私有字段。 我们可以通过闭包的记录来做同样的事情；闭包从环境中使用的变量对应于私有字段。

举个例子，考虑一个支持创建一个更大的整数集合并查看一个整数是否在集合中的实现。我们的集合是无变异的，也就是说将一个整数添加到集合中会产生一个新的、不同的集合。（我们也可以使用ML的引用来定义一个可变版本。）在ML中，我们可以定义一个描述我们接口的类型：

```
datatype set = S of { insert : int -> set, member : int -> bool, size : unit -> int }
```

粗略地说，一个集合是一个包含三个函数的记录。 写起来会更简单：

```
type set = { insert : int -> set, member : int -> bool, size : unit -> int }
```

但是在ML中这样做是不起作用的，因为类型绑定不能递归。 所以我们不得不处理一点小不便，即在定义集合的函数记录周围加上构造函数S，尽管集合是每个类型，而不是一个类型。 请注意，我们没有使用任何新的类型或特性；我们只是有一个描述具有名为insert，member和size的字段的记录类型，每个字段都包含一个函数。

一旦我们有了一个空集合，我们可以使用它的insert字段创建一个单元素集合，然后使用该集合的insert字段创建一个两个元素的集合，依此类推。所以我们接口需要的唯一其他东西是一个绑定，像这样：

```
val empty_set = ... : 集合
```

在实现这个接口之前，让我们看看客户端如何使用它（许多括号是可选的，
但可能有助于理解代码）：

```
fun use_sets () =
    let val S s1 = empty_set
        val S s2 = (#insert s1) 34
        val S s3 = (#insert s2) 34
        val S s4 = #insert s3 19
    in
        如果 (#member s4) 42
        那么 99
        否则如果 (#member s4) 19
        那么 17 + (#size s3) ()
        否则 0
    结束
```

再次，我们没有使用任何新功能。 #insert s1正在读取一个记录字段，在这种情况下会产生一个我们可以用34来调用的函数。. 如果我们在Java中，可能会写成s1.insert(34)来做类似的事情。 这些 val绑定使用模式匹配来“剥离”类型为 set的值上的 S构造函数。

我们可以用多种方式来定义空集合; 它们都使用闭包的技术来“记住”一个集合有哪些元素。这是其中一种方式：

```
val empty_set =
    let
        fun make_set xs = (* xs是结果中的“私有字段” *)
```

```
            let (* 在结果中包含一个“私有方法” *)
                fun contains i = List.exists (fn j => i=j) xs
            in
                S { insert = fn i => 如果 contains i
                                     那么 make_set xs
                                     否则 make_set (i::xs),
                    member = contains,
                    size   = fn () => length xs
                  }
            结束
    在
        make_set []
    结束
```

所有的花哨都在 make_set中， empty_set只是make_set []返回的记录。 make_set返回的是一个类型为set的值。 它本质上是一个带有三个闭包的记录。 这些闭包可以使用 xs，辅助函数 contains和 make_set。像所有的函数体一样，它们在被调用之前不会执行。

可选：其他语言中的闭包

为了总结我们对函数闭包的研究，我们从ML转向Java（使用泛型和接口）和C（使用函数指针带有显式环境参数）。展示类似的编程模式。
我们不会在这个材料上测试你，你可以跳过它。然而，通过在其他环境中看到类似的思想，它可能帮助你理解闭包，并且它应该帮助你看到一个语言中的核心思想如何影响你在其他语言中解决问题的方式。 也就是说，它可能使你成为一个更好的Java或C程序员。

对于Java和C，我们将“移植”这个ML代码，它定义了我们自己的多态链表类型构造函数和三个多态函数（两个高阶函数）在该类型上。 我们将研究一些在Java或C中编写类似代码的方法，这将帮助我们更好地理解闭包和对象之间的相似之处（对于Java），以及如何使环境显式（对于C）。在ML中，没有理由定义我们自己的类型构造函数，因为已经有了“列表”，但这样做将帮助我们与Java和C版本进行比较。

```
datatype 'a mylist = Cons of 'a * ('a mylist) | Empty

fun map f xs =
    case xs of
        Empty => Empty
      | Cons(x,xs) => Cons(f x, map f xs)

fun filter f xs =
    case xs of
        Empty => Empty
      | Cons(x,xs) => 如果 f x 那么 Cons(x,filter f xs) 否则 filter f xs

fun length xs =
    case xs of
        Empty => 0
```

```
      | Cons(_,xs) => 1 + length xs
```

使用这个库，这里有两个客户端函数。（后者不是特别高效，但展示了对length和filter的简单使用。）

```
val doubleAll = map (fn x => x * 2)
fun countNs (xs, n : int) = length (filter (fn x => x=n) xs)
```

可选：在Java中使用对象和接口实现闭包

Java 8包括对闭包的支持，就像其他大多数主流面向对象语言一样（C#，Scala，Ruby等），但是我们可以考虑如何在没有此支持的情况下在Java中编写类似的代码，因为这已经是必要的近20年了。 虽然我们没有一级函数、柯里化或类型推断，但我们有泛型（Java以前没有）并且我们可以定义只有一个方法的接口，我们可以像函数类型一样使用它们。 话不多说，这是代码的Java类似版本，接下来是对你可能之前没有见过的特性的简要讨论以及我们可以编写代码的其他方式：

```
interface Func<B,A> {
    B m(A x);
}
interface Pred<A> {
    boolean m(A x);
}
class List<T> {
    T       head;
    List<T> tail;
    List(T x, List<T> xs) {
        head = x;
        tail = xs;
    }
    static <A,B> List<B> map(Func<B,A> f, List<A> xs) {
        if(xs==null)
            return null;
        return new List<B>(f.m(xs.head), map(f,xs.tail));
    }
    static <A> List<A> filter(Pred<A> f, List<A> xs) {
        if(xs==null)
            return null;
        if(f.m(xs.head))
            return new List<A>(xs.head, filter(f,xs.tail));
        return filter(f,xs.tail);
    }
    static <A> int length(List<A> xs) {
        int ans = 0;
        while(xs != null) {
            ++ans;
            xs = xs.tail;
        }
        返回 ans;
```

```
    }
}

类 ExampleClients {
    static List<Integer> doubleAll(List<Integer> xs) {
        return List.map((new Func<Integer,Integer>() {
                             public Integer m(Integer x) { return x * 2; }
                         }),
                        xs);
    }
    static int countNs(List<Integer> xs, final int n) {
        return List.length(List.filter((new Pred<Integer>() {
                                            public boolean m(Integer x) { return x==n; }
                                        }),
                                        xs));
    }
}
```

这段代码使用了几种有趣的技术和特性：

- 在 map和 filter中，我们使用了泛型接口代替了（推断的）函数类型'a -> 'b和'a -> bool，这些接口只有一个方法。 实现这些接口的类可以拥有任何需要的字段，这些字段将充当闭包环境的角色。

- 泛型类 List扮演了数据类型绑定的角色。 构造函数按照预期的方式初始化了 head和 tail字段，使用了标准的Java约定，对于空列表使用 null。

- 在Java中，静态方法可以是泛型的，只要类型变量明确地出现在返回类型的左边。 除了语法之外， map和 filter的实现与ML的实现类似，都使用了 Func或 Pred接口中的一个方法作为传递的函数。 对于 length，我们可以使用递归，但选择遵循Java对循环的偏好。

- 如果你从未见过匿名内部类，那么方法 doubleAll和 countNs看起来很奇怪。有点像匿名函数，这种语言特性允许我们创建一个实现接口的对象，而不给该对象的类命名。 相反，我们使用 new来实现接口（适当地实例化类型变量），然后为方法提供定义。 作为内部类，这个定义可以使用封闭对象的字段或者封闭方法的最终局部变量和参数，以更繁琐的语法获得闭包环境的许多便利。 （匿名内部类是为了支持回调和类似的习惯用法而添加到Java中的。）

我们可以用许多不同的方式编写Java代码。特别感兴趣的是：

- 在Java的实现中，尾递归不如循环高效，因此可以合理地偏好基于循环的 map和 filter的实现。 这样做而不翻转中间列表比你想象的更复杂（你需要保留对前一个元素的指针，并编写特殊代码来处理第一个元素），这就是为什么这种类型的程序经常在编程面试中被问到的原因。 递归版本很容易理解，但对于非常长的列表来说是不明智的。

- 更面向对象的方法是将 map、 filter和 length作为实例方法而不是静态方法。 方法签名将会改变为：

```
<B> List<B> map(Func<B,T> f) {...}
List<T> filter(Pred<T> f) {...}
int length() {...}
```

这种方法的缺点是，如果客户端可能有一个空列表，我们必须在使用这些方法的任何地方添加特殊情况。 原因是空列表被表示为 null，使用 null作为调用的接收者会引发NullPointerException异常。 因此，方法 doubleAll和 countNs必须检查它们的参数是否为 null，以避免此类异常。

- 另一种更面向对象的方法是不使用 null表示空列表。 相反，我们将有一个抽象列表类，其中包含两个子类，一个用于空列表，一个用于非空列表。 这种方法更符合具有多个构造函数的数据类型的面向对象方法，使用它可以使之前的实例方法的建议不需要特殊情况。 对于习惯使用 null的程序员来说，这种方法似乎更复杂和更长。

- 匿名内部类只是一种方便的方式。 我们可以定义“普通”类来实现Func<Integer,Integer>和Pred<Integer>并创建实例来传递给 map和 filter。
  对于 countNs的例子，我们的类将有一个用于保存 n的 int字段，并将该字段的值传递给类的构造函数，该构造函数将初始化该字段。

可选：在C中使用显式环境的闭包

C确实有函数，但它们不是闭包。 如果你传递一个函数指针，它只是一个代码指针。
正如我们所学过的，如果一个函数参数只能使用它的参数，高阶函数就没有那么有用。 那么在像C这样的语言中我们能做什么呢？ 我们可以将高阶函数改为以下形式：

- 将环境作为另一个参数显式传递。
- 函数参数也接受一个环境。
- 在调用函数参数时，将环境传递给它。

所以，代替高阶函数看起来像这样：

```
int f(int (*g)(int), list_t xs) { ... g(xs->head) ... }
```

我们将其改为这样：

```
int f(int (*g)(void*,int), void* env, list_t xs) { ... g(env,xs->head) ... }
```

我们使用 void*因为我们希望 f能够处理使用不同类型环境的函数，所以没有更好的选择。 客户端将不得不将 void*从其他兼容类型进行转换。 我们在这里不讨论这些细节。

虽然C代码有很多其他细节，但在map和 filter的定义和使用中明确使用环境是与其他语言版本的关键区别：

```
#include <stdlib.h>
#include <stdint.h>
#include <stdbool.h>
```

```
typedef struct List list_t;
struct List {
  void * head;
  list_t * tail;
}.
list_t * makelist (void * x, list_t * xs) {
  list_t * ans = (list_t *)malloc(sizeof(list_t));
  ans->head = x;
  ans->tail = xs;
  return ans;
}
list_t * map(void* (*f)(void*,void*), void* env, list_t * xs) {
  if(xs==NULL)
    return NULL;
  return makelist(f(env,xs->head), map(f,env,xs->tail));
}
list_t * filter(bool (*f)(void*,void*), void* env, list_t * xs) {
  if(xs==NULL)
    return NULL;
  if(f(env,xs->head))
    return makelist(xs->head, filter(f,env,xs->tail));
  return filter(f,env,xs->tail);
}
int length(list_t* xs) {
  int ans = 0;
  while(xs != NULL) {
    ++ans;
    xs = xs->tail;
  }
  返回 ans;
}
void* doubleInt(void* ignore, void* i) { // type casts to match what map expects
  return (void*)(((intptr_t)i)*2);
}
list_t * doubleAll(list_t * xs) { // assumes list holds intptr_t fields
  return map(doubleInt, NULL, xs);
}
bool isN(void* n, void* i) { // 类型转换以匹配过滤器所需的类型
  return ((intptr_t)n)==((intptr_t)i);
}
int countNs(list_t * xs, intptr_t n) { // 假设列表包含intptr_t字段
  return length(filter(isN, (void*)n, xs));
}
```

与Java一样，使用递归而不是循环更简单，但可能效率较低。另一种选择是定义将代码和环境放在一个值中的结构体，但我们的方法是在每个高阶函数中使用额外的void*参数，这在C代码中更常见。

对于那些对C规范细节感兴趣的人：还要注意上面的客户端代码，特别是doubleInt，isN和countNs函数中的代码，因为假设intptr_t可以被转换为void*并返回，除非该值起始于指针（而不是适合intptr_t的数字）。虽然上述代码是一种常见的方法，但是有可移植版本

要么需要使用指向数字的指针，要么用 intptr_t替换库中的 void*的使用。
后一种方法仍然是一个可重用的库，因为任何指针都可以转换为 intptr_t并返回。

标准库文档

这个主题与本单元的其余部分没有密切关系，但我们需要它一点来完成作业3，它对于任何编程语言都很有用，并展示了一些有用的函数（高阶或非高阶）在ML中预定义。

ML，像许多语言一样，有一个标准库。 这是语言中的程序可以假设始终可用的代码。 代码在标准库中的原因有两个常见且不同的原因：

- 我们需要一个标准库来与“外部世界”进行接口，以提供其他情况下无法实现的功能。 例如打开文件或设置计时器。
- 标准库可以提供常见且有用的函数，以便一次定义所有程序都可以使用相同的函数名称、参数顺序等。例如，连接两个字符串，对列表进行映射等函数。

标准库通常非常庞大，因此期望被教授它们是没有意义的。 你需要熟悉查找文档和开发大致直觉的能力，了解“可能提供什么”和“可能在哪里提供”。因此，在作业3中，我们将让你自己了解更多关于ML标准库中的一些简单函数。

在线文档与大多数现代语言相比非常简陋，但对我们的需求来说完全足够。只需访问：

http://www.standardml.org/Basis/manpages.html

这些函数是使用ML的模块系统进行组织的，我们将在下一单元中学习其基础知识。
例如，有关字符的有用函数位于结构 Char中。 要使用结构Bar中的函数 foo，您需要写 Bar.foo，这正是我们一直在使用类似 List.map的函数的方式。 一个小问题是，字符串结构的函数在签名 STRING下进行了文档化。 签名基本上是结构的类型，我们将在以后学习。 某些库函数被认为非常有用，因此它们不在一个结构中，比如 hd。 这些绑定在http://www.standardml.org/Basis/top-level-chapter.html中有描述。

精确和完整的代码库文档是无可替代的，但有时在编程过程中查找完整的文档可能会不方便，只需要一个快速提醒。 例如，很容易忘记参数的顺序或函数是柯里化还是元组化。 通常，您可以使用REPL快速获取所需的信息。 毕竟，如果您输入类似于 List.map的函数，它会评估此表达式并返回其类型。 如果您不记得函数的名称，甚至可以猜测函数的名称。 如果您猜错了，您只会收到一个未定义变量的消息。 最后，使用我们将要学习的功能之外的功能，您可以让REPL打印出结构提供的所有绑定。 只需执行以下操作：

```
结构X = List; (* List是我们想了解的结构 *)
结构X: LIST (* 这是REPL返回的内容 *)
签名 X = 列表; (* 写列表，因为它是在前一行的 : 后面 *)
```

因为在 REPL 中查找东西非常方便，一些其他语言的 REPL 进一步提供了特殊命令来打印与函数或库相关的文档。

CSE341：程序设计语言 2016年春季
第4单元总结

标准描述：这个总结大致涵盖了课堂和复习部分的内容。 当回顾材料时，以叙述方式阅读材料并将整个单元的材料放在一个文档中可能会有所帮助。 请报告这些笔记中的错误，甚至是打字错误。 这个总结不能完全替代上课、阅读相关代码等。

目录

命名空间管理的模块

我们首先展示了如何使用ML模块将绑定分离到不同的命名空间中。 然后我们在这个基础上讨论了使用模块隐藏绑定和类型这个更有趣和重要的主题。

为了学习ML、模式匹配和函数式编程的基础知识，我们编写了一些只包含一系列绑定的小程序。 对于更大的程序，我们希望用更多的结构来组织我们的代码。
在ML中，我们可以使用结构来定义包含一系列绑定的模块。 最简单的情况下，你可以写成$structure\ Name = struct\ bindings\ \ end$其中 `Name`是你的结构的名称（你可以选择任何名称；大写是一种约定），bindings 是任何绑定的列表，包括值、函数、异常、数据类型和类型同义词。 在结构内部，你可以像我们一直在“顶层”（即在任何模块之外）使用之前的绑定一样。 在结构之外，你可以引用一个绑定 b

通过写入名称来名称。 我们已经在使用这种符号来使用函数，如 List.foldl; 现在
你知道如何定义自己的结构。

虽然我们在示例中不这样做，但是你可以将结构嵌套在其他结构中以创建树形层次结构。 但是在ML中，模块不是表达式：你不能在函数内部定义它们，存储它们在元组中，将它们作为参数传递等。

如果在某个作用域中你正在使用另一个结构的许多绑定，那么多次写入SomeLongStructureName.foo可能会不方便。 当然，你可以使用val绑定来避免这种情况，例如，val foo = SomeLongStructureName.foo，但是如果我们从结构中使用许多不同的绑定（我们需要为每个绑定使用一个新变量）或者在模式中使用构造函数名称时，这种技术是无效的。 因此，ML允许你写入open SomeLongStructureName，这提供了对模块中任何在模块签名中提到的绑定的“直接”访问（你只需写入 foo）。

一个打开的范围是封闭结构的其余部分（或者在顶层的整个程序）。

一个常见的用途 open是在模块外部编写简洁的测试代码。 其他用途of open经常被人们不赞同，因为它可能引入意外的遮蔽，尤其是因为不同模块可能重用绑定名称。 例如，一个列表模块和一个树模块都可能有名为 map的函数。

签名

到目前为止，结构只提供了命名空间管理，一种避免程序中不同部分的不同绑定相互遮蔽的方法。 命名空间管理非常有用，但并不是很有趣。 更有趣的是给结构提供签名，这是模块的类型。 它们让我们提供严格的接口，外部代码必须遵守。 ML有几种不同的方法来实现这一点，具有微妙的语法和语义；我们只展示一种为模块编写显式签名的方法。 这里是一个示例签名定义和结构定义，它表示结构 MyMathLib必须具有签名 MATHLIB：

```
签名 MATHLIB =
开始
val fact : int -> int
val half_pi : real
val doubler : int -> int
结束

结构 MyMathLib :> MATHLIB =
结构
fun fact x =
    if x=0
    then 1
    else x * fact (x - 1)

val half_pi = Math.pi / 2.0

fun doubler y = y + y
end
```

由于:> MATHLIB，结构 MyMathLib只有在实际提供了签名 MATHLIB所声明的一切并且类型正确时才能通过类型检查。 签名还可以包含数据类型、异常，

和类型绑定。 因为我们在编译 MyMathLib时检查签名，所以我们可以在检查使用 MyMathLib的任何代码时使用这些信息。 换句话说，我们可以假设客户端假设签名是正确的。

隐藏事物

在学习如何使用ML模块隐藏实现细节之前，让我们记住，将接口与实现分离可能是构建正确、健壮、可重用程序的最重要策略。 此外，我们已经可以使用函数以各种方式隐藏实现。

例如，这3个函数都会将它们的参数加倍，而客户端（即调用者）无法知道我们是否用不同的函数替换了其中一个：

```
fun double1 x = x + x
fun double2 x = x * 2
val y = 2
fun double3 x = x * y
```

我们用于隐藏实现的另一个特性是在其他函数内部局部定义函数。 我们可以稍后更改、删除或添加局部定义的函数，而不会影响任何其他代码。 从工程角度来看，这是一个关键的关注点分离。 我可以改进函数的实现，并确保我不会破坏任何客户端。 相反，客户端无法破坏上述函数的工作方式。

但是，如果您想要有两个顶级函数，其他模块中的代码可以使用这两个函数，并且这两个函数都使用相同的隐藏函数怎么办？ 有办法可以做到这一点（例如，创建一个函数记录），但是拥有一些对模块“私有”的顶级函数会很方便。 在ML中，没有像其他语言中那样的“private”关键字。 相反，您可以使用仅提及较少内容的签名：在签名中未明确指定的任何内容都无法从外部使用。 例如，如果我们将上述签名更改为：

```
签名 MATHLIB =
开始
val fact : int -> int
val half_pi : real
end
```

那么客户端代码无法调用MyMathLib.doubler。 绑定只是不在范围内，所以不能使用它进行类型检查。 一般来说，我们可以按照自己的喜好实现模块，只有在签名中明确列出的绑定才能被客户端直接调用。

介绍我们的扩展示例

我们的模块系统研究的其余部分将以实现有理数的小模块作为示例。 虽然真正的库会提供更多功能，但我们的库只支持创建分数、将两个分数相加和将分数转换为字符串。 我们的库意图是（1）防止分母为零，（2）保持分数的最简形式（3/2而不是9/6和4而不是4/1）。 虽然负分数是可以的，但是库内部从不有负的分母（−3/2而不是3/ −2和3/2而不是 −3/ −2）。 下面的结构实现了所有这些想法，使用辅助函数 reduce来减少分数，该函数本身使用 gcd。

我们的模块保持不变式，就像代码顶部的注释中所示。 这些是分数的属性，所有函数都假设为真并保证保持为真。 如果一个函数违反了不变式，其他函数可能会做错事。 例如， gcd函数对于负数参数是不正确的，但因为分母从不为负数， gcd从不使用负数参数。

```
结构有理数1 =
结构
(* 不变式1：所有分母 > 0
   不变式2：有理数保持简化形式，包括
                Frac 永远不会有分母为1的情况 *)
  数据类型有理数 = 整数 of int | 分数 of int*int
  异常 错误分数

(* gcd 和 reduce 帮助保持分数简化，
   但客户端不需要知道它们 *)
(* 它们 _假设_ 输入不是负数 *)
  fun gcd (x,y) =
       if x=y
       then x
       else if x < y
       then gcd(x,y-x)
       else gcd(y,x)

   fun reduce r =
       案例 r of
            整体 _ => r
          | 分数(x,y) =>
            如果 x=0
            那么 整体 0
            else 让 d = gcd(abs x,y) in (* 使用不变式 1 *)
                      如果 d=y
                      那么 整体(x div d)
                      否则 分数(x div d, y div d)
                 结束

(* 在制作分数时，我们禁止零分母 *)
   fun make_frac (x,y) =
       如果 y = 0
       那么 抛出 BadFrac
       否则 如果 y < 0
       那么 reduce(分数(~x,~y))
       否则 reduce(分数(x,y))

(* 使用数学属性，假设结果的两个不变式都成立
   假设参数的不变式也成立 *)
   fun add (r1,r2) =
       案例 (r1,r2) of
            (整体(i),整体(j))       => 整体(i+j)
          | (整体(i),分数(j,k)) => 分数(j+k*i,k)
          | (分数(j,k),整体(i)) => 分数(j+k*i,k)
          | (Frac(a,b),Frac(c,d)) => reduce (Frac(a*d + b*c, b*d))
```

```
(* 给定不变量，以简化形式打印 *)
   fun toString r =
       case r of
           Whole i => Int.toString i
         | Frac(a,b) => (Int.toString a) ^ "/" ^ (Int.toString b)
```

结束

我们的示例签名

现在让我们尝试给我们的示例模块一个签名，以便客户端可以使用它，但不违反其不变量。

由于 reduce和 gcd是我们不希望客户端依赖或滥用的辅助函数，一个自然的签名如下所示：

```
签名 RATIONAL_A =
sig
数据类型 rational = Frac of int * int | Whole of int
异常 BadFrac
val make_frac : int * int -> rational
val add : rational * rational -> rational
val toString : rational -> string
end
```

为了使用这个签名隐藏 gcd和 reduce，我们只需要将结构定义的第一行改为structure Rational1 :> RATIONAL_A.

虽然这种方法确保客户端不会直接调用 gcd或 reduce（因为它们在模块之外“不存在”），但这还不足以确保模块中的绑定被正确使用。 对于一个模块来说，“正确”是什么意思取决于该模块的规范（而不是ML语言的定义），所以让我们更具体地描述一下我们的有理数库的一些期望属性：

- 属性： toString总是返回一个简化形式的字符串表示
- 属性：没有代码会进入无限循环
- 属性：没有代码会除以零
- 属性：没有分母为0的分数

这些属性是外部可见的；它们是我们向客户端承诺的。 相比之下，不变式是内部的；它们是关于实现的事实，有助于确保属性。 上面的代码维护了不变式，并在某些地方依赖它们来确保属性，特别是：

- 如果使用≤ 0的参数调用gcd，将违反属性，但由于我们知道分母大于0，reduce可以将分母传递给gcd而不用担心。
- toString和大多数情况下的add不需要调用reduce，因为它们可以假设它们的参数已经是简化形式。

- add使用数学性质，即两个正数的乘积是正数，所以我们知道非正分母不会引入。

不幸的是，在签名RATIONAL_A下，客户端仍然必须被信任不会破坏属性和不变量！因为签名暴露了数据类型绑定的定义，ML类型系统将不会阻止客户端直接使用构造函数Frac和Whole，绕过我们建立和保持不变量的所有工作。 客户端可以创建“不好”的分数，如Rational.Frac(1,0)、Rational.Frac(3,~2)或Rational.Frac(9,6)，其中任何一个都可能导致gcd或toString根据我们的规范出现错误行为。 虽然我们可能只希望客户端使用make_frac、add和toString，但我们的签名允许更多。

一个自然的反应是通过删除该行来隐藏数据类型绑定
数据类型有理数 = 分数 of int * int | 整数 of int. 虽然这是正确的直觉，但结果-签名没有意义，会被拒绝：它反复提到一个类型 rational，而这个类型并不存在。 我们想要表达的是有一个类型 rational，但客户端不能知道除了它存在之外的任何关于该类型的信息. 在签名中，我们可以通过抽象*type*来实现这一点，就像这个签名所示：

```
签名 RATIONAL_B =
开始
类型有理数 (*类型现在是抽象的*)
异常 BadFrac
val make_frac : int * int -> rational
val add : rational * rational -> rational
val toString : rational -> string
end
```

（当然，我们还必须改变结构定义的第一行，以使用这个签名。这
总是正确的，所以我们将停止提及它。）

抽象类型的这个新特性，只在签名中有意义，正是我们想要的。 它允许我们的模块在不透露该类型的实现的情况下定义对该类型的操作。 语法只是给出一个类型绑定而没有定义。 模块的实现没有改变；我们只是改变了客户端拥有的信息量。

现在，客户端如何创建有理数？ 嗯，第一个有理数必须使用 make_frac创建。 之后，更多的有理数可以使用 make_frac或 add创建。 没有其他方法，所以感谢我们编写的make_frac和 add，所有的有理数都将始终以简化形式存在，且分母为正数。

相比于 RATIONAL_A， RATIONAL_B从客户端那里拿走了构造函数 Frac和 Whole。 因此，客户端不能直接创建有理数，也不能对有理数进行模式匹配。 他们不知道有理数的内部表示方式。 他们甚至不知道有理数是如何实现为数据类型的。

抽象类型在编程中非常重要。

一个有趣的变化：暴露 Whole 函数

通过将有理数类型设为抽象，我们从客户端那里拿走了 Frac和 Whole构造函数。 虽然这对于确保客户端不能创建未简化或具有非正分母的分数至关重要，但只有 Frac构造函数是有问题的。 由于允许客户端直接创建整数不会违反我们的规范，我们可以添加一个类似于的函数：

```
fun make_whole x = Whole x
```

对于我们的结构和val make_whole : int -> rational对于我们的签名。 但这是不必要的函数包装; 更短的实现方法是：

```
val make_whole = Whole
```

当然，客户端无法知道我们使用的 make_whole的哪个实现。 但为什么要创建一个新的binding make_whole，它只是和 Whole一样的东西？ 相反，我们可以将构造函数作为一个函数导出，具有这个签名，而且对我们的结构没有任何更改或添加:

```
签名 RATIONAL_C =
开始
type rational (* 类型仍然是抽象的 *)
exception BadFrac
val Whole : int -> rational (* 客户端只知道 Whole 是一个函数 *)
val make_frac : int * int -> rational
val add : rational * rational -> rational
val toString : rational -> string
结束
```

这个签名告诉客户端，有一个绑定到 Whole的函数，它接受一个 int并产生一个 rational。这是正确的：这个绑定是结构中的数据类型绑定的一部分。 所以我们正在暴露部分数据类型绑定提供的内容： rational是一个类型， Whole绑定到一个函数。 我们仍然隐藏了数据类型绑定提供的其余部分： Frac构造函数和与 Frac和 Whole的模式匹配。

签名匹配的规则

到目前为止，我们对于一个结构是否“应该进行类型检查”给定一个特定的签名的讨论一直相当不正式。现在让我们列举更精确的规则，说明一个结构与一个签名匹配的含义。
（这个术语与模式匹配无关。）如果一个结构与分配给它的签名不匹配，那么该模块将无法进行类型检查。 如果一个结构名称与一个签名 BLAH匹配，则：

- 对于 BLAH中的每个val绑定，名称必须具有具有该类型或更一般类型的绑定（例如，即使签名中说它不是多态的，实现也可以是多态的-请参见下面的示例）。
- 对于 BLAH中的每个非抽象类型绑定，名称必须具有相同的类型绑定。
- 对于 BLAH中的每个抽象类型绑定，名称必须具有创建该类型的某个绑定（可以是数据类型绑定或类型同义词）。

注意名称可以有任何额外的绑定，这些绑定不在签名中。

等效实现

鉴于我们保持属性和不变量的签名 RATIONAL_B和 RATIONAL_C，我们知道客户端不能依赖于任何辅助函数或模块中定义的有理数的实际表示。 因此，只要具有相同属性的任何等效实现，我们都可以替换实现：只要

如果模块中对toString绑定的任何调用都产生相同的结果，客户端将无法区分。 这是另一个重要的软件开发任务：以不破坏客户端的方式改进/更改库。 知道客户端遵守ML签名所强制的抽象边界是非常宝贵的。

作为一个简单的例子，我们可以在 reduce内部定义一个局部函数 gcd，并且知道没有客户端会因为不能依赖 gcd的存在而无法工作。 更有趣的是，让我们改变结构的一个不变量。 让我们不要保持分数的简化形式。 相反，在将有理数转换为字符串之前，让我们将有理数化简。 这简化了 make_frac和 add，但使 toString复杂化，现在它是唯一需要 reduce的函数。 这是整个结构，仍然可以匹配签名 RATIONAL_A， RATIONAL_B或 RATIONAL_C：

```
结构有理数2 :> 有理数_A (* 或 B 或 C *) =
结构
  数据类型有理数 = 整数 of int | 分数 of int*int
  异常 错误分数

  函数 制作分数 (x,y) =
      如果 y = 0
      那么 抛出 错误分数
      否则 如果 y < 0
      那么 分数(~x,~y)
      否则 分数(x,y)

  函数 加 (r1,r2) =
      情况 (r1,r2) of
          (整数(i),整数(j))       => 整数(i+j)
        | (整数(i),分数(j,k)) => 分数(j+k*i,k)
        | (分数(j,k),整数(i)) => 分数(j+k*i,k)
        | (分数(a,b),分数(c,d)) => 分数(a*d + b*c, b*d)

  函数 转为字符串 r =
      让 函数 最大公约数 (x,y) =
              如果 x=y
              那么 x
              否则 如果 x < y
              那么 最大公约数(x,y-x)
              否则 最大公约数(y,x)

          fun reduce r =
              案例 r of
                  整体 _ => r
                | 分数(x,y) =>
                  如果 x=0
                  那么 整体 0
                  否则
                      让 val d = gcd(abs x,y) in
                          如果 d=y
                          那么 整体(x div d)
                          否则 分数(x div d, y div d)
                      结束
      在
          案例 reduce r of
```

```
            整体 i      => Int.toString i
          | 分数(a,b) => (Int.toString a) ^ "/" ^ (Int.toString b)
    结束
结束
```

如果我们给予有理数1和有理数2签名 RATIONAL_A，两者都可以通过类型检查，但客户端仍然可以区分它们。 例如，有理数1.toString(有理数1.分数(21,3))产生"21/3"，但有理数2.toString(有理数2.分数(21,3))产生"7"。 但是如果我们给予有理数1和有理数2签名 RATIONAL_B或 RATIONAL_C，那么对于任何可能的客户端，这些结构都是等价的。 这就是为什么最好一开始就使用限制性签名，比如 RATIONAL_B：这样你可以在不检查所有客户端的情况下更改结构。

尽管我们迄今为止的两个结构保持不同的不变性，但它们确实使用相同的有理数类型定义。 对于签名 RATIONAL_B或 RATIONAL_C，这是不必要的；具有这些签名的不同结构可以以不同的方式实现该类型。 例如，假设我们意识到在内部特殊处理整数可能会带来更多麻烦而不值得。 相反，我们可以只使用 int*int并定义这个结构：

```
结构有理数3 :> RATIONAL_B (或C) =
结构
    类型有理数 = int*int
    异常 BadFrac

    fun make_frac (x,y) =
        如果 y = 0
        则引发 BadFrac
        否则如果 y < 0
        则 (~x, ~y)
        否则 (x, y)

    fun Whole i = (i,1)

    fun add ((a,b),(c,d)) = (a*d + c*b, b*d)

    fun toString (x,y) =
        if x=0
        then "0"
        else
            let fun gcd (x,y) =
                    if x=y
                    then x
                    else if x < y
                    then gcd(x,y-x)
                    else gcd(y,x)
                val d = gcd (abs x,y)
                val num = x div d
                val denom = y div d
            in
                Int.toString num ^ (if denom=1
                                    then ""
                                    else "/" ^ (Int.toString denom))
            end
end
```

（这个结构采用了 Rational2的方法来缩小分数，但这个问题与有理数的定义基本无关。）

注意，这个结构提供了 RATIONAL_B所需的一切。 函数 make_frac很有趣，它接受一个 int*int并返回一个 int*int，但客户端不知道实际的返回类型，只知道抽象类型 rational。 而且，虽然在签名中给它一个 rational的参数类型会匹配，但这会使模块无用，因为客户端将无法创建一个类型为 rational的值。尽管如此，客户端不能只传递任何 int*int给 add或 toString；他们必须传递他们知道具有类型 rational的东西。与我们的其他结构一样，这意味着有理数只能通过 make_frac和 add来创建，这样可以强制执行我们的不变量。

我们的结构与 RATIONAL_A不匹配，因为它没有提供有构造函数的有理数作为数据类型 Frac和 Whole。

我们的结构与签名 RATIONAL_C匹配，因为我们明确添加了一个正确类型的函数 Whole。 没有客户端能够区分我们的"真实函数"和之前结构中 Whole构造函数作为函数的使用。

事实上fun Whole i = (i,1)与val Whole : int -> rational匹配是有趣的。 模块中Whole的类型实际上是多态的：'a -> 'a * int。 ML签名匹配允许'a -> 'a * int与int -> rational匹配，因为'a -> 'a * int比int -> int * int更一般，而int -> rational是int -> int * int的正确抽象。 更不正式地说， Whole在模块内部具有多态类型并不意味着签名必须在模块外部给它一个多态类型。

实际上，使用抽象类型时，它不能使用 Whole类型，因为它的类型不能是'a -> int * int或者 'a -> rational.

不同的模块定义了不同的类型

虽然我们定义了不同的结构（例如， Rational1， Rational2和 Rational3），但它们具有相同的签名（例如， RATIONAL_B），但这并不意味着来自不同结构的绑定可以互相使用。 例如，Rational1.toString(Rational2.make_frac(2,3))无法通过类型检查，这是一件好事，因为它会打印一个未简化的分数。 它无法通过类型检查的原因是Rational2.rational和Rational1.rational是不同的类型. 它们不是由相同的数据类型绑定创建的，即使它们看起来相同。此外，在模块外部，我们不知道它们看起来相同。实际上，Rational3.toString(Rational2.make_frac(2,3))确实不需要通过类型检查，因为Rational3.toString期望一个 int*int，但Rational2.make_frac(2,3))返回一个由Rational2.Frac构造函数创建的值。

什么是类型推断?

虽然我们一直在使用ML类型推断，但我们还没有仔细研究过它。 我们将首先仔细定义类型推断是什么，然后通过几个例子来看看ML类型推断是如何工作的。

Java、C和ML都是静态类型的语言，这意味着每个绑定都有一个在编译时确定的类型，即在程序的任何部分运行之前。类型检查器是一个在编译时接受或拒绝程序的过程。 相比之下，Racket、Ruby和Python是动态类型的语言，这意味着绑定的类型不是提前确定的，像将42绑定到x，然后将x视为字符串会导致运行时错误。 在我们使用Racket进行一些编程之后，我们将比较静态类型和动态类型的优缺点作为一个重要的课题。

与Java和C不同，ML是隐式类型的，这意味着程序员很少需要写下绑定的类型。 这通常很方便（尽管有人对于它是否使代码更容易或更难阅读存在争议），但这并不改变ML是静态类型的事实。 相反，类型检查器必须更加复杂，因为它必须推断出类型注释“本应该是什么”，如果程序员写了所有的类型注释。 原则上，类型推断和类型检查可以是分开的步骤（推断器可以完成其部分，检查器可以查看结果是否应该进行类型检查），但在实践中它们通常被合并为“类型检查器”。请注意，正确的类型推断器必须在存在解决方案的情况下找到所有类型应该是什么，否则它必须拒绝该程序。

对于特定的编程语言来说，类型推断是容易、困难还是不可能往往不明显。 它与类型系统的宽松程度无关。例如，“接受一切”和“接受什么都不接受”的“极端”类型系统都很容易进行推断。当我们说类型推断可能是不可能的时候，我们是指在技术上的不可判定性，就像著名的停机问题一样。 我们的意思是存在一些类型系统，没有计算机程序可以实现类型推断，使得（1）推断过程总是终止，（2）如果推断是可能的，推断过程总是成功，（3）如果推断是不可能的，推断过程总是失败。

幸运的是，ML被设计得相当巧妙，以便可以通过一个相当直接且优雅的算法来执行类型推断。 虽然有些程序的推断速度非常慢，但实际上人们编写的程序从来不会导致这种行为。 我们将通过几个例子演示ML类型推断算法的关键方面。 这将让你感觉到类型推断并不是“魔法”。为了继续其他课程主题，我们将不描述完整的算法或编写代码来实现它。

ML类型推断与参数多态性紧密相连 - 当推断器确定函数的参数或结果“可以是任何东西”时，生成的类型使用'a'，'b'等。但类型推断和多态性是完全独立的概念：一种语言可以具有其中之一。 例如，Java具有泛型，但没有方法参数/结果类型的推断。

ML类型推断概述

这是ML类型推断的概述（更多示例将在后面给出）：

- 它按顺序确定绑定的类型，使用先前绑定的类型来推断后续绑定的类型。 这就是为什么你不能在一个文件中使用后续绑定。 （当你需要时，你可以使用相互递归和类型推断来确定所有相互递归绑定的类型。
  相互递归在本单元的后面部分介绍。）
- 对于每个 `val`或 `fun`绑定，它分析绑定以确定关于其类型的必要事实。 例如，如果我们看到表达式 `x+1`，我们可以得出结论 `x`必须具有类型 `int`。 我们对函数调用、模式匹配等收集类似的事实。
- 之后，在函数参数或结果中使用类型变量（例如 `'a`）来表示任何未约束的类型。
- （强制执行值限制 - 只有变量和值可以具有多态类型，如后面所讨论的。）

ML类型系统的惊人之处在于以这种方式“按顺序”从不导致我们拒绝一个可以进行类型检查的程序，也从不接受我们不应该接受的程序。因此，显式类型注释实际上是可选的，除非你使用像 `#1`这样的特性。 （ `#1`的问题在于它不提供足够的信息给类型推断，以知道元组/记录应该具有哪些其他字段，而ML类型系统要求知道确切的字段数和所有字段的名称。）

这是一个初始的，非常简单的例子：

```
val x = 42
fun f(y,z,w) = if y then z+x else 0
```

类型推断首先给出 x的类型 int，因为 42的类型是 int。 然后它继续推断 f的类型。 接下来，我们将通过其他例子来逐步学习更详细的过程，但在这里，让我们列出关键事实：

- y必须具有 bool类型，因为我们在条件语句中对其进行了测试。
- z必须具有 int类型，因为我们将其添加到我们已经确定具有 int类型的某个值上。
- w可以具有任何类型，因为它从未被使用。
- f必须返回一个 int类型，因为它的主体是一个条件语句，其中两个分支都返回一个 int类型。 （如果它们不一致，类型检查将失败。）

因此， f的类型必须是bool * int * 'a -> int。

更详细的ML类型推断示例

我们现在将逐步解析几个示例，生成类型推断算法所需的所有事实。 请注意，进行头脑中的类型推断的人通常像进行头脑中的算术一样采取捷径，但关键是有一个通用算法，它系统地遍历代码，收集约束并将它们组合以得到答案。

作为第一个例子，考虑推断这个函数的类型：

```
fun f x =
   let val (y,z) = x in
       (abs y) + z
   end
```

下面是我们如何推断类型的：

- 观察第一行， f必须有类型 T1->T2，其中 T1和 T2是某些类型，并且在函数体中 f有这个类型， x有类型 T1。
- 观察 val绑定， x必须是一个对偶类型（否则模式匹配没有意义），所以实际上 T1=T3*T4，其中 T3和 T4是某些类型， y有类型 T3， z有类型 T4。
- 观察加法表达式，我们从上下文中知道 abs有类型 int->int，所以 y有类型 T3意味着 T3=int。 同样地，由于abs y有类型 int， +的另一个参数必须有类型int，所以 z有类型 T4意味着 T4=int。
- 由于加法表达式的类型是 int，let表达式的类型也是 int。 由于let表达式的类型是 int，函数 f的返回类型是 int，即 T2=int。

将所有这些约束放在一起， T1=int*int（因为 T1=T3*T4）， T2=int，所以 f的类型是int*int->int。

下一个例子：

```
fun sum xs =
   case xs of
     [] => 0
   | x::xs' => x + (xs'的和)
```

- 从第一行可以得出，存在类型 T1和 T2使得 sum的类型是 T1->T2， xs的类型是 T1。
- 观察case表达式， xs必须具有与所有模式兼容的类型。 观察模式，两个模式都匹配任何列表，因为它们是由列表构造函数构建的（在 x::xs'的情况下，子模式匹配任何类型的任何内容）。 所以由于 xs的类型是 T1，实际上T1=T3 list，其中 T3是某种类型。
- 观察case分支的右侧，我们知道它们必须与彼此具有相同的类型，而这个类型是 T2。 由于0的类型是 int，所以 T2=int。
- 观察第二个case分支，我们在一个上下文中对其进行类型检查，其中 x和 xs'是可用的。 由于我们正在将模式 x::xs'与一个T3 list进行匹配，所以必须是 x具有类型 T3和 xs'具有类型T3 list。
- 现在观察右侧，我们添加了 x，所以实际上 T3=int。 此外，递归调用进行了类型检查，因为 xs'具有类型T3 list和T3 list=T1，而 sum具有类型 T1->T2。 最后，由于 T2=int，所以sum xs'通过了类型检查。

将所有内容放在一起，我们得到 sum具有类型int list -> int。

请注意，在我们到达sum xs'之前，我们已经推断出了所有内容，但我们仍然需要检查类型是否一致，如果不一致则拒绝。 例如，如果我们写成sum x，那是无法通过类型检查的——它与先前的事实不一致。 让我们更详细地看看发生了什么：

```
fun broken_sum xs =
   情况 xs of
     [] => 0
   | x::xs' => x + (broken_sum x)
```

- 对于 broken_sum的类型推断基本上与 sum相同。 前面例子中的前四个要点都适用，给出 broken_sum的类型T3 list ->int， x3的类型T3 list， x的类型 T3，以及xs'的类型T3 list。此外， T3=int。
- 我们在调用broken_sum x时与正确的 sum实现有所不同。 对于这个调用的类型检查， x必须与 broken_sum的参数具有相同的类型，换句话说， T1=T3。 然而，我们知道T1=T3 list，所以这个新的约束 T1=T3实际上产生了一个矛盾：T3=T3 list。
  如果我们想更具体一些，我们可以利用我们知道的 T3=int来重写为int=int list。
  从 broken_sum的定义来看，很明显这就是问题所在：我们试图将 x作为一个 int和一个int list来使用。

当你的ML程序无法通过类型检查时，类型检查器会报告发现矛盾的表达式以及涉及到的类型。 虽然有时这些信息是有帮助的，但其他时候实际问题可能出现在另一个表达式上，但类型检查器直到后来才发现矛盾。

多态类型的示例

我们剩下的示例将推断出多态类型。 我们所做的只是按照上面的步骤进行，但完成后，函数类型的某些部分仍然是*unconstrained*的。 对于每个可以是任何类型的Ti，我们使用一个类型变量（'a，'b等）。

```
fun length xs =
   case xs of
     [] => 0
   | x::xs' => 1 + (xs'的长度)
```

类型推断的过程与 sum类似。我们最终确定：

- length的类型是 T1->T2。
- xs的类型是 T1。
- T1=T3 list(由于模式匹配)
- T2=int因为0可以是 length的调用结果。
- x的类型是 T3，而 xs'的类型是T3 list。
- 递归调用length xs'通过类型检查，因为 xs'的类型是T3 list，即 length的参数类型 T1。 而且我们可以将结果相加，因为 T2=int。

所以我们有与 sum相同的约束条件，只是我们没有 T3=int。 实际上， T3可以是任何值， length仍然可以通过类型检查。 因此，类型推断在完成时识别出 length的类型是T3 list -> int，而 T3可以是任何值。 所以最终得到的类型是'a list -> int，与预期相符。 再次强调，规则很简单：对于最终结果中无法约束的每个 T i，使用一个类型变量。

第二个例子：

```
fun compose (f,g) = fn x => f (g x)
```

- 由于 compose的参数必须是一对（根据其参数的模式）， compose的类型为 T1*T2->T3， f的类型为 T 1， g的类型为 T2。
- 由于 compose返回一个函数， T3是一个 T4->T5，在该函数的主体中， x的类型为 T4。
- 所以 g必须具有类型 T4->T6，其中 T6为某个类型，即 T2=T4->T6。
- 而 f必须具有类型 T6->T7，其中 T7为某个类型，即 T1=T6->T7。
- 但 f的结果是 compose返回的函数的结果，所以 T7=T5，因此 T1=T6->T5。

将 T1=T6->T5、 T2=T4->T6和 T3=T4->T5组合起来，我们得到 compose的类型为(T6->T5)*(T4->T6) -> (T4->T5)。 没有其他约束 T4、 T5和 T6的类型，因此我们一致地替换它们，得到('a->'b)*('c->'a) -> ('c->'b)，如预期的那样（最后一组括号是可选的，但这只是语法上的）。这里有一个更简单的例子，也有多个类型变量：

```
fun f (x,y,z) =
    if true
    then (x,y,z)
    else (y,x,z)
```

- 第一行要求 f的类型为T1*T2*T3 -> T4， x的类型为 T1， y的类型为 T2，而 z的类型为 T3。
- 条件语句的两个分支必须具有相同的类型，这个类型也是函数的返回类型 T4。 因此， T4=T1*T2*T3且T4=T2*T1*T3。 这个约束要求 T1=T2。

将这些约束整合在一起（没有其他约束）， f将以类型T1*T1*T3 -> T1*T1*T3通过类型检查，对于任意类型 T1和 T3。 因此，将每个类型一致地替换为一个类型变量，我们得到'a*'a*'b -> 'a*'a*'b，这是正确的： x和 y必须具有相同的类型，但 z可以（但不必）具有不同的类型。 注意，类型检查器总是要求条件语句的两个分支具有相同的类型，即使在这里我们知道哪个分支将被执行。

可选：值限制

正如在本单元中所描述的那样，ML类型系统是不完备的，这意味着它会接受那些在运行时可能具有错误类型值的程序，比如将一个整数放在我们期望一个字符串的位置。 这个问题是由多态类型和可变引用的组合引起的，解决方法是一种特殊的类型系统限制，称为值限制。

这是一个演示问题的示例程序：

```
val r = ref NONE         (* 'a option ref *)
val _ = r := SOME "hi" (* 实例化 'a 为字符串 *)
val i = 1 + valOf(!r) (* 实例化 'a 为整数 *)
```

直接使用类型检查/推断规则会接受这个程序，尽管我们不应该这样做 - 我们最终尝试将 1加到 "hi"上。 然而，给定函数/运算符 ref ('a -> 'a ref)、 := ('a ref * 'a -> unit)和 ! ('a ref -> 'a)的类型，似乎一切都能通过类型检查。

为了恢复完整性，我们需要一个更严格的类型系统，不允许这个程序通过类型检查。 ML的选择是防止第一行具有多态类型。 因此，第二行和第三行将无法通过类型检查，因为它们无法实例化一个'a'为string或int。 一般来说，只有当val-binding中的表达式是一个值或变量时，ML才会给一个变量赋予多态类型。 这被称为值约束。 在我们的例子中，ref NONE是对函数ref的调用。函数调用不是变量或值。 所以我们得到一个警告，并且r被赋予了类型?X1 option ref，其中?X1是一个“虚拟类型”，而不是类型变量。 这使得r变得无用，后面的行无法通过类型检查。

这个限制是否足以使类型系统具有完整性并不明显，但事实上它是足够的。

对于上面的例子，我们可以使用表达式ref NONE，但是我们必须使用类型注释给出一个非多态类型，比如int option ref。 无论我们选择什么，下面的两行中的一行都无法通过类型检查。

正如我们之前在研究部分应用时看到的，即使在不使用突变的情况下，值限制有时也是繁琐的。 我们看到这个绑定受到了值限制的限制，不能变成多态的：

```
val pairWithOne = List.map (fn x => (x,1))
```

我们看到了多种解决方法。 其中一种方法是使用函数绑定，即使没有值限制，它也是不必要的函数包装。 这个函数具有所需的类型'a list -> ('a * int) list：

```
fun pairWithOne xs = List.map (fn x => (x,1)) xs
```

有人可能会想为什么我们不能只对引用（我们需要它的地方）强制执行值限制，而不对不可变类型如列表强制执行。 答案是ML类型检查器无法总是知道哪些类型是真正的引用，哪些不是。在下面的代码中，我们需要在最后一行强制执行值限制，因为'a foo和'a ref是相同的类型。

```
type 'a foo = 'a ref
val f : 'a -> 'a foo = ref
val r = f NONE
```

由于ML的模块系统，类型检查器并不总是知道类型同义词的定义
（回想一下这是一件好事）。 所以为了安全起见，它对所有类型强制执行值限制。

可选：使类型推断更加困难的一些事情

现在我们已经看到了ML类型推断的工作原理，我们可以得出两个有趣的观察结果：

- 如果ML具有子类型化（例如，如果每个三元组也可以是一个二元组），那么推断将更加困难，因为我们将无法得出"T3=T1*T2"这样的结论，因为等号将过于限制。
  相反，我们需要约束来指示 T3是一个至少有两个字段的元组。 根据不同的细节，这是可以做到的，但类型推断更加困难，结果更加难以理解。

- 如果ML没有参数多态性，推理将更加困难，因为我们必须为像 length和 compose这样的函数选择一些类型，并且这可能取决于它们的使用方式。

相互递归

我们已经看到了许多递归函数的例子，也看到了许多使用其他函数作为辅助函数的函数的例子，但是如果我们需要一个函数 f调用 g，而 g又调用 f呢？ 这肯定是有用的，但是ML的规则是绑定只能使用先前的绑定，这使得它更加困难——f还是 g应该先出现？

事实证明，ML对相互递归有特殊支持，使用关键字 and并将相互递归的函数放在一起。 同样，我们可以有相互递归的 datatype绑定。 在展示了这些新构造之后，我们将展示，实际上可以通过使用高阶函数来解决对相互递归函数的支持不足的问题，这在一般情况下和特别是在ML中是一个有用的技巧，如果你不想让你的相互递归函数放在一起的话。

我们的第一个例子使用互递归来处理一个整数列表，并返回一个布尔值。 如果列表严格交替出现1和2，并以2结尾，则返回true。 当然，有很多方法可以实现这样的函数，但我们的方法在每个"状态"（比如"下一个必须是1"或"下一个必须是2"）都有一个函数，做得很好。 一般来说，计算机科学中的许多问题都可以用这种方式建模。

有限状态机是一种优雅的实现有限状态机的方法，每个状态都有一个相互递归的函数。

```
fun match xs =
    let fun s_need_one xs =
            case xs of
                [] => true
              | 1::xs' => s_need_two xs'
              | _ => false
        and s_need_two xs =
            case xs of
                [] => false
              | 2::xs' => s_need_one xs'
              | _ => false
    in
        s_need_one xs
    end
```

（该代码在模式中使用整数常量，这是一种偶尔方便的ML特性，但对于示例来说并不是必需的。）

在语法上，我们通过简单地用关键字 fun替换除第一个函数以外的所有函数来定义相互递归函数。 类型检查器将一起对所有函数（例如上面的两个函数）进行类型检查，允许它们之间的调用无论顺序如何。

这是一个第二个（愚蠢的）示例，也使用了两个相互递归的数据类型绑定。 类型 t1和 t2的定义相互引用，这是通过在第二个定义中使用 and而不是 datatype来实现的。这定义了两个新的数据类型， t1和 t2。

```
数据类型t1= Foo of int | Bar of t2
和t2= Baz of string | Quux of t1

fun no_zeros_or_empty_strings_t1 x =
    情况 x of
        Foo i => i <> 0
      | Bar y => no_zeros_or_empty_strings_t2 y
和 no_zeros_or_empty_strings_t2 x =
    情况 x of
        Baz s => size s > 0
      | Quux y => no_zeros_or_empty_strings_t1 y
```

现在假设我们想要实现上面代码中的“没有零或空字符串”的功能，但由于某种原因我们不想将函数放在一起，或者我们所使用的语言不支持相互递归的函数。 我们可以通过将“后面”函数传递给一个接受函数作为参数的“前面”函数的版本来编写几乎相同的代码：

```
fun no_zeros_or_empty_strings_t1(f,x) =
    情况 x of
        Foo i => i <> 0
      | Bar y => f y
```

[1]因为所有的函数调用都是尾调用，所以代码在很小的空间中运行，就像对有限状态机的实现所期望的那样。

```
fun no_zeros_or_empty_string_t2 x =
    情况 x of
        Baz s => size s > 0
      | Quux y => no_zeros_or_empty_strings_t1(no_zeros_or_empty_string_t2,y)
```

这是另一种强大的函数允许函数的习惯用法。

激励和定义等价性

一个代码片段与另一个代码片段“等价”的想法是编程和计算机科学的基础。 每当你简化一些代码或说，“这是另一种做同样事情的方法”时，你都在非正式地思考等价性。这种推理在几种常见情况下出现：

- 代码维护：你能简化、清理或重新组织代码而不改变程序的其他部分的行为吗？
- 向后兼容性：你能添加新功能而不改变任何现有功能的工作方式吗？
- 优化：你能用更快或更节省空间的实现替换代码吗？
- 抽象：外部客户端能否察觉到我对代码进行了这个改变？

还要注意，我们在之前的讲座中对于限制性签名的使用主要是关于等价性：通过使用更严格的接口，我们使得更多不同的实现等价，因为客户端无法区分它们之间的差异。

我们希望对等价性有一个精确的定义，以便我们可以判断某些代码维护形式或不同的签名实现是否真的可以接受。 我们不希望定义过于严格，以至于我们无法进行改进代码的更改，但我们也不希望定义过于宽松，以至于用一个“等价”的函数替换另一个函数会导致我们的程序产生不同的答案。 希望通过学习等价性的概念和理论，能够改善你对任何语言编写的软件的看法。

有很多不同的可能定义可以在严格/宽松的张力中稍微有所不同。 我们将专注于一个对于设计和实现编程语言的人们来说有用且常见的定义。 我们还将简化讨论，假设我们有两个函数的实现，并且我们想知道它们是否等价。

我们定义的直觉如下：

- 函数 `f`等价于函数 `g`（或其他代码片段类似），如果它们产生相同的答案，并且无论在任何程序中以任何参数调用它们，它们都具有相同的副作用。
- 等价性不要求相同的运行时间，相同的内部数据结构，相同的辅助函数等。所有这些都被认为是“不可观察的”，即不影响等价性的实现细节。

举个例子，考虑两种非常不同的排序列表的方式。只要它们对所有输入产生相同的最终
答案，无论它们在内部如何工作或者其中一种方式是否

更快。然而，如果它们对于某些列表的行为不同，也许对于有重复元素的列表，那么它们就不等价。

然而，上面的讨论是简化的，隐含地假设函数总是返回并且除了产生答案之外没有其他影响。 为了更加精确，我们需要当两个函数在相同的环境中给出相同的参数时：

1. 产生相同的结果（如果它们产生结果）
2. 具有相同的（非）终止行为；即，如果一个永远运行，另一个也必须永远运行
3. 以相同的方式改变（对客户可见的）内存。
4. 执行相同的输入/输出
5. 引发相同的异常

这些要求对于知道如果我们有两个等价的函数，我们可以用另一个替换一个，并且程序中的任何使用都不会有不同的行为是很重要的。

无副作用编程的另一个好处

确保两个函数具有相同的副作用（改变引用，执行输入/输出等）的一种简单方法是根本没有副作用。这正是像ML这样的函数式语言所鼓励的。 是的，在ML中，你可以让函数体改变一些全局引用或其他东西，但这通常是不好的风格。
其他函数式语言是纯函数式语言，意味着在（大多数）函数内部没有办法进行变异
。
如果作为一项政策，在函数体中不进行变异、打印等操作，那么调用者可以假设很多等价性，否则不行。 例如，我们可以用`(f x)*2`替换`(f x)+(f x)`吗？ 一般来说，这可能是错误的，因为调用 `f`可能会更新某个计数器或打印一些东西。 在ML中，这也是可能的，但作为一种风格，这种情况发生的可能性要小得多，所以我们倾向于有更多的等价性。 在纯函数式语言中，我们保证替换不会改变任何东西。 总的来说，当你尝试确定两段代码是否等价时，变异确实会妨碍你的判断——这是避免变异的一个很好的理由。

除了在维护无副作用程序时能够消除重复计算（如上面的`(f x)`），我们还可以更自由地重新排序表达式。 例如，在Java、C等语言中：

```
int a = f(x);
int b = g(y);
返回 b - a;
```

可能会产生不同的结果：

```
返回 g(y) - f(x);
```

因为 `f`和 `g`可能以不同的顺序被调用。 再次强调，在ML中也是可能的，但是如果我们避免副作用，这种情况发生的可能性就会小很多。 （然而，我们可能仍然需要担心不同的异常被抛出
以及其他细节。）

标准等价性

等价性是微妙的，特别是当你试图在不知道
它们可能被调用的所有位置的情况下判断两个函数是否等价时。 然而，这是很常见的，比如当你在编写一个未知客户可能使用的库时。我们现在考虑几种情况，在任何情况下都可以保证等价性，因此
这些是很好的经验法则，也是关于函数和闭包如何工作的良好提醒。

首先，回顾我们学过的各种形式的语法糖。 我们总是可以在函数体中使用或不使用语法糖，并得到一个等价的函数。 如果我们不能，那么我们使用的结构实际上不是语法糖。 例如，无论 g绑定到什么，这些对 f的定义都是等价的：

```
fun f x =                fun f x =
    如果 x                   x andalso g x
    那么 g x
    否则 false
```

请注意，我们不能必然地用if g x then x else false替换x andalso g x，因为 g可能具有副作用或不终止。

其次，我们可以更改局部变量（或函数参数）的名称，只要我们一致地更改所有使用它的地方。 例如，这两个 f的定义是等价的：

```
val y = 14               val y = 14
fun f x = x+y+x          fun f z = z+y+z
```

但是有一个规则：在选择新的变量名时，不能选择函数体已经用来引用其他东西的变量。 例如，如果我们尝试用 y替换 x，我们得到fun y = y+y+y，这与我们开始的函数不同。 以前未使用的变量从不是问题。

第三，我们可以使用或不使用辅助函数。 例如，这两个定义 g是等价的：

```
val y = 14               val y = 14
fun g z = (z+y+z)+z      fun f x = x+y+x
                         fun g z = (f z)+z
```

同样，我们必须小心，不要因为 f和 g可能有不同的环境而改变变量的含义。 例如，在这里 g的定义是不等价的：

```
val y = 14               val y = 14
val y = 7                fun f x = x+y+x
fun g z = (z+y+z)+z      val y = 7
                         fun g z = (f z)+z
```

第四，正如我们之前解释过的匿名函数一样，不必要的函数包装是不好的风格，因为有一种更简单的等价方式。例如，fun g y = f y和val g = f总是等价的。 然而，这里有一个微妙的复杂性。 当我们有一个像 f这样的变量绑定到我们正在调用的函数时，这是有效的，但在更一般的情况下，我们可能有一个 *expression*评估为一个我们然后调用的函数。 对于任何 *expression* e，fun g y = e y和val g = e总是相同的吗？

不。

作为一个愚蠢的例子，考虑fun h() (print "hi" ; fn x => x+x)和 e是 h()。 然后fun g y = (h()) y是一个每次调用都会打印的函数。 但是val g = h()是一个不打印的函数 -

程序在创建 g的绑定时会打印 "hi"一次。 这不应该是神秘的：我们知道val-绑定会立即评估其右侧，但是函数体在被调用之前不会被评估。

一个不那么愚蠢的例子可能是如果 h可能引发异常而不是返回一个函数。

第五，几乎可以认为let val p = e1 in e2 end可以是(fn p => e2) e1的语法糖。 毕竟，对于任何表达式 e1和 e2以及模式 p，以下两段代码都成立：

- 将 e1评估为一个值
- 将值与模式进行匹配 p
- 如果匹配成功，则在扩展了模式匹配的环境中将e2求值为一个值
- 返回求值e 2的结果

由于这两段代码“做”的事情完全相同，它们必须是等价的。 在Racket中，这将是这种情况（使用不同的语法）。 在ML中，唯一的区别是类型检查器：在let版本中，变量在p中允许具有多态类型，但在匿名函数版本中不允许。

例如，考虑let val x = (fn y => y) in (x 0, x true) end。 这个愚蠢的代码进行类型检查并返回(0,true)，因为x的类型是'a->'a。 但是(fn x => (x 0, x true)) (fn y => y)无法进行类型检查，因为我们无法给x一个非多态类型，并且函数参数不能具有多态类型。 这就是ML中类型推断的工作原理。

重新审视我们对等价性的定义

根据设计，我们对等价性的定义忽略了函数评估所需的时间或空间。 因此，即使两个函数总是返回相同的答案，它们也可能是等价的，即使一个函数需要一纳秒，另一个函数需要一百万年。 从某种意义上说，这是一件好事，因为这个定义允许我们用纳秒版本替换百万年版本。

但显然其他定义也很重要。数据结构和算法课程研究渐进复杂度，以便能够区分一些算法为“更好”（这显然意味着一些“差异”），即使更好的算法产生相同的答案。 此外，渐进复杂度根据设计忽略了在某些程序中可能很重要的“常数因子开销”，因此这个更严格的等价性定义可能过于宽松：我们可能实际上想知道两个实现需要“大约相同的时间”。

这些定义都没有优势。 所有这些都是计算机科学家经常使用的有价值的观点。 可观察行为（我们的定义），渐近复杂度和实际性能都是软件开发人员几乎每天都在使用的智力工具。

CSE341：程序设计语言 2016年春季
第5单元总结

标准描述：这个总结大致涵盖了课堂和复习部分的内容。 当回顾材料时，以叙述方式阅读材料并将整个单元的材料放在一个文档中可能会有所帮助。 请报告这些笔记中的错误，甚至是打字错误。 这个总结不能完全替代上课、阅读相关代码等。

目录

从ML切换到Racket

在接下来的几周里，我们将使用Racket编程语言（而不是ML）和Dr-Racket编程环境（而不是SML/NJ和Emacs）。有关安装和基本使用说明的笔记在课程网站上的另一个文档中。

我们的重点仍然主要集中在关键的编程语言结构上。 我们将“切换”到Racket，因为其中一些概念在Racket中更加突出。尽管如此，Racket和ML有许多相似之处：它们都是大部分是函数式语言（即，变异存在但不鼓励使用）具有闭包、匿名函数、对列表的方便支持、没有返回语句等。在第二种语言中看到这些特性应该有助于加强基本思想。 一个中等的不同之处是我们将不在Racket中使用模式匹配。

对我们来说，Racket和ML之间最重要的区别是：

- Racket不使用静态类型系统。因此它可以接受更多的程序，程序员也不需要

要一直定义新类型，但大多数错误直到运行时才会发生。

- Racket具有非常简约和统一的语法。

Racket具有许多高级语言特性，包括宏、与ML非常不同的模块系统，引用/求值，一级续延，合同等等。我们只有时间涵盖其中的一小部分话题。

前几个主题涵盖了基本的Racket编程，因为我们需要在开始使用它之前介绍Racket来学习更高级的概念。我们将快速完成这个任务，因为(a)我们已经见过类似的语言和(b)Racket指南，`http://docs.racket-lang.org/guide/index.html`和其他文档在`http://racket-lang.org/`上都是优秀且免费的。

Racket vs. Scheme

Racket是从Scheme演变而来的，Scheme是一个自1975年以来不断发展的著名编程语言。（Scheme又是从LISP演变而来的，LISP自1958年以来不断发展。）Racket的设计者们在2010年决定进行足够的改变和添加，以至于给结果起一个新的名字比仅仅将其视为Scheme的方言更有意义。这两种语言在很多方面仍然非常相似，有一小部分关键差异（如空列表的写法、由cons构建的对是否可变、模块的工作方式），还有一长串次要差异和Racket提供的一长串新增功能。

总的来说，Racket是一门现代化的语言，正在积极开发中，并已被用于构建几个"真实"的（无论这意味着什么）系统。相对于Scheme的改进使其成为本课程和实际编程的良好选择。然而，它更像是一个"移动目标"——设计者们在努力使语言和配套的DrRacket系统更好时，并不感觉受到历史先例的束缚。因此，课程材料中的细节更有可能过时。

入门：定义，函数，列表（和 `if`）

Racket文件（也是Racket模块）的第一行应该是

```
#lang racket
```

这在课程的安装/使用说明中有讨论。这些讲义将更多地关注这行之后的内容。Racket文件包含一系列定义。

像这样的定义

```
(define a 3)
```

扩展顶层环境，使得 `a`绑定到3。Racket对变量名中可以出现的字符有非常宽松的规定，一个常见的约定是使用连字符来分隔单词，例如`my-favorite-identifier`。

后续的定义如

```
(define b (+ a 2))
```

将 `b`绑定到5。一般来说，如果我们有`(define x e)`其中 `x`是一个变量， `e`是一个表达式，我们会对 `e`求值并改变环境，使得 `x`绑定到该值。除了语法之外，

尽管在讲座结束时我们将讨论这一点，但这应该看起来非常熟悉，与ML不同的是，绑定可以引用文件中的后续绑定。 在Racket中，一切都是前缀的，比如上面使用的加法函数。

一个接受一个参数的匿名函数写作(lambda (x) e)其中参数是变量 x，主体是表达式 e。 所以这个定义将一个立方函数绑定到 cube1：

```
(定义 cube1
  (lambda (x)
    (* x (* x x))))
```

在Racket中，不同的函数确实接受不同数量的参数，如果用错误的数量调用函数，会导致运行时错误。 一个有三个参数的函数看起来像(lambda (x y z) e)。 然而，许多函数可以接受任意数量的参数。 乘法函数 * 就是其中之一，所以我们可以写成

```
 (定义 立方2
  (lambda (x)
    (* x x x)))
```

您可以查阅Racket文档以了解如何定义自己的变量数量可变的函数。

与ML不同，您可以在匿名函数中使用递归，因为定义本身在函数体中是可见的：

```
(定义 幂
  (lambda (x y)
    (如果 (= y 0)
        1
        (* x (幂 x (- y 1))))))
```

上面的例子还使用了if表达式，其一般语法为(if e1 e2 e3)。它评估e1。 如果结果是#f（Racket的false常量），则评估e3以获得结果。 如果结果是其他任何值，包括#t（Racket的true常量），则评估e2以获得结果。 请注意，这在类型方面比ML中的任何内容都更灵活。

有一种非常常见的语法糖形式，您应该用于定义函数。 它不明确使用lambda这个词：

```
(定义 (立方3 x)
  (* x x x))
(定义 (幂 x y)
    (如果 (= y 0)
        1
        (* x (幂 x (- y 1)))))
```

这更像是ML的 fun绑定，但在ML中，fun不仅仅是语法糖，因为它对于递归是必要的。

我们可以在Racket中使用柯里化。毕竟，Racket的一等函数就像ML中的闭包一样，并且柯里化只是一种编程习惯。

```
(定义 幂
  (lambda (x)
    (lambda (y)
      (如果 (= y 0)
          1
          (* x ((幂 x) (- y 1)))))))))

(定义 三次方 (幂 3))
(定义 八十一 (三次方 4))
(定义 十六 ((幂 2) 4))
```

因为Racket的多参数函数确实是多参数函数（不是其他东西的语法糖），所以柯里化不太常见。 调用柯里化函数没有语法糖：我们必须写成((幂 2) 4)因为(pow 2 4)调用绑定到pow的单参数函数，带有两个参数，这是一个运行时错误。 Racket已经为定义柯里化函数添加了语法糖。我们可以这样写：

```
(define ((pow x) y)
   (如果 (= y 0)
       1
       (* x ((pow x) (- y 1)))))
```

这是一个相当新的功能，可能不为人所知。

Racket内置了列表，就像ML一样，在实践中Racket程序可能更经常使用列表。 我们将使用内置函数来构建列表，提取部分，并检查列表是否为空。 函数名 car和 cdr是一个历史意外。

原始	描述	例子
null	空列表	null
cons	构建列表	(cons 2 (cons 3 null))
car	获取列表的第一个元素	(car some-list)
cdr	获取列表的尾部	(cdr some-list)
null?	对于空列表返回 #t，否则返回 #f	(null? some-value)

与Scheme不同，你不能写 ()表示空列表。 你可以写 '()，但我们更喜欢使用 null。

还有一个内置函数 list用于从任意数量的元素构建列表，所以你可以写(list 2 3 4)而不是(cons 2 (cons 3 (cons 4 null)))。 列表不需要保存相同类型的元素，所以你可以创建(list #t "hi" 14)而不会出错。

这里有三个列表处理函数的例子。 map和 append实际上是默认提供的，所以我们不需要自己编写。

```
(define (sum xs)
  (if (null? xs)
      0
      (+ (car xs) (sum (cdr xs)))))

(define (append xs ys)
  (if (null? xs)
      ys
      (cons (car xs) (append (cdr xs) ys))))
```

```
(define (map f xs)
  (if (null? xs)
      null
      (cons (f (car xs)) (map f (cdr xs)))))
```

语法和括号

忽略一些花哨的东西，Racket有一个非常简单的语法。 语言中的一切都是：

- 某种形式的原子，例如 `#t`, `#f`, `34`, `"hi"`, `null`等等。特别重要的原子形式是标识符，它可以是变量（例如， `x`或`something-like-this!`）或特殊形式，例如`define`, `lambda`, `if`等等。

- 一系列括号中的事物`(t1 t2 ... tn)`.

序列中的第一件事情影响了序列的其余部分的含义。 例如，`(define ...)`意味着我们有一个定义，下一件事可以是要定义的变量或者是函数定义的糖化版本的序列。

如果序列中的第一件事不是特殊形式，并且序列是表达式的一部分，那么我们有一个函数调用。 Racket中的许多事物只是函数，例如 `+`和 `>`.

作为一个小注，Racket还允许在任何地方使用 `[`和 `]`代替（和 ）. 作为一种风格，我们将展示一些常见的首选选项，其中 `[...]`是常见的首选选项。 Racket不允许不匹配的括号形式：（必须与)匹配， `[`必须与 `]`匹配。 DrRacket使这变得容易，因为如果你输入)来匹配 `[`，它会自动输入 `]`。

通过“将所有内容括在括号中”，Racket具有一个明确的语法。 关于是否 `1+2*3`是 `1+(2*3)`还是 `(1+2)*3`以及是否`f x y`是`(f x) y`还是`f (x y)`，从来没有任何规则需要学习。 它使得解析程序文本并将其转换为表示程序结构的树变得非常简单。请注意，基于XML的语言（如HTML）采用了相同的方法。 在HTML中，“开括号”看起来像 `<foo>`，而匹配的闭括号看起来像 `</foo>`。

由于某种原因，HTML很少被批评为充斥着括号，但是对于LISP、Scheme和Racket来说，这是一个常见的抱怨。如果你在街上遇到一个程序员并问他或她对这些语言的看法，他们可能会说一些关于“所有这些括号”的话。这是一种奇怪的困扰：使用这些语言的人很快就会习惯，并且发现统一的语法很愉快。 例如，它使得编辑器能够很容易地正确缩进您的代码。

从学习编程语言和基本编程结构的角度来看，你应该意识到对括号（无论是支持还是反对）的强烈偏见是一种语法偏见。
虽然每个人都有自己对语法的个人看法，但不应该让它阻止你学习Racket中的高级思想，比如卫生宏或动态类型语言中的抽象数据类型或一流的延续。 一个类比是，如果一个欧洲历史学生不想学习法国大革命，因为他或她对带有法国口音的人不感兴趣。

尽管如此，在Racket中进行实际编程确实需要正确使用括号，而Racket与ML、Java、C等在一个重要方面有所不同：括号会改变程序的含义。
你不能随意添加或删除括号，它们从来都不是可选的或无意义的。

在表达式中，`(e)`表示评估`e`，然后用0个参数调用结果函数。 所以`(42)`将会导致运行时错误：你把数字42当作函数来处理。 同样地，`((+ 20 22))`是一个错误的表达式。

同样的原因。

对于新接触Racket的程序员来说，有时很难记住括号的重要性，并确定程序为什么会失败，通常是在运行时，当它们被错误地使用括号时。 以这七个定义为例。 第一个是阶乘的正确实现，其他的都是错误的：

```
(define (fact n) (if (= n 0) 1 (* n (fact (- n 1)))))       ; 1
(define (fact n) (if (= n 0) (1) (* n (fact (- n 1))))) ; 2
(define (fact n) (if = n 0 1 (* n (fact (- n 1)))))         ; 3
(define fact (n) (if (= n 0) 1 (* n (fact (- n 1)))))       ; 4
(define (fact n) (if (= n 0) 1 (* n fact (- n 1))))         ; 5
(define (fact n) (if (= n 0) 1 (* n ((fact) (- n 1))))) ; 6
(define (fact n) (if (= n 0) 1 (n * (fact (- n 1)))))       ; 7
```

行	错误
2	调用 1作为一个不带参数的函数
3	使用 if有5个子表达式而不是3个
4	错误的定义语法： (n) 看起来像是一个表达式后面跟着更多的东西
5	调用 *其中一个参数是一个函数
6	调用 fact 没有参数
7	将 n作为一个函数并用 *调用它

动态类型（和 cond)

Racket在运行之前不使用静态类型系统来拒绝程序。作为一个极端的例子，函数(lambda () (1 2))是一个完全正常的零参数函数，如果你从来不调用它，它会导致错误。我们将在以后的讲座中花费大量时间比较动态和静态类型以及它们的相对优势，但现在我们想要适应动态类型。

举个例子，假设我们想要有一个数字列表，但其中一些元素实际上可以是其他包含数字或其他列表等等的列表，可以无限层级。 Racket直接允许这样做，例如，(list 2 (list 4 5) (list (list 1 2) (list 6)) 19 (list 14 0))。 在ML中，这样的表达式将无法通过类型检查；我们需要创建自己的数据类型绑定，并在正确的位置使用正确的构造函数。

现在在Racket中，假设我们想要对这样的列表进行计算。 这也不是问题。 例如，这里我们定义了一个函数来对这样的数据结构中的所有数字求和：

```
(define (sum xs)
  (if (null? xs)
      0
      (if (number? (car xs))
          (+ (car xs) (sum (cdr xs)))
          (+ (sum (car xs)) (sum (cdr xs))))))
```

这段代码简单地使用了内置的空列表判断函数（null?）和数字判断函数（number?）。 最后一行假设(car xs)是一个列表；如果不是，那么函数的使用方式错误，我们将得到一个运行时错误。

我们现在离题引入 cond特殊形式，这比实际上使用多个if表达式更好的嵌套条件语句的风格。 我们可以将前一个函数重写为：

```
(define (sum xs)
```

```
  (cond [(null? xs) 0]
        [(number? (car xs)) (+ (car xs) (sum (cdr xs)))]
        [#t (+ (sum (car xs)) (sum (cdr xs)))]))
```

一个 cond只有任意数量的带括号的表达式对，[e1 e2]。第一个是一个测试；如果它求值为 #f，我们跳到下一个分支。 否则我们求值 e2，这就是答案。 作为一种风格，你的最后一个分支应该有测试 #t，这样你就不会“掉到底部”，在这种情况下结果是某种“空对象”，你不想处理它。

与 if一样，测试的结果不一定是 #t或 #f。 除了 #f之外的任何值在测试的目的上都被解释为真。 有时候滥用这个特性是不好的风格，但它可以很有用。

现在让我们进一步改变动态类型的规范，为我们的 sum函数做出改变。 假设我们甚至想要在列表中允许非数字和非列表，这种情况下我们只想通过将0添加到总和中来“忽略”这些元素。如果这是你想要的（可能并不是 - 这可能会在你的程序中悄悄隐藏错误），那么我们可以在Racket中实现。除非初始参数既不是数字也不是列表，否则此代码将永远不会引发错误：

```
(define (sum xs)
  (cond [(null? xs) 0]
        [(number? xs) xs]
        [(list? (car xs)) (+ (sum (car xs)) (sum (cdr xs)))]
        [#t (sum (cdr xs))]))
```

本地绑定： let, let*, letrec, local define

出于通常的原因，我们需要能够在函数内部定义局部变量。 像ML一样，我们可以在任何地方使用表达式形式来实现这一点。 与ML不同的是，不是只有一种用于局部绑定的结构，而是有四种。 这种多样性很好：不同的情况下使用不同的形式是方便的，并且使用最自然的形式可以向阅读代码的任何人传达有关局部绑定如何相互关联的有用信息。 这种多样性还将帮助我们了解作用域和环境，而不仅仅接受只能有一种let表达式和一种语义的事实。 在环境中查找变量是一种编程语言的基本特性。

首先，有形式的表达式

```
(let ([x1 e1]
      [x2 e2]
      ...
      [xn en])
  e)
```

正如你所预期的那样，这会创建本地变量 x1, x2, ... xn, 绑定到评估结果 e1, e2,..., en.. 然后， e的主体可以使用这些变量（即它们在环境中）， e的结果是整体结果。 从语法上讲，注意在绑定集合周围的“额外”括号和我们使用方括号的常见样式。

但是上面的描述遗漏了一件事：我们用哪个环境来评估 e1, e2, ..., en？ 事实证明，我们使用“之前”的let表达式的环境。 也就是说，后面的变量在它们的环境中没有早期变量。 如果 e3使用 x1或 x2，那要么是一个错误，要么意味着同名的某个外部变量。 这不是ML let表达式的工作方式。 作为一个愚蠢的例子，这个函数将其参数加倍：

```
(定义 (愚蠢的双倍 x)
(让 ([x (+ x 3)]
      [y (+ x 2)])
   (+ x y -5)))
```

这种行为有时很有用。 例如，在某个局部范围内交换 x和 y的含义，可以写成(let ([x y][y x]) ...)。 更常见的是使用 let，在这种情况下，语义上的区别“每个绑定在其环境中都有前面的绑定”并不重要：它传达了表达式是相互独立的。

如果我们在let的位置写上 let*，那么语义上会对每个绑定的表达式在前面绑定产生的环境中进行求值。 这就是ML let表达式的工作方式。 这通常很方便：如果我们只有“常规”的 let，那么我们必须将let表达式嵌套在彼此内部，以便每个后续绑定都在外部let表达式的主体中。 （我们将不得不使用 n 嵌套的 let表达式，每个表达式都有一个绑定，而不是一个具有 n 绑定的 let*。下面是一个使用 let*的示例：）

```
(定义 (愚蠢的双倍 x)
(let* ([x (+ x 3)]
       [y (+ x 2)])
   (+ x y -8)))
```

如上所示，通常在语义上不相关时使用 let而不是 let*是常见的风格。

既不是 let也不是 let*允许递归，因为 e1， e2，...， en不能引用正在定义或之后的绑定。 为了做到这一点，我们有第三个变体 letrec，它允许我们编写：

```
(define (triple x)
  (letrec ([y (+ x 2)]
           [f (lambda (z) (+ z y w x))]
           [w (+ x 7)])
    (f -9)))
```

通常使用 letrec来定义一个或多个（相互）递归函数，例如这个非常慢的取非负数模2的方法：

```
(define (mod2 x)
  (letrec
      ([even? (lambda (x) (if (zero? x) #t (odd? (- x 1))))]
       [odd?  (lambda (x) (if (zero? x) #f (even? (- x 1))))])
    (if (even? x) 0 1)))
```

或者，你可以通过使用局部定义来获得与 letrec相同的行为，这在真实的Racket代码中非常常见，事实上是优先选择的风格，而不是let表达式。 在这门课程中，如果你愿意的话可以使用它，但不是必须的。对于局部定义的使用有一些限制；函数体的开头是一个常见的允许位置。

```
(define (mod2_b x)
  (define even? (lambda(x) (if (zero? x) #t (odd? (- x 1)))))
  (define odd?  (lambda(x) (if (zero? x) #f (even? (- x 1)))))
  (如果 (even? x) 0 1))
```

我们需要小心处理 letrec和局部定义：它们允许代码引用稍后初始化的变量，但每个绑定的表达式仍然按顺序进行评估。

对于相互递归的函数，这从来不是一个问题：在上面的例子中， even?的定义
指的是 odd?的定义尽管绑定到 odd?的表达式还没有被评估。这是可以的，因为在 even?中使用 是在函数体中，所以在 odd?被初始化之前不会被使用。相比之下， letrec的使用是不好的： 这个letrec的使用是不好的：

```
(定义 (bad-letrec x)
 (letrec ([y z]
          [z 13])
   (如果 x y z)))
```

对于 letrec的语义要求，用于初始化 y的 z必须引用 letrec中的 z，但是
z的表达式（ 13）尚未被求值。在这种情况下，Racket在
调用bad-letrec时会引发错误。 （在Racket版本6.1之前，它会将 y绑定到一个特殊的“未定义”对象，这几乎总是隐藏了一个错误。）

对于这门课程，你可以决定是否使用本地定义。讲座材料通常不会这样做，而是选择最方便和最好沟通的 let、 let*或 letrec。 但是欢迎你使用本地定义，其中“相邻的定义”的行为类似于 letrec绑定。

顶层定义

Racket文件是一个带有一系列定义的模块。 就像let表达式一样，使用哪个环境来定义什么非常重要对于语义来说。 在ML中，一个文件就像是一个隐式的 let*。
在Racket中，它基本上就像是一个隐式的 letrec。 这很方便，因为它允许你在模块中按照任何你喜欢的顺序排列函数。 例如，你不需要将相互递归的函数放在一起或使用特殊的语法。 另一方面，还有一些需要注意的新的“陷阱”：

- 你不能使用相同的变量进行两个绑定。 这没有意义：使用哪个变量的使用？ 使用 letrec-类似的语义，如果它们在相同的互递归绑定集合中定义，我们不会使一个变量遮蔽另一个变量。
- 如果一个较早的绑定使用一个较晚的绑定，它需要在函数体中这样做，以便在使用时较晚的绑定已经初始化。在Racket中，使用未初始化的值的“不好”情况会在使用模块时引发错误（例如，在DrRacket中单击“运行”文件时）。
- 因此，在一个模块/文件中，没有顶级遮蔽（您仍然可以在定义或*let*-表达式中进行遮蔽），但一个模块可以遮蔽另一个文件中的绑定，例如从Racket的标准库中隐式包含的文件。 例如，虽然这是不好的风格，但我们可以用我们自己的函数遮蔽内置的*list*函数。我们自己的函数甚至可以像任何其他递归函数一样递归调用自己。 然而，REPL中的行为是不同的，因此不要在REPL中用自己的递归函数定义遮蔽一个函数。在定义窗口中定义递归函数并在REPL中使用它仍然按预期工作。

绑定通常是可变的： set！存在

虽然Racket鼓励使用闭包和避免副作用的函数式编程风格，
但事实上它有赋值语句。 如果 x在你的环境中，那么(set! x 13)会改变这个

绑定，使得 x现在映射到值13。 这样做会影响所有具有这个 x的环境的代码。
发音为“set-bang”，感叹号是一种约定，用于提醒代码读者副作用
正在发生，可能会影响其他代码。以下是一个例子：

```
(定义 b 3)
(定义 f (lambda (x) (* 1 (+ x b))))
(定义 c (+ b 4))
(设定! b 5)
(定义 z (f 4))
(定义 w c)
```

在评估这个程序之后， z被绑定为9，因为当评估时，绑定到 f的函数体将查找 b并找到 5。 然而， w被绑定为 7，因为当我们评估(定义 c (+ b 4))时，我们发现 b是 3，通常情况下，结果是将 c绑定为 7，无论我们如何得到 7。 所以当我们评估(定义 w c)时，我们得到 7；不管 b是否改变，这都是无关紧要的。

你也可以使用 set！来定义局部变量，同样的推理也适用：当你查找一个变量时，你必须考虑
何时you查找一个变量以确定你得到的值。但是，习惯于具有
赋值语句的语言的程序员对此已经习以为常。

改变顶层绑定尤其令人担忧，因为我们可能不知道正在使用定义的所有代码。 例如，我们上面的函数 f使用了 b，并且如果b被改变为一个意外的值，它可能会表现出奇怪的行为，甚至会失败。 如果 f需要防止这种可能性，它需要在 b可能改变之前避免使用 b。 在软件开发中，有一种常用的技术你应该了解：如果某个东西可能被改变并且你需要旧值，在发生改变之前先进行复制。 在Racket中，我们可以很容易地编写出这个代码：

```
(定义 f
   (让 ([b b])
      (lambda (x) (* 1 (+ x b)))))
```

这段代码使函数体中的 b引用一个被初始化为全局 b的局部 b。

但是这样的防御是否足够呢？ 由于 *和 +只是绑定到函数的变量，我们可能需要防止它们在以后被改变：

```
(定义 f
   (let ([b b]
         [+ +]
         [* *])
      (lambda (x) (* 1 (+ x b)))))
```

如果 f使用其他辅助函数，情况会变得更糟：除非这些函数也复制了它们的所有辅助函数，否则仅仅复制绑定到函数的变量是不够的。

幸运的是，在Racket中这一切都是不必要的，因为有一个合理的妥协：除非定义它的模块包含一个 set!，否则顶层绑定是不可变的。所以如果包含(define b 4)的文件没有改变它的 set!，那么我们可以放心地说没有其他文件可以使用 set!来改变这个绑定（这将导致错误）。 而且所有预定义的函数如 +和 *都在一个不使用 set!的模块中，所以它们也不能被改变。 （在Scheme中，所有的顶层绑定都是可变的，但程序员通常假设它们不会被改变，因为假设它们会被改变太痛苦了。）

所以之前的讨论不会影响你的大部分Racket编程，但了解set！的含义以及通过复制来防止变异是很有用的。 关键是，Racket通常避免变异的可能性，这使得编写正确的代码非常困难。

有关 cons的真相

到目前为止，我们已经使用了 cons, null, car, cdr和null?来创建和访问列表。 例如，(cons 14 (cons #t null))创建了列表'(14 #t)其中引号字符表示这是打印一个列表值，而不是表示一个（错误的）函数调用到14。

但事实上*cons*只是创建一个对，其中你可以用*car*获取第一个部分，用cdr获取第二个部分。这样的对在像Racket这样的语言中通常被称为*cons*单元。所以我们可以写(*cons* (+ 7 7) #t)来生成对'(14 . #t)，其中的句点表示这不是一个列表。根据约定和list?预定义函数，列表要么是*n*ull，要么是一个cdr（即第二个组件）是列表的对。如果*cons*单元不是一个列表，通常被称为不正常的列表，特别是如果它在第二个位置有嵌套的*cons*单元，例如(*cons* 1 (*cons* 2 (*cons* 3 4)))，其中的结果打印为'(1 2 3 . 4)。

大多数列表函数，如 length如果传递了不正确的列表，将会产生运行时错误。 另一方面，内置的pair?原语对于使用 cons构建的任何内容（即任何不正确或正确的列表除了空列表）都返回true。

不正确的列表有什么好处？ 真正的重点是，对于构建具有多个部分的每个类型，即一些通用的有用的方法。 在动态类型语言中，列表所需的只是对列表末尾的识别方式，按照惯例Racket使用 null常量（打印为'()）。从风格上讲，您应该使用正确的列表而不是不正确的列表来处理可能具有任意数量元素的集合。

Cons单元是不可变的，但是有 mcons

Cons单元是不可变的：当您创建一个cons单元时，其两个字段被初始化并且永远不会改变。（这是Racket和Scheme之间的一个重要区别。）因此，我们可以继续享受知道cons单元不能被程序中的其他代码改变的好处。 它还有另一个有些微妙的优势：Racket实现可以聪明地使list?成为一个常数时间操作，因为它可以在创建cons单元时存储每个cons单元是否是一个正确的列表。 如果cons单元是可变的，这是行不通的，因为列表中的一个远处的变化可能会将其变成一个不正确的列表。

要意识到cons单元格实际上是不可变的，即使我们有set！。考虑这段代码：

```
(定义 x (cons 14 null))
(定义 y x)
(set! x (cons 42 null))
(定义 fourteen (car y))
```

对于 x的 set！会改变 x绑定的内容为不同的对；它不会改变旧对的内容，旧对是由 x引用的。你可能尝试做类似于(set! (car x) 27)的操作，但这是一个语法错误： set！需要一个变量来赋值，而不是其他类型的位置。

然而，如果我们想要可变的对，Racket很乐意提供一组不同的原语：

- mcons 创建一个可变对

- mcar返回可变对的第一个组件
- mcdr返回可变对的第二个组件
- mpair？如果给定一个可变对，返回 #t
- set-mcar！接受一个可变对和一个表达式，并将第一个组件更改为表达式的结果
- set-mcdr！接受一个可变对和一个表达式，并将第二个组件更改为表达式的结果

由于我们将要学习的一些强大的习语使用变异来存储先前计算的结果，我们将会发现可变对很有用。

延迟评估和thunks简介

语言构造的一个关键语义问题是其子表达式何时被评估。 例如，在Racket（以及类似的ML和大多数但不是所有的编程语言中），给定（e1 e2 ... en）我们在执行函数体之前只评估一次函数参数 e2，...， en，并给定一个函数（lam*bd*a (...) ...）我们在调用函数之前不评估函数体。 我们可以对比这个规则

（“提前评估参数”）与（如果e1 e2 e3）的工作方式：我们不评估 e2和 e3。
这就是为什么：

```
(定义 (my-if-bad x y z) (如果 x y z))
```

是一个函数，不能在任何地方使用if表达式；对于求值子表达式的规则是根本不同的。 例如，这个函数永远不会终止，因为每次调用都会进行递归调用：

```
(定义 (factorial-wrong x)
 (my-if-bad (= x 0)
            1
            (* x (factorial-wrong (- x 1)))))
```

然而，我们可以利用函数体在函数被调用之前不会被求值的事实，来创建一个更有用的“if函数”版本：

```
(定义 (my-if x y z) (如果 x (y) (z)))
```

现在，无论我们在哪里写(if e1 e2 e3)，我们都可以写成(my-if e1 (lambda () e2) (lambda () e3))。
my-if的函数体要么调用绑定到y的零参数函数，要么调用绑定到z的零参数函数。 因此，这个函数是正确的（对于非负参数）：

```
(定义 (阶乘 x)
  (我的-if (= x 0)
          (lambda () 1)
          (lambda () (* x (阶乘 (- x 1))))))
```

虽然没有理由将Racket的“if”包装在这种方式中，但是使用零-
参数函数来延迟评估（现在不评估表达式，而是在零-
参数函数被调用时再评估）的一般习惯用法非常强大。 作为方便的术语/行话，当我们使用零-
参数函数来延迟评估时，我们称该函数为一个 *thunk*。 你甚至可以说，“thunk the argument”
意思是“使用`(lambda () e)`而不是 `e`”。

使用thunks是一种强大的编程习惯。 这不是特定于Racket的——我们可以在ML中同样地研究这种编程。

使用延迟和强制进行惰性评估

假设我们有一个大的计算，我们知道如何执行，但我们不知道是否需要执行它。程序的其他部分知道计算结果所需的位置，可能有0个、1个或多个不同的位置。 如果我们使用thunk，那么我们可能会重复执行大的计算多次。 但是如果我们不使用thunk，即使我们不需要，我们也会执行大的计算。为了得到“两全其美”，我们可以使用一种编程习惯，有几个不同的名称（可能在技术上略有不同）：惰性评估、按需调用、promises。 这个想法是使用变异来记住我们第一次使用thunk时的结果，这样我们就不需要再次使用thunk了。

Racket中的一个简单实现是：

```
(定义 (my-delay f)
  (mcons #f f))

(定义 (my-force th)
  (如果 (mcar th)
      (mcdr th)
      (开始 (set-mcar! th #t)
            (set-mcdr! th ((mcdr th)))
            (mcdr th))))
```

我们可以创建一个thunk `f`并将其传递给 `my-delay`。 这将返回一个pair，其中第一个字段表示我们尚未使用thunk。然后，如果 `my-force`发现thunk尚未使用，则使用它，然后使用mutation将pair更改为保存使用thunk的结果。 这样，对于相同的pair，任何将来对`my-force`的调用都不会重复计算。 具有讽刺意味的是，尽管我们在实现中使用了mutation，但如果传递给 `my-delay`的thunk具有副作用或依赖于可变数据，那么这种习惯用法可能会出现错误，因为这些效果最多只会发生一次，并且很难确定第一次调用 `my-force`将发生的时间。

考虑这个愚蠢的例子，我们想要通过递归算法来乘以两个表达式的结果 `e1`和 `e2`（当然，实际上你只需要使用 `*`，而且如果 `e1`产生一个负数，这个算法是不起作用的）：

```
(定义 (my-mult x y)
 (条件 [(= x 0) 0]
       [(= x 1) y]
       [#t (+ y (my-mult (- x 1) y))]))
```

现在调用`(my-mult e1 e2)`会对 `e1`和 `e2`进行一次求值，然后进行0次或多次相加。 但是如果 `e1`求值为0，而 `e2`计算时间很长呢？ 那么求值 `e2`就是浪费的。所以我们可以将其封装为thunk：

```
(定义 (my-mult x y-thunk)
 (条件 [(= x 0) 0]
       [(= x 1) (y-thunk)]
       [#t (+ (y-thunk) (my-mult (- x 1) y-thunk))]))
```

现在我们将调用(my-mult e1 (lambda () e2)). 如果 e1的计算结果为0，那么这个方法非常好，如果 e1的计算结果为1，那么这个方法还可以，但是如果 e1的计算结果为一个很大的数，那么这个方法就非常糟糕。 毕竟，现在我们在每次递归调用时都要计算 e2。 所以让我们使用 my-delay和 my-force来兼顾两者的优点：

```
(my-mult e1 (let ([x (my-delay (lambda () e2))]) (lambda () (my-force x))))
```

请注意，在调用 my-mult之前，我们只创建了一次延迟计算，然后第一次调用传递给 my-mult的thunk被调用时， my-force将计算 e2并记住结果以供将来调用my-force x时使用。 一个看起来更简单的替代方法是将 my-mult重写为期望从 my-delay获得结果而不是任意的thunk：

```
(定义 (my-mult x y-promise)
 (条件 [(= x 0) 0]
       [(= x 1) (my-force y-promise)]
       [#t (+ (my-force y-promise) (my-mult (- x 1) y-promise))]))

(my-mult e1 (my-delay (lambda () e2)))
```

一些语言，尤其是Haskell，对所有函数调用都使用这种方法，即函数调用的语义在这些语言中是不同的：如果一个参数从未被使用，则不会被计算，否则只计算一次。 这被称为需要调用而我们将使用的所有语言都是值调用（在调用之前完全计算参数）。

流

流是一个无限序列的值。 我们显然无法显式地创建这样的序列（它将永远持续下去），但我们可以创建知道如何生成无限序列的代码和其他知道如何请求所需序列的代码。

在计算机科学中，流非常常见。 你可以将同步电路产生的比特序列视为一个流，每个时钟周期产生一个值。 电路不知道应该运行多长时间，但它可以无限产生新的值。 UNIX管道（cmd1 | cmd2）是一个流；它使得 cmd1只产生 cmd2所需的输出。 对于在网页上对用户点击做出反应的Web程序来说，可以将用户的活动视为一个流 - 不知道下一个活动何时到达或有多少个，但准备做出适当的响应。 更一般地说，流可以是一种方便的分工方式：软件的一部分知道如何产生无限序列中的连续值，但不知道需要多少个值和/或如何处理它们。 另一部分可以确定需要多少个值，但不知道如何生成它们。

有许多编码流的方法；我们将采用简单的方法，将流表示为一个惰性求值函数，当调用时会产生一个包含（1）序列中的第一个元素和（2）表示第二个到无穷元素的流的惰性求值函数。 定义这样的惰性求值函数通常使用递归。以下是三个示例：

```
(定义 ones (lambda () (cons 1 ones)))
```

```
(定义 nats
  (letrec ([f (lambda (x) (cons x (lambda () (f (+ x 1)))))])
    (lambda () (f 1))))
(定义 powers-of-two
  (letrec ([f (lambda (x) (cons x (lambda () (f (* x 2)))))])
    (lambda () (f 2))))
```

给定流的编码和一个流 s，我们可以通过(car (s))获取第一个元素，通过(car ((cdr (s))))获取第二个元素，通过(car ((cdr ((cdr (s))))))获取第三个元素，以此类推。记住括号的重要性： (e)调用thunk(e)。

我们可以编写一个高阶函数，它接受一个流和一个谓词函数，并返回在谓词函数返回true之前产生了多少个流元素：

```
 (定义 (number-until stream tester)
  (letrec ([f (lambda (stream ans) )
               (let ([pr (stream)])
                  (如果 (tester (car pr) )
                      ans
                       (f (cdr pr) (+ ans 1) ) ) ) ) ])
    (f流1) ) )
```

作为一个例子， (number-until powers-of-two (lambda (x) (= x 16))) 计算为4。

作为一个附注，以上所有的流都可以在最多给出前一个元素的情况下产生它们的下一个元素。
因此，我们可以使用高阶函数来抽象出这些函数的共同方面，这样我们可以将流创建逻辑放在一个地方，将特定流的细节放在另一个地方。 这只是使用高阶函数重用常见功能的另一个例子：

```
(define (stream-maker fn arg)
  (letrec ([f (lambda (x)
                (cons x (lambda () (f (fn x arg)))))])
    (lambda () (f arg))))
(define ones (stream-maker (lambda (x y) 1) 1))
(定义 nats (stream-maker + 1))
(定义 powers-of-two (stream-maker * 2))
```

记忆化

与延迟求值相关的习语，实际上不使用thunks的是记忆化。 如果一个函数没有副作用，那么如果我们多次使用相同的参数调用它，我们实际上不需要多次调用。 相反，我们可以查找第一次使用参数调用函数时的答案。

这是否是一个好主意取决于权衡。 在表中保留旧答案会占用空间，并且表查找确实需要一些时间，但与重新执行昂贵的计算相比，这可能是一个巨大的优势。再次强调，为了使这种技术正确，给定相同的参数，函数将始终返回相同的结果，并且没有副作用。 因此，能够使用这个记忆表（即进行记忆化）是避免突变的另一个优势。

为了实现记忆化，我们使用了变异：每当函数被调用时，我们计算答案然后将结果添加到表中（通过变异）。

作为一个例子，让我们考虑一个函数的3个版本，它接受一个 x并返回斐波那契数（x）。（斐波那契数是一个众所周知的定义，在建模人口等方面非常有用。）一个简单的递归定义是：

```
（定义（斐波那契 x）
  （如果（或（= x 1）（= x 2））
      1
      （+（斐波那契（- x 1））
         （斐波那契（- x 2））））
```

不幸的是，这个函数运行时间呈指数增长。我们可能会注意到（斐波那契 30）的暂停，而（斐波那契 40）比这长一千倍，（斐波那契 10000）需要的时间比宇宙中的粒子还要多。现在，我们可以通过采取“递增计数”的方法来修复这个问题，记住以前的答案：

```
（定义（斐波那契 x）
  （letrec （[f （lambda （acc1 acc2 y）
                （if （= y x）
                    （+ acc1 acc2）
                    （f （+ acc1 acc2） acc1 （+ y 1））））]）
    （if （or （= x 1） （= x 2））
        1
        （f 1 1 3））））
```

这需要线性时间，所以斐波那契数列 10000几乎立即返回（并且返回一个非常大的数），但它对问题需要一种完全不同的方法。通过记忆化，我们可以将斐波那契数列转化为一种对很多算法都有效的技术。它与“动态规划”密切相关，这是你在高级算法课程中经常学到的。这是进行记忆化的版本（下面将介绍 assoc库函数）：

```
(define 斐波那契数列
  (letrec([memo null]
          [f (lambda (x)
               (let ([ans (assoc x memo)])
                 (if ans
                     (cdr ans)
                     (让 ([new-ans (if (or (= x 1) (= x 2))
                                       1
                                       (+ (f (- x 1))
                                          (f (- x 2))))])
                       (begin
                         (set! memo (cons (cons x new-ans) memo))
                         new-ans)))))])
    f))
```

不同的调用 f使用相同的可变记忆表是至关重要的：如果我们在调用内部创建表，那么每次调用都会使用一个新的空表，这是没有意义的。但我们不会将表放在顶层，因为那样做是不好的风格，因为它的存在只应该为斐波那契的实现所知。

为什么这种技术可以使(fibonacci 10000)快速完成？因为当我们在任何递归调用中评估(f (- x 2))时，结果已经在表中，所以不再有指数级的递归调用。

递归调用的数量。 这比调用(fibonacci 10000)第二次更快完成更重要，因为答案将在记忆表中。

对于一个大表，使用一个列表和Racket的 assoc函数可能是一个糟糕的选择，但对于演示记忆化概念来说是可以的。assoc只是Racket中的一个库函数，它接受一个值和一个由对组成的列表，并返回列表中第一个对的car等于该值的对。 如果没有对的car等于该值，则返回 #f。 （ assoc返回对而不是对的cdr的原因是为了区分没有匹配的对和匹配的对的cdr中有 #f的情况。这是我们在ML中使用选项的一种情况。）

宏：关键点

本单元的最后一个主题是宏，它通过让程序员定义自己的语法糖来增加语言的语法。 在学习如何在Racket中定义宏之前，我们先从关键思想开始。

宏定义引入了一些新的语法到语言中。 它描述了如何将新的语法转换为语言本身中的不同语法。 宏系统是一种用于定义宏的语言（或更大语言的一部分）。 宏使用只是使用先前定义的宏之一。 宏使用的语义是用宏定义所定义的适当语法替换宏使用。 这个过程通常被称为宏展开，因为语法转换通常会产生更多的代码。

关键点是宏展开发生在我们学到的任何其他事情之前：在类型检查之前，在编译之前，在评估之前。 将“展开所有宏”视为在任何其他事情发生之前对整个程序进行预处理。 因此，宏在函数体中扩展，条件语句的两个分支中等等。

下面是三个Racket中可能定义的宏的例子：

- 一个宏，使程序员可以写(my-if e1 then e2 else e3)其中 my-if, then, 和 else 是关键字，这个宏展开为(if e1 e2 e3)。
- 一个宏，使程序员可以写(comment-out e1 e2)并将其转换为 e2，即，这是一种方便的方法将表达式 e1从程序中移除（用 e2替换），而不实际删除任何内容。
- 一个宏，使程序员可以写(my-delay e)并将其转换为(mcons #f (lambda () e))。 这与我们之前定义的 my-delay函数不同，因为函数要求调用者传入一个thunk。 在这里，宏展开完成了thunk的创建，宏用户不应该包含显式的thunk。

Racket拥有出色且复杂的宏系统。 出于精确和技术原因，它的宏系统优于许多更为知名的宏系统，特别是C或C++中的预处理器。因此，我们可以使用Racket来了解一般宏的一些陷阱。下面我们讨论：

- 宏系统必须处理标记化、括号化和作用域的问题 - 以及Racket如何比C/C++更好地处理括号化和作用域
- 如何在Racket中定义宏，例如上面描述的宏
- 宏定义应该注意表达式的求值顺序和求值次数
- 宏中的变量绑定问题和卫生概念的关键问题

标记化、括号化和作用域

宏的定义和宏展开比在文本编辑器中进行的“查找和替换”或手动编写脚本来执行程序中的某些字符串替换更加结构化和微妙。 宏展开在大致上有三个不同之处。

首先，考虑一个宏，它“将每个使用 head替换为 car”。在宏系统中，这并不意味着某个变量 headt会被重写为 cart。 因此，宏的实现至少要理解编程语言的文本是如何被分解成 *tokens*（即单词）的。 这个tokens的概念在不同的语言中是不同的。 例如，在大多数语言中， a-b会被分解成三个tokens（一个变量，一个减法，和另一个变量），但在Racket中是一个token。

其次，我们可以问宏是否理解括号的使用。 例如，在C/C++中，如果你有一个宏

```
#define ADD(x,y) x+y
```

那么ADD(1,2/3)*4会被重写为1 + 2 / 3 * 4，这与(1 + 2/3)*4不是同一件事。 因此，在这样的语言中，宏的编写者通常在宏定义中包含大量的显式括号，例如

```
#define ADD(x,y) ((x)+(y))
```

在Racket中，宏展开保留了代码结构，因此这个问题不是一个问题。 Racket的宏使用总是像这样(x ...)其中 x是宏的名称，展开的结果“保持在相同的括号内”（例如，(my-if x then y else z)可能展开为(if x y z)）。 这是Racket简洁一致语法的一个优点。

第三，我们可以问宏展开是否会在创建变量绑定时发生。 如果不会，那么局部变量可以屏蔽宏，这可能是你想要的。 例如，假设我们有：

```
(let ([hd 0] [car 1]) hd) ; 评估为0
(let* ([hd 0] [car 1]) hd) ; 评估为0
```

如果我们将 car替换为 hd，那么第一个表达式将会出错（尝试绑定两次 hd），而第二个表达式现在评估为1。 在Racket中，宏展开不适用于变量定义，即上面的car是不同的，并且会屏蔽任何可能在作用域内的 car宏。

使用define-syntax定义宏

现在让我们来介绍一下在Racket中定义宏的语法（多年来，Racket的前身Scheme有很多变种；这是我们将使用的一种现代方法）。下面是一个宏的例子，它允许用户使用任意表达式 e1, e2和 e3来编写(my-if e1 then e2 else e3)，并且它的意思完全等同于(if e1 e2 e3)：

```
(define-syntax my-if
  (syntax-rules (then else)
    [(my-if e1 then e2 else e3)
     (if e1 e2 e3)]))
```

- define-syntax是用于定义宏的特殊形式。

- my-if是我们宏的名称。 它将 my-if添加到环境中，以便形如(my-if ...)的表达式可以根据宏定义的其余语法规则进行宏展开。
- syntax-rules是一个关键字。
- 下一个括号列表（在这种情况下（then else））是这个宏的“关键字”列表，即，在 my-if的主体中使用 then或 else是语法，而不在这个列表中的任何内容（和不是my-if本身）都表示任意表达式。
- 剩下的是一对对： my-if可能被使用的方式以及如果以这种方式使用它应该如何重写。
- 在这个例子中，我们的列表只有一个选项： my-if必须在形式为（my-if e1 then e2 else e3）的表达式中使用，并且变成(if e1 e2 e3)。 否则会导致错误。 注意：重写发生在对表达式 e1， e2或 e3进行任何评估之前，与函数不同。 这是我们对 my-if这样的条件表达式所希望的。

这是第二个简单的例子，我们在其中使用宏来“注释掉”一个表达式。 我们使用(comment-out e1 e2)来重写为 e2，意味着 e1将永远不会被评估。 这在调试代码时可能比实际使用注释更方便。

```
(define-syntax comment-out
  (syntax-rules ()
    [(comment-out e1 e2) e2]))
```

我们的第三个例子是一个宏 my-delay，与之前定义的 my-delay函数不同，用户将使用(my-delay e)来创建一个承诺，使得 my-force将评估 e并记住结果，而不是用户编写(my-delay (lambda () e))。 只有宏，而不是函数，可以通过添加一个thunk来“延迟评估”，因为函数调用总是评估它们的参数。

```
(define-syntax my-delay
  (syntax-rules ()
    [(my-delay e)
     (mcons #f (lambda () e))]))
```

我们不应该创建 my-force的宏版本，因为我们之前的函数版本正是我们想要的。 给定(my-force e)，我们确实希望评估 e为一个值，这个值应该是由 my-delay创建的mcons-cell，然后在 my-force函数中执行计算。 定义一个宏没有任何好处，而且可能容易出错。 考虑这个糟糕的尝试：

```
(define-syntax my-force
  (syntax-rules ()
     [(my-force e)
      (if (mcar e)
          (mcdr e)
          (begin (set-mcar! e #t)
                 (set-mcdr! e ((mcdr e)))
                 (mcdr e)))]))
```

由于宏展开，使用这个宏的参数将会被多次计算，如果参数具有副作用，可能会产生奇怪的行为。 宏的使用者不会预料到这一点。在以下代码中：

```
(let ([t (my-delay some-complicated-expression)])
  (my-force t))
```

这并不重要，因为t已经绑定到一个值，但在以下代码中：

```
(my-force (begin (print "hi") (my-delay some-complicated-expression)))
```

我们会打印多次。 请记住，宏展开会将整个参数复制到宏定义的每个位置，但我们通常只希望它被计算一次。 这个版本的宏在这方面做得很好：

```
(define-syntax my-force
  (syntax-rules ()
    [(my-force e)
     (let ([x e])
       (if (mcar x)
           (mcdr x)
           (begin (set-mcar! x #t)
                  (set-mcdr! x ((mcdr x)))
                  (mcdr x))))]))
```

但是，再次强调，没有理由定义这样的宏，因为函数完全可以满足我们的需求。只需要使用：

```
(定义 (my-force th)
 (如果 (mcar th)
     (mcdr th)
     (开始 (set-mcar! th #t)
           (set-mcdr! th ((mcdr th)))
           (mcdr th))))
```

变量、宏和卫生

让我们考虑一个将其参数加倍的宏。 请注意，这是一种不好的风格，因为如果你想要将参数加倍，你应该直接写一个函数：(define (double x) (* 2 x)) or(define (double x) (+ x x))这两者是等价的。 但是这个简短的例子将让我们研究宏参数在何时以及在什么环境中被求值，所以我们将它作为一个不好的风格的例子。

这两个宏是不等价的：

```
(define-syntax double1
  (syntax-rules ()
         [(double1 e)
          (* 2 e)]))
(define-syntax double2
  (syntax-rules ()
    [(double2 e)
     (+ e e)]))
```

原因是 double2会对其参数进行两次求值。 所以(double1 (begin (print "hi") 17))只打印一次"hi"，但(double2 (begin (print "hi") 17))会打印两次"hi"。 函数版本的打印"hi"一次，因为函数参数在调用函数之前被求值为值。

为了修复 double2而不“改变算法”为乘法而不是加法，我们应该使用一个局部变量：

```
(define-syntax double3
  (syntax-rules ()
    [(double3 e)
     (let ([x e])
       (+ x x))]))
```

在宏定义中使用局部变量来控制是否/何时对表达式进行求值，这正是你应该做的，但在较弱的宏语言中（再次，C/C++是一个容易被嘲笑的目标），通常避免使用局部变量。 原因与作用域和所谓的卫生有关。 为了举例说明，考虑这个愚蠢的变体 double3：

```
(define-syntax double4
  (syntax-rules ()
    [(double4 e)
     (let* ([zero 0]
            [x e])
       (+ x x zero))]))
```

在Racket中，这个宏总是按预期工作，但这可能/应该让你感到惊讶。毕竟，假设我有
这个用法：

```
(let ([zero 17])
   (double4 zero))
```

如果按预期进行语法重写，你将得到

```
(let ([zero 17])
  (let* ([zero 0]
         [x zero])
       (+ x x zero)))
```

但是这个表达式的值为0，而不是34。 问题是宏使用中的一个自由变量（在(double4 zero)中的 zero）最终进入了宏定义的局部变量的作用域。 这就是为什么在C/C++中，宏定义中的局部变量往往具有像__x_hopefully_no_conflict这样的有趣名称，希望这种情况不会发生。 在Racket中，宏展开的规则更复杂，以避免这个问题。 基本上，每次使用宏时，它的所有局部变量都会被重写为新的变量名，这些变量名与程序中的其他任何内容都不冲突。 这是按照定义使Racket宏具有卫生性的“一半”。

另一半与宏定义中的自由变量有关，确保它们不会错误地出现在宏使用的某个局部变量的作用域中。 例如，考虑这段使用 double3的奇怪代码：

```
(let ([+ *])
   (double3 17))
```

朴素的重写会产生：

```
(let ([+ *])
  (let ([x 17])
     (+ 17 17)))
```

然而，这会产生17^2，而不是34。 再次强调，朴素的重写不是Racket所做的。 宏定义中的自由变量总是指的是宏定义所在环境中的内容，而不是宏使用的环境。 这使得编写始终按预期工作的宏变得更加容易。 再次强调，C/C++中的宏工作方式类似于朴素的重写。

有些情况下，你不希望使用卫生宏。 例如，假设你想要一个用于for循环的宏，宏用户可以指定一个变量来保存循环索引，而宏定义者则确保该变量在每次循环迭代中都保持正确的值。 Racket的宏系统有一种方法可以实现这一点，它涉及明确违反卫生宏的规则，但我们在这里不会演示。

更多宏示例

最后，让我们考虑一些更有用的宏定义，包括使用多个情况进行重写的宏定义。 首先，这是一个宏，它允许您使用 `let*`语义来编写最多两个let绑定，但是使用更少的括号：

```
(define-syntax let2
  (syntax-rules ()
    [(let2 () body)
     body]
    [(let2 (var val) body)
     (let ([var val]) body)]
    [(let2 (var1 val1 var2 val2) body)
     (let ([var1 val1])
       (let ([var2 val2])
         body))]))
```

例如，`(let2 () 4)`的结果是4，`(let2 (x 5) (+ x 4)`的结果是9，`(let2 (x 5 y 6) (+ x y))`的结果是11。

事实上，如果支持递归宏，我们可以完全使用let来重新定义Racket的 `let*`。 我们需要一种方法来讨论“语法列表的其余部分”，Racket的 `...`给了我们这个功能：

```
(define-syntax my-let*
  (syntax-rules ()
    [(my-let* () body)
     body]
    [(my-let* ([var0 val0]
               [var-rest val-rest] ...)
              body)
     (let ([var0 val0])
       (my-let* ([var-rest val-rest] ...)
                body))]))
```

由于宏是递归的，没有任何东西可以阻止您在宏展开期间（即在代码运行之前）生成无限循环或无限数量的语法。 上面的示例没有这样做，因为它在一个较短的绑定列表上进行递归。

最后，这是一个用于有限形式的for循环的宏，它执行它的body $hi - lo$次。 （它是有限的，因为body没有给出当前迭代号。）请注意使用let表达式确保我们只评估一次lo和hi，但我们正确评估body的次数。

```
(define-syntax for
  (syntax-rules (to do)
    [(for lo to hi do body)
     (let ([l lo]
           [h hi])
       (letrec ([loop (lambda (it)
                        (if (> it h)
                            #t
                            (begin body (loop (+ it 1)))))])
         (loop l)))]))
```

CSE341：程序设计语言 2016年春季
第6单元总结

标准描述：这个总结大致涵盖了课堂和复习部分的内容。 当回顾材料时，以叙述方式阅读材料并将整个单元的材料放在一个文档中可能会有所帮助。 请报告这些笔记中的错误，甚至是打字错误。 这个总结不能完全替代上课、阅读相关代码等。

目录

无数据类型的数据编程

在ML中，我们使用数据类型绑定来定义我们自己的一种类型，包括用于树状数据的递归数据类型，例如算术表达式的小语言。 数据类型绑定引入了一个新类型到静态环境中，同时引入了用于创建该类型数据的构造函数和用于模式匹配的函数

使用数据类型。 Racket作为一种动态类型语言，没有直接对应于数据类型绑定的东西，但它支持相同类型的数据定义和编程。

首先，在ML中我们需要数据类型的一些情况在Racket中更简单，因为我们可以使用动态类型将任何类型的数据放在任何地方。 例如，我们知道在ML中列表是多态的，但是任何特定的列表必须具有相同类型的元素。 所以我们不能直接构建一个包含“字符串或整数”的列表。相反，我们可以定义一个数据类型来绕过这个限制，就像这个例子中一样：

```
数据类型int_or_string = I of int | S of string

fun funny_sum xs =
     情况xs的
        [] => 0
      | (I i)::xs' => i + funny_sum xs'
      | (S s)::xs' => String.size s + funny_sum xs'
```

在Racket中，不需要这样的解决办法，因为我们可以编写适用于元素为数字或字符串的列表的函数：

```
(定义 (有趣的和 xs)
(cond [(null? xs) 0]
      [(数字? (第一个 xs)) (+ (第一个 xs) (有趣的和 (剩下的 xs)))]
      [(字符串? (第一个 xs)) (+ (字符串长度 (第一个 xs)) (有趣的和 (剩下的 xs)))]
      [#t (错误 "期望数字或字符串")]))
```

这种方法的关键是Racket具有像null?, number?和 string?这样的内置原语用于在运行时测试数据类型。

但对于像这个ML定义的递归数据类型，如算术表达式：

```
数据类型 exp = Const of int | Negate of exp | Add of exp * exp | Multiply of exp * exp
```

将我们的编程习惯适应到Racket中将更有趣。

我们首先考虑一个评估类型为 exp的函数，但这个函数的返回类型与我们之前在课程中编写的类似函数不同。 然后我们将考虑两个
在Racket中，有不同的方法来定义和使用这种算术表达式的“类型”。我们将会论证第二种方法更好，但是第一种方法对于理解Racket总体和第二种方法特别重要。

改变我们评估算术表达式数据类型的方式

最明显的函数是接受ML数据类型 exp的值，并计算算术表达式并返回结果。 之前我们像这样编写了这样一个函数：

```
fun eval_exp_old e =
    case e of
        Const i => i
      | Negate e2   => ~ (eval_exp_old e2)
      | Add(e1,e2) => (eval_exp_old e1) + (eval_exp_old e2)
      | Multiply(e1,e2) => (eval_exp_old e1) * (eval_exp_old e2)
```

eval_exp_old的类型是exp -> int。 特别地，返回类型是int，一个ML整数，我们可以使用ML的算术运算符进行加法、乘法等操作。

在本课程单元的其余部分中，我们将使用这种类型的函数来返回一个表达式，所以ML类型将变为exp -> exp。 调用的结果（包括递归调用）将具有形式Const i，其中i是一个整数，例如Const 17。 调用者必须检查 exp返回的类型确实是一个Const，提取底层数据（在ML中使用模式匹配），然后根据需要使用 Const构造函数返回一个 exp。 对于我们的小型算术语言，这种方法会导致一个相对复杂的程序：

```
异常 错误 of string
fun eval_exp_new e =
    let
        fun get_int e =
            case e of
                Const i => i
              | _ => raise (错误 "期望常量结果")
    in
        case e of
            Const _ => e（注意我们在这里返回整个exp）
          | Negate e2   => Const (~ (get_int (eval_exp_new e2)))
          | Add(e1,e2) => Const ((get_int (eval_exp_new e1)) + (get_int (eval_exp_new e2)))
          | Multiply(e1,e2) => Const ((get_int (eval_exp_new e1)) * (get_int (eval_exp_new e2)))
    end
```

这种额外的复杂性对于我们简单的类型 exp几乎没有好处，但我们这样做是有一个非常好的原因：很快我们将定义具有多种结果类型的小语言。 假设计算的结果不必是一个数字，因为它也可以是一个布尔值、一个字符串、一个对、一个函数闭包等等。那么我们的 eval_exp函数需要返回某种一种类型的子集，使用 exp类型定义的可能性将满足我们的需求。 然后，像加法这样的 eval_exp的情况将需要检查递归结果是否是正确的值类型。 如果这个检查不成功，那么上面的 get_int行将被执行（而对于我们目前的简单示例，异常将永远不会被引发）。

通过Racket列表递归数据类型

在我们能够编写类似于上面的MLeval_exp_new函数的Racket函数之前，我们需要定义
算术表达式本身。 我们需要一种方法来构造常量、否定、加法和
乘法，一种方法来测试我们拥有的表达式的类型（例如，“它是一个加法吗？”），以及一种方法来
访问这些部分（例如，“获取加法的第一个子表达式”）。 在ML中，数据类型绑定给了我们
所有这些。

在Racket中，动态类型让我们可以使用列表来表示任何类型的数据，包括算术表达式。
一种足够的惯用法是使用第一个列表元素来指示“它是什么类型的东西”，并使用后续列表
元素来保存底层数据。 通过这种方法，我们可以自己定义Racket函数来
构造、测试和访问：

```
; 用于构建的辅助函数
(define (Const i) (list 'Const i))
(定义 (否定 e) (列表 '否定 e))
(定义 (加 e1 e2) (列表 '加 e1 e2))
```

```
(define (Multiply e1 e2) (list 'Multiply e1 e2))
; 用于测试的辅助函数
(define (Const? x) (eq? (car x) 'Const))
(定义 (否定? x) (eq? (car x) '否定))
(定义 (加? x) (eq? (car x) '加))
(define (Multiply? x) (eq? (car x) 'Multiply))
; 用于访问的辅助函数
(define (Const-int e) (car (cdr e)))
(定义 (否定-e e) (car (cdr e)))
(定义 (加-e1 e) (car (cdr e)))
(定义 (加-e2 e) (car (cdr (cdr e))))
(定义 (Multiply-e1 e) (car (cdr e)))
(定义 (Multiply-e2 e) (car (cdr (cdr e))))
```

(作为一个正交注释，我们之前没有见过语法' foo。这是一个Racket符号。对于我们的目的这里，符号' foo在某种意义上很像一个字符串 "foo"，你可以使用任何字符序列，但符号和字符串是不同的东西。比较两个符号是否相等是一种快速操作，比字符串相等性更快。你可以使用 eq?比较符号，而不应该使用 eq?对于字符串。 我们也可以使用字符串来完成这个例子，使用 equal?而不是 eq?。)

现在我们可以编写一个Racket函数来"评估"一个算术表达式。 它直接类似于ML版本中的eval_exp_new定义，只是使用我们的辅助函数而不是数据类型构造函数和模式匹配：

```
(定义 (eval-exp e)
    (条件 [(Const? e) e] ; 注意返回一个表达式，而不是一个数字
        [(Negate? e) (Const (- (Const-int (eval-exp (Negate-e e)))))]
        [(加? e) (让 ([v1 (常数-整数 (eval-exp (加-e1 e)))]
                      [v2 (常数-整数 (eval-exp (加-e2 e)))])
                  (常数 (+ v1 v2)))]
        [(Multiply? e) (let ([v1 (Const-int (eval-exp (Multiply-e1 e)))]
                             [v2 (Const-int (eval-exp (Multiply-e2 e)))])
                         (Const (* v1 v2)))]
        [#t (error "eval-exp 期望一个表达式")]))
```

同样，我们可以使用我们的辅助函数来定义算术表达式：

```
(定义 test-exp (Multiply (Negate (Add (Const 2) (Const 2))) (Const 7)))
(定义 test-ans (eval-exp test-exp))
```

请注意 test-ans是'(Const -28)，而不是 -28。

还要注意，使用动态类型，程序中没有任何东西定义"什么是算术表达式"。只有我们的文档和注释会指示算术表达式是如何由常量、否定、加法和乘法构建的。

通过Racket的 struct实现递归数据类型

上述定义算术表达式的方法不如我们现在介绍的第二种方法好，这种方法使用Racket中的特殊 struct结构。一个 struct定义的样子如下：

(结构 foo (bar baz quux) #:transparent)

这定义了一个名为 foo的新“struct”，类似于ML构造函数。 它在环境中添加了用于构造foo、测试某个东西是否为foo以及从foo中提取字段bar、baz和quux的函数。这些绑定的名称是根据构造函数名 foo系统地形成的：

- foo是一个函数，它接受三个参数并返回一个具有一个bar字段保存第一个参数，一个baz字段保存第二个参数和一个quux字段保存第三个参数的foo值。
- foo?是一个函数，它接受一个参数并对通过调用foo创建的值返回#t，对于其他所有值返回#f。
- foo-bar是一个函数，它接受一个foo并返回bar字段的内容，如果传递给它的不是foo，则引发错误。
- foo-baz是一个函数，它接受一个foo并返回baz字段的内容，如果传递给它的不是foo，则引发错误。
- foo-quux是一个函数，它接受一个foo并返回quux字段的内容，如果传递给它的不是foo，则引发错误。

在s*t*ruc*t*定义中，我们可以包含一些有用的属性来修改它们的行为，其中我们在这里讨论了两个。

首先，#:transparent属性使得字段和访问器函数在定义结构体的模块之外也可见。 从模块化的角度来看，这种做法是有问题的，但在使用DrRacket时有一个很大的优势：它允许REPL打印结构体的值及其内容，而不仅仅是一个抽象值。 例如，对于我们定义的struct foo，(foo "hi" (+ 3 7) #f)的结果打印为(foo "hi" 10 #f)。 如果没有#:transparent属性，它将打印为#<foo>，并且从调用foo函数产生的每个值都将以相同的方式打印。这个特性在检查由结构体的递归使用构建的值时变得更加有用。

其次，#:mutable属性通过提供类似set-foo-bar!、set-foo-baz!和set-foo-quux!的修改器函数使得所有字段都是可变的。 简而言之，当定义一个结构体时，程序员可以决定是否优势于具有可变字段的劣势。 还可以将一些字段设置为可变的，而将一些字段设置为不可变的。

我们可以使用结构体来定义一种新的表示算术表达式的方式，并且还可以定义一个评估这种算术表达式的函数：

```
(结构体 const (int) #:透明)
(结构体 negate (e) #:透明)
(结构体 add (e1 e2) #:透明)
(结构体 multiply (e1 e2) #:透明)

(定义 (eval-exp e)
  (条件 [(const? e) e] ; 注意返回的是一个表达式，而不是一个数字
        [(negate? e) (const (- (const-int (eval-exp (negate-e e)))))]
        [(add? e) (让 ([v1 (const-int (eval-exp (add-e1 e)))]
                       [v2 (const-int (eval-exp (add-e2 e)))])
                   (const (+ v1 v2)))]
        [(multiply? e) (let ([v1 (const-int (eval-exp (multiply-e1 e)))]
                             [v2 (const-int (eval-exp (multiply-e2 e)))])
                         (const (* v1 v2)))]
        [#t (error "eval-exp expected an exp")]))
```

与我们之前的方法一样，语言中没有任何指示如何定义算术表达式以常量、否定、加法和乘法为基础。 这个版本的 eval-exp的结构几乎与之前的版本完全相同，只是使用了结构定义提供的函数而不是我们自己的列表处理函数。 使用构造函数定义表达式也是类似的：

```
(define test-exp (multiply (negate (add (const 2) (const 2))) (const 7)))
(定义 test-ans (eval-exp test-exp))
```

为什么 struct方法更好

定义结构不是我们之前采用的列表方法的语法糖。 关键区别在于结构定义创建了一个新类型的值。给定

```
(结构体 add (e1 e2) #:透明)
```

函数 add返回导致add?返回 #t和其他每个类型测试函数如number?, pair?, null?, negate?和 multiply?返回 #f。 类似地，访问add值的 e1和 e2字段的唯一方法是使用 add-e1和 add-e2- 使用 car, cdr,multiply-e1等会导致运行时错误。 （相反，对于任何不是add的东西， add-e1和 add-e2会引发错误。）

注意我们的第一种方法使用列表没有这些属性。 从我们定义的Add函数构建的东西是一个列表，所以pair?对此返回#t，尽管这是不好的风格，我们可以直接使用car和cdr访问部分。

因此，除了更简洁外，我们基于结构的方法更优越，因为它可以更早地捕捉错误。
在我们的算术语言中，使用cdr或Multiply-e2对一个加法表达式几乎肯定是一个错误，但是我们基于列表的方法将其视为使用Racket原语访问列表的一种方式。同样，没有什么可以阻止我们代码的不明智客户写入(list 'Add "hello")，但是我们基于列表的Add? 函数将返回#t给出结果列表'(Add "hello")。。

话虽如此，关于结构定义我们在这里使用的方式并没有真正强制执行不变量。 特别是，我们希望确保任何加法表达式的 e1和 e2字段只包含其他算术表达式。 Racket有很好的方法来做到这一点，但我们在这里不研究它们。 首先，Racket有一个模块系统我们可以用来向客户端只暴露结构定义的一部分，所以我们可以隐藏构造函数并暴露一个不同的函数来强制执行不变量（就像我们在ML的模块系统中所做的那样）。[1]其次，Racket有一个合同系统允许程序员定义任意函数来检查结构字段的属性，例如只允许某些类型的值在字段中。

最后，我们指出Racket的 struct是一个强大的原语，不能用其他东西（如函数定义或宏定义）来描述或定义。 它真的创建了一种新的数据类型。 从 add得到的结果导致 add?返回 #t，但是其他类型测试返回 #f是任何以列表、函数、宏等为基础的方法所无法做到的。除非语言提供了一种原语来创建这样的新类型，否则任何其他的算术表达式编码都必须产生一些导致其他类型测试（如pair?或procedure?）返回 #t的值。返回#t的是一种无法通过列表、函数、宏等方法实现的特性，除非语言提供了一种原语来创建这样的新类型。

[1]许多人错误地认为动态类型的语言不能像这样强制实现模块化。 Racket的结构体和其他语言中的类似功能证明了这一点。 您不需要抽象类型和静态类型来强制实施ADT。只需要有一种创建新类型的方法，然后不直接暴露这些类型的构造函数即可。

一般情况下实现一种编程语言

虽然这门课程主要讲解编程语言特性的含义，而不是它们的实现方式，但实现一个小型编程语言仍然是一种宝贵的经验。 首先，了解某些特性的语义的一个很好的方法是必须实现这些特性，这迫使你思考所有可能的情况。 其次，它消除了高阶函数或对象之类的东西是“魔法”的想法，因为我们可以用更简单的特性来实现它们。 第三，许多编程任务类似于为编程语言实现一个解释器。 例如，处理像pdf文件这样的结构化文档，并将其转换为用于显示的像素矩形，类似于将输入程序转换为答案。

我们可以描述一种典型的语言实现工作流程如下。 首先，我们获取一个包含语言中程序的具体语法的字符串。 通常，这个字符串将是一个或多个文件的内容。 如果这个字符串在语法上不正确，即字符串不可能包含语言中的程序，比如关键字使用错误，括号放错位置等，解析器会报错。如果没有这样的错误，解析器会生成一个表示程序的树。 这被称为抽象语法树，简称AST。 它是下一步语言实现的更方便的表示形式。如果我们的语言包括类型检查规则或其他原因导致AST仍然不是一个合法的程序，类型检查器将使用这个AST来生成错误消息或不生成。然后将AST传递给实现的其余部分。

对于实现某种编程语言 B的其余部分，基本上有两种方法。 首先，我们可以用另一种语言 A编写一个解释器，该解释器接受 B中的程序并产生答案。 称这样的程序为 A中的“评估器”或“执行器”可能更合理，但是“B的解释器”已经成为标准术语数十年了。 其次，我们可以用另一种语言 A编写一个编译器，该编译器接受 B中的程序并产生一些其他语言 C（不一定是语言C）的等效程序，然后使用一些预先存在的 C的实现。对于编译，我们将 B称为源语言，将 C称为目标语言。 比“编译器”更好的术语可能是“翻译器”，但是“编译器”这个术语无处不在。 对于解释器方法或编译器方法，我们将编写 B的实现所用的语言 A称为元语言。

虽然有许多“纯”解释器和编译器，但现代系统通常结合了每个方面，并且使用多个层次的解释和翻译。 例如，一个典型的Java系统将Java源代码编译成可移植的中间格式。 Java“虚拟机”可以开始解释这种格式的代码，但通过将代码进一步编译为可以直接在硬件上执行的代码，可以获得更好的性能。 我们可以将硬件本身视为用晶体管编写的解释器，然而许多现代处理器实际上在硬件中具有将二进制指令转换为更小更简单的指令的翻译器，然后再执行它们。 即使在运行程序的这个多层次的故事中，也有许多变化和增强，但从根本上说，每一步都是解释或翻译的某种组合。

一个简短的讲道：解释器与编译器是特定编程语言实现的特性，而不是编程语言的特性。计算机科学中更令人讨厌和普遍的误解之一是存在“编译语言”（如C）和“解释语言”（如R*a*cket）。这是无稽之谈：我可以为C编写一个解释器，也可以为R*a*cket编写一个编译器。（事实上，DrR*a*cket采用了与J*ava*类似的混合方法。）C的实现历史悠久，使用编译器，而函数式语言的实现则使用解释器，但函数式语言的编译器已经存在了几十年。 例如，SML/NJ将每个模块/绑定编译为二进制代码。

在另一种语言中实现编程语言

我们的 eval-exp函数对算术表达式的处理是一个小型编程语言解释器的完美示例。 这里的语言是由常量、否定、加法和乘法表达式的构造函数正确构建的表达式。 “正确”一词的定义取决于语言；在这里，我们指的是常量保存数字，否定/加法/乘法保存其他正确的子表达式。 我们还需要一个关于我们小型语言的值（即结果）的定义，这也是语言定义的一部分。 这里我们指的是常量，即由 const构造函数构建的表达式的子集。 然后， eval-exp是一个解释器，因为它是一个根据我们语言的语义规则，将表达式转化为值的函数。 Racket只是我们编写解释器的“其他”语言，即元语言。

解析和类型检查发生了什么？ 简而言之，我们跳过了它们。 通过使用Racket的构造函数，
我们基本上直接以抽象语法树的形式编写程序，依赖于具有方便的
编写树的语法，而不是需要编写语法和解析器。 也就是说，我们编写了
以下表达式的程序：

```
(negate (add (const 2) (const 2)))
```

而不是像"- (2 + 2)"这样的字符串。

尽管将算术表达式这样的语言嵌入到Racket这样的另一种语言中可能看起来不方便，但它具有甚至超出不需要编写解析器的优点。 例如，下面我们将看到如何使用元语言（在这种情况下是Racket）来编写像宏一样的东西，用于我们的语言。

关于合法AST的假设和非假设

在我们的算术表达式语言中，有两种不同类型的“错误”AST之间存在微妙的区别。 为了使这种区别更清晰，让我们用三种更多的表达式扩展我们的语言：

```
(结构 常量 (整数) #:透明) ; 整数应该保存一个数字
(结构 否定 (e1) #:透明) ; e1应该保存一个表达式
(结构 加 (e1 e2) #:透明) ; e1，e2应该保存表达式
(结构 乘 (e1 e2) #:透明) ; e1，e2应该保存表达式
(结构 布尔 (b) #:透明) ; b应该保存#t或#f
(结构 如果-那么-否则 (e1 e2 e3) #:透明) ; e1，e2，e3应该保存表达式
(结构 等于-数字 (e1 e2) #:透明) ; e1，e2应该保存表达式
```

新功能包括布尔值（true或false），条件语句，以及用于比较两个数字并返回布尔值（当且仅当数字相同时为true）的结构。 至关重要的是，这种语言中表达式的求值结果现在可以是：

- 一个整数，例如 (const 17)
- 一个布尔值，例如 (bool true)
- 不存在，因为当我们尝试评估程序时，会出现“运行时类型错误”- 尝试将布尔值视为数字或反之亦然

换句话说，我们的语言现在有两种类型的值 - 数字和布尔值 - 并且有一些操作如果子表达式的评估结果是错误类型的值，则应该失败。

这最后一种可能性是解释器应该检查并给出适当的错误消息。 如果评估某种表达式（例如加法）需要评估子表达式的结果具有特定类型（例如像(const 4)这样的数字而不是像(bool #t)这样的布尔值），那么解释器应该检查此结果（例如使用 const?）而不是假设递归结果具有正确的类型。 这样，错误消息就是适当的（例如，“加法的参数不是数字”），而不是以解释器的实现为基础的内容。

与这些笔记对应的课程材料中发布的代码有两个完整的解释器
，第一个不包含任何检查，而第二个更好。 第一个解释器不包含任何检查，而第二个更好。 调用第一个解释器eval-exp-wrong和第二个解释器 eval-exp，这里只是两者的加法案例：

```
; eval-exp-wrong
[(add? e)
 (let ([i1 (const-int (eval-exp-wrong (add-e1 e)))]
       [i2 (const-int (eval-exp-wrong (add-e2 e)))])
   (const (+ i1 i2)))]

; eval-exp
[(add? e)
 (let ([v1 (eval-exp (add-e1 e))]
       [v2 (eval-exp (add-e2 e))])
   (if (and (const? v1) (const? v2))
       (const (+ (const-int v1) (const-int v2)))
       (error "add applied to non-number")))]
```

然而， eval-exp假设它正在评估的表达式是该语言的合法AST。 它可以处理(add (const 2) (const 2))，这将评估为(const 4)或(add (const 2) (bool #f))，这将遇到错误，但它不能优雅地处理(add #t #f)或(add 3 4)。 根据我们在注释中的规则，这些都不是合法的AST。

- 一个 const的int字段应该保存一个Racket数字。
- 一个 bool的b字段应该保存一个Racket布尔值。
- 表达式的所有其他字段应该保存其他合法的AST。（是的，定义是递归的。）

对于一个解释器来说，假设它得到了一个合法的AST是合理的，所以如果给它一个非法的AST，它可以“崩溃”并显示一个奇怪的、与实现相关的错误消息。

带有变量的语言需要环境的解释器

我们的算术表达式语言最大的缺点是没有变量。 这就是为什么我们可以只有一个递归函数，它接受一个表达式并返回一个值。 正如我们从课程一开始就知道的那样，由于表达式可以包含变量，所以评估它们需要一个将变量映射到值的环境。 因此，一个带有变量的语言的解释器需要一个递归的辅助函数，它接受一个表达式和一个环境，并产生一个值。[2]

[2]事实上，对于具有变异或异常功能的语言，辅助函数需要更多的参数。

环境的表示是解释器在元语言中的实现的一部分，而不是语言的抽象语法的一部分。 许多表示都可以，适合提供对常用变量快速访问的复杂数据结构。 但是对于我们的目的来说，忽略效率是可以的。 因此，以Racket作为我们的元语言，一个简单的关联列表，其中包含字符串（变量名称）和值（变量绑定到的值）的对，就足够了。

给定一个环境，解释器在不同的情况下以不同的方式使用它：

- 要评估变量表达式，它在环境中查找变量的名称（即字符串）。
- 要评估大多数子表达式，例如加法操作的子表达式，解释器将相同的环境传递给递归调用，该环境用于评估外部表达式。
- 为了评估像let表达式的主体这样的东西，解释器将稍微不同的环境传递给递归调用，例如在其中多绑定（即，字符串和值的一对）的环境。

为了评估整个程序，我们只需调用递归辅助函数，该函数接受一个带有程序和适当初始环境的环境，例如空环境，其中没有绑定。

实现闭包

为了实现具有函数闭包和词法作用域的语言，我们的解释器需要“记住”函数定义时的“当前”环境，以便在调用函数时使用该环境而不是调用者的环境。 做到这一点的“技巧”相当直接：我们可以字面上创建一个小的数据结构，称为闭包，其中包括环境和函数本身。这对（闭包）就是解释函数的结果。换句话说，函数不是一个值，闭包是，因此函数的评估产生一个“记住”我们评估函数时的环境的闭包。

我们还需要实现函数调用。 一个调用有两个表达式 `e1`和 `e2`，看起来像`e1 e2`在ML中或`(e1 e2)`in Racket中。 （我们在这里考虑单参数函数，尽管实现自然支持模拟多参数函数的柯里化。）我们按照以下方式评估一个调用：

- 我们使用当前环境评估 `e1`。 结果应该是一个闭包（否则就是运行时错误）。
- 我们使用当前环境评估 `e2`。 结果将成为闭包的参数。
- 我们使用闭包的环境部分扩展为调用点处的参数映射到代码部分的参数，评估闭包的代码部分。

在与这些课程材料相关的作业中，还有一个额外的环境扩展，允许闭包递归调用自身。 但关键思想是相同的：我们扩展存储在闭包中的环境来评估闭包的函数体。

这确实是解释器实现闭包的方式。 这是我们学习闭包时学到的语义，只是在解释器中“编码”。

更高效地实现闭包

在每个闭包中存储“整个当前环境”可能看起来很昂贵。 首先，当环境是关联列表时，并不那么昂贵，因为不同的环境只是彼此的扩展，并且在使用 cons创建更长的列表时我们不会复制列表。 （回想一下，这种共享是不变列表的一个重要好处，并且我们不会改变环境。）其次，在实践中，我们可以通过仅存储函数体可能使用的环境部分来节省空间。我们可以查看函数体并查看它具有的自由变量（函数体中使用的在函数体外定义的变量），闭包中存储的环境只需要这些变量。 毕竟，如果函数体不使用变量，闭包的执行永远不需要从环境中查找变量。 语言实现在开始评估之前预先计算每个函数的自由变量。 它们可以将结果与每个函数一起存储，以便在构建闭包时快速获取这组变量。

最后，你可能会想知道如果目标语言本身没有闭包，编译器是如何实现闭包的。 作为翻译的一部分，函数定义仍然会评估为具有两个部分（代码和环境）的闭包。 然而，我们没有一个带有“当前环境”的解释器，每当我们遇到需要查找的变量时，我们需要查找。因此，我们将程序中的所有函数更改为接受一个额外的参数（环境），并将所有函数调用更改为显式传递这个额外的参数。 现在，当我们有一个闭包时，代码部分将有一个额外的参数，调用者将传递环境部分作为这个参数。 然后，编译器只需要将所有自由变量的使用转换为使用额外参数查找正确的值的代码。 在实践中，使用良好的环境数据结构（如数组）可以使这些变量查找非常快（与从数组中读取值一样快）。

通过元语言中的函数定义“宏”

在实现解释器或编译器时，将实现语言和用于实现的语言（元语言）分开是至关重要的。 例如，eval-exp是一个Racket函数，它接受一个算术表达式语言的表达式（或者我们正在实现的任何语言），并生成一个算术表达式语言的值。 因此，例如，算术表达式语言的表达式永远不会包含对eval-exp或Racket加法表达式的使用。

但是由于我们在Racket中编写要评估的程序，我们可以使用Racket助手函数来帮助我们创建这些程序。 这样做基本上是使用Racket函数来定义我们的语言的宏语言。这是一个例子：

```
(define (double e) ; 接受语言实现的语法并生成语言实现的语法
   (multiply e (const 2)))
```

这里的double是一个Racket函数，它接受算术表达式的语法并生成算术表达式的语法。 调用double会产生我们语言中的抽象语法，就像宏展开一样。 例如，(negate (double (negate (const 4))))会产生(negate (multiply (negate (const 4)) (const 2)))。 请注意，这个“宏”double不会以任何方式评估程序：我们生成抽象语法，然后可以对其进行评估，放入更大的程序中等等。

能够做到这一点是“嵌入”我们的小语言在Racket元语言中的一个优势。
无论选择哪种元语言，相同的技术都适用。 然而，这种方法不能很好地处理与变量屏蔽相关的问题，而真正的宏系统具有卫生宏。

这是一个有趣的“宏”的不同之处在于两个方面。 首先，参数是一个 *Racket*语言的列表（语法）。 其次，“宏”是递归的，对参数列表中的每个元素调用自身一次：

```
(定义 (list-product es)
  (如果 (null? es)
      (const 1)
      (multiply (car es) (list-product (cdr es)))))
```

ML与Racket

在研究静态类型和优缺点的一般主题之前，有必要对我们迄今为止学习的两种语言，ML和Racket进行更具体的比较。这两种语言在许多方面都很相似，具有鼓励函数式风格的结构（避免突变，使用一级闭包），同时在适当的情况下允许突变。 还有许多不同之处，包括语法方面非常不同，ML对模式匹配的支持与Racket对结构体的访问器函数相比，Racket对let表达式的多个变体等。

但是这两种语言之间最普遍的区别是ML有一个静态类型系统，而Racket没有。[3]

我们将在下面准确地研究静态类型系统是什么，ML的类型系统保证了什么，以及静态类型的优缺点。那些已经在ML和Racket中编程的人可能已经对这些话题有了一些想法，当然：ML在运行之前通过类型检查和报告错误来拒绝很多程序。 为了做到这一点，ML强制执行某些限制（例如，列表的所有元素必须具有相同的类型）。 因此，ML确保在“编译时”不存在某些错误（例如，我们永远不会尝试将字符串传递给加法运算符）。

更有趣的是，我们能否用更类似Racket的思想来描述ML及其类型系统，反之亦然，我们能否用ML的思想来描述Racket风格的编程？ 事实证明我们可以，并且这样做既能拓宽思维，也是后续主题的良好前导。

首先考虑一下一个Racket程序员可能如何看待ML。忽略语法差异和其他问题，我们可以将ML描述为大致定义了Racket的一个子集：运行的程序产生相似的答案，但是ML拒绝了更多的程序作为非法的，即不属于该语言的一部分。 那有什么优势呢？ ML被设计为拒绝可能是错误的程序。 Racket允许像(define (f y) (+ y (car y)))这样的程序，但是对 f的任何调用都会导致错误，所以这几乎不是一个有用的程序。 所以ML拒绝这个程序而不是等到程序员测试 f时才发现。 同样，类型系统可以捕捉到由程序的不同部分引起的不一致假设导致的错误。 函数(define (g x) (+ x x))和(define (h z) (g (cons z 2)))分别是合理的，但是如果 h中的 g绑定到这个 g的定义，那么对 h的任何调用都会失败，就像对 f的任何调用一样。 另一方面，ML也拒绝了类似Racket的不是错误的程序。 例如，在这段代码中，if表达式和绑定到 xs的表达式都无法通过类型检查，但是根据情况，它们代表了合理的Racket习惯用法。

```
(定义 (f x) (如果 (> x 0) #t (列表 1 2)))
(定义 xs (列表 1 #t "嗨"))
(定义 y (f (car xs)))
```

那么现在ML程序员如何看待Racket？ 一个观点是与上述讨论相反，Racket接受一组程序的超集，其中一些是错误的，一些不是。一个更有趣的观点是，Racket只是ML的一个表达式是一个大数据类型的一部分。 从这个角度来看，每个计算的结果都会被一个构造函数隐式地“包装”到一个大数据类型中

[3]还有一个相关的语言Typed Racket也可以在DrRacket系统中使用，它与Racket和许多其他语言良好地交互，允许您混合使用不同语言编写的文件来构建应用程序。 我们在本课程中不会研究这个，所以我们只提到Racket语言。

像 +这样的原始类型有实现，它们会检查其参数的“标签”（例如，检查它们是否为数字），并根据需要引发错误。 更详细地说，就像Racket有这个数据类型绑定一样：

```
datatype theType = Int of int
                 | String of string
                 | Pair of theType * theType
                 | Fun of theType -> theType
                 | ... (*每个内置类型一个构造函数*)
```

然后，当程序员写下类似于42的东西时，它实际上是隐式地Int 42，以便每个表达式的结果都具有类型 theT ype。 然后，像 +这样的函数如果两个参数没有正确的构造函数，则会引发错误，并且如果需要，它们的结果也会用正确的构造函数包装。 例如，我们可以将 car视为：

```
fun car v = case v of Pair(a,b) => a | _ => raise ... (*给出一些错误*)
```

由于这种“秘密的模式匹配”对程序员不可见，Racket还提供了程序员可以使用的which-constructor函数。 例如，原始的 pair?可以被视为：

```
有趣的对吗？ v = case v of Pair _ => true | _ => false
```

最后，Racket的结构定义做了一件你用ML数据类型绑定无法完全做到的事情：它们动态地向数据类型中添加新的构造函数。[4]

我们可以将Racket看作是theType的一种形式，这表明在ML中你可以做到的任何事情，在Racket中也可以做到，尽管可能更加笨拙：ML程序员可以使用类似于上面的theType定义来显式地编程。

什么是静态检查?

通常所说的“静态检查”是指在程序成功解析之后但在运行之前对其进行拒绝的任何操作。 如果程序无法解析，我们仍然会得到一个错误，但我们称这样的错误为“语法错误”或“解析错误”。相反，静态检查的错误，通常是“类型错误”，将包括未定义的变量或使用数字而不是对。 我们在没有任何程序输入被识别的情况下进行静态检查 - 这是“编译时检查”，尽管语言实现在静态检查成功后使用编译器还是解释器是无关紧要的。

静态检查的内容是编程语言的一部分。 不同的语言可以做不同的事情；有些语言根本不进行静态检查。在具有特定定义的语言中，您还可以使用其他工具进行更多的静态检查，以尝试找到错误或确保其不存在，尽管这些工具不是语言定义的一部分。

定义语言的静态检查最常见的方法是通过类型系统。 当我们学习ML时（以及您学习Java时），我们为每个语言构造提供了类型规则：每个变量都有一个类型，条件语句的两个分支必须具有相同的类型等。ML的静态检查是检查这些规则是否被遵循（在ML的情况下，还会推断类型来进行检查）。 但这是语言对静态检查的方法（它是如何进行的），这与静态检查的目的（它实现了什么）不同。 目的是拒绝那些“毫无意义”或“可能试图滥用语言特性”的程序。

[4]你可以在ML中使用 exn类型来实现这个，但不能使用datatype绑定。 如果可以的话，对于缺少模式匹配子句的静态检查将不可能进行。

类型系统通常无法防止的错误（例如数组越界错误）以及类型系统无法防止的其他错误，除非提供有关程序应该做什么的更多信息。例如，如果程序将条件语句的分支放错顺序或者调用 +而不是 *，这仍然是一个程序，只是不是预期的那个。

例如，ML的类型系统的一个目的是防止将字符串传递给算术原语，如除法运算符。 相比之下，Racket使用“动态检查”（即运行时检查），通过为每个值添加标记，并使除法运算符检查其参数是否为数字。 ML实现不需要为此目的标记值，因为它可以依靠静态检查。 但正如我们将在下面讨论的那样，这种权衡是静态检查器必须拒绝一些实际上不会出错的程序。

正如ML和Racket所展示的，防止“坏事情”发生的典型时机是“编译时”和“运行时”。然而，值得注意的是，关于何时将某个错误声明为错误，实际上存在着一个连续性的渴望。 以除零为例，大多数类型系统在静态上无法阻止这种情况。 如果我们有一个包含表达式(/ 3 0)的函数，那么什么时候会导致错误：

- 按键时间：调整编辑器，以便我们甚至不能写下分母为0的除法。 这是近似的，因为也许我们正要写0.33，但是我们不被允许写0。
- 编译时：一旦我们看到这个表达式。 这是近似的，因为也许上下文是(if #f (/ 3 0) 42)。
- 链接时：一旦我们看到包含(/ 3 0)的函数可能会被某个“主”函数调用。
  这比编译时更不准确，因为有些代码可能永远不会被使用，但我们仍然需要
  近似估计哪些代码可能被调用。
- 运行时：一旦我们执行除法操作。
- 稍后：与其引发错误，我们可以返回某种表示除以零的值，并且直到该值被用于需要实际数字的地方（例如索引数组）之前，不引发错误。

虽然“稍后”选项一开始可能看起来太宽松，但这正是浮点计算所做的。(/ 3.0 0.0)产生+inf.0，仍然可以进行计算，但无法转换为精确数字。 在科学计算中，这非常有用，可以避免许多额外的情况：也许我们会计算 $\pi/2$的正切，但只有在最终答案中不使用时才这样做。

正确性：完备性，完整性，不可判定性

直观上，静态检查器如果能够防止它声称要防止的事情发生，那么它就是正确的 - 否则，需要修复语言定义或静态检查的实现。 但是，我们可以通过定义“正确性”的术语来给出更精确的描述，即定义“完备性”和“完整性”。 对于这两者，定义是相对于我们希望防止的某个事物X而言的。 例如，X可以是“程序查找一个环境中不存在的变量”。

如果一个类型系统在运行某个输入时从不接受一个程序，那么它就是“正确”的。

如果一个类型系统无论用什么输入运行一个程序都不会拒绝它，那么它就是“完备”的。

理解这些定义的一个好方法是，正确性防止了假阴性，而完备性防止了假阳性。 术语“假阴性”和“假阳性”来自统计学和医学：假设有一种针对某种疾病的医学测试，但它并不是一个完美的测试。 如果测试没有检测到疾病，但患者实际上患有该疾病，那么这是一个假阴性（测试结果是阴性，但实际上是错误的）。 如果测试检测到疾病，但患者实际上并没有患有该疾病，那么这是一个假阳性（测试结果是阳性，但实际上是错误的）。 在静态检查中，疾病是“在某个输入下执行X”，而测试是“程序是否通过类型检查？”术语“正确性”和“完备性”来自逻辑学，并且在编程语言的研究中常常使用。 一个正确的逻辑只证明真实的事情。 一个完备的逻辑证明所有真实的事情。 在这里，我们的类型系统是逻辑，我们试图证明的是“X不会发生”。

在现代语言中，类型系统是完备的（它们可以防止它们声称的事情），但不完全的（它们拒绝了它们不需要拒绝的程序）。 完备性很重要，因为它使语言用户和语言实现者能够依赖于从未发生的事情。 完全性是好的，但希望在实践中很少发生程序被不必要地拒绝的情况下，在这些情况下，希望程序员能够轻松修改程序以使其通过类型检查。

类型系统不完全，因为对于几乎任何您可能希望静态检查的内容，都不可能实现一个静态检查器，该检查器可以给出您语言中的任何程序（a）始终终止，（b）是完备的，（c）是完全的。 由于我们必须放弃其中一个，（c）似乎是最好的选择（程序员不喜欢可能不终止的编译器）。

不可能性结果正是计算理论研究的核心思想不可判定性。 这是一门必修课程（CSE 311）中的一个重要主题。 了解程序的非平凡属性是不可判定的含义对于成为受过教育的计算机科学家来说是基本的。 不可判定性直接意味着静态检查的固有近似（即不完全性）是不可判定性的最重要的影响。 我们简单地不能编写一个程序，该程序以ML / Racket / Java /等作为输入，始终正确地回答诸如“这个程序会除以零吗？”“这个程序会将字符串视为函数吗？”“这个程序会终止吗？”等问题。

弱类型

现在假设一个类型系统对于某个属性 X是不安全的。 那么为了安全起见，语言实现
至少在某些情况下应该执行动态检查以防止 X发生，并且语言
definition应该允许这些检查可能在运行时失败。

但是另一种选择是说如果 X发生了，那是程序员的错，语言定义不需要检查。 事实上，如果 X发生了，那么运行中的程序可以做任何事情：崩溃，损坏数据，产生错误答案，删除文件，启动病毒或点燃计算机。 如果一种语言的程序允许合法的实现点燃计算机（尽管可能不会这样做），我们称这种语言为弱类型。 行为受限的错误程序的语言被称为强类型。 这些术语有点不幸，因为类型系统的正确性只是问题的一部分。 毕竟，Racket是动态类型的，但仍然是强类型的。 此外，在弱类型语言中，实际的未定义和不可预测行为的一个重要来源是数组边界错误（它们不需要检查边界 - 它们可能只是错误地访问其他数据），然而很少有类型系统检查数组边界。

C和C++是众所周知的弱类型语言。 为什么它们被定义成这样？ 简而言之，因为设计者不希望语言定义强制实现执行所有必要的动态检查。 虽然执行检查会有时间成本，但更大的问题是实现必须保留额外的数据（如值上的标签）来执行检查，而C/C++是经过设计的。

作为低级语言，程序员可以期望额外的“隐藏字段”不会被添加。

一个较旧但现在很少见的观点支持弱类型，体现在这句话中：“强类型适用于弱智。”这个观点是，任何强类型语言要么在静态上拒绝程序，要么在动态上执行不必要的测试（参见上面的不可判定性），所以人类应该能够在他/她知道这些测试是不必要的地方“否决”这些检查。 实际上，人类非常容易出错，即使自动检查必须对我们保持谨慎，我们也应该欢迎它。此外，类型系统随着时间的推移变得更加表达能力强大（例如，多态），语言实现也变得更好，可以优化掉不必要的检查（但永远不可能完全消除所有检查）。 与此同时，软件变得非常庞大、非常复杂，并且被整个社会所依赖。 一个在C语言中编写的3000万行操作系统中的1个错误会导致整个计算机容易受到安全漏洞的攻击，这是一个非常严重的问题。

虽然这仍然是一个真正的问题，而C语言提供的支持很少，但使用其他工具来对C代码进行静态和/或动态检查以尝试防止此类错误变得越来越常见。

更灵活的基元是一个相关但不同的问题

假设我们改变了ML，使得类型系统接受任何表达式`e1 + e2`只要 `e1`和 `e2`有 *some*type，并且我们改变了加法的求值规则，如果其中一个参数没有得到一个数字，则返回 `0`。 这会使ML成为一种动态类型的语言吗？从某种意义上说，它在语言更宽松并且一些“可能”的错误没有被及时检测到方面更加动态，但仍然有一个拒绝程序的类型系统 - 我们只是改变了“非法”操作的定义以允许更多的加法。 我们也可以类似地改变Racket，如果给出错误的参数，则不会出现错误。 Racket设计者选择不这样做是因为它很可能掩盖错误而没有太多用处。

其他语言通过扩展原始操作的定义来减少错误报告，使得在这种情况下不会出错。 除了对任何类型的数据进行算术运算之外，还有一些例子：

- 允许超出数组边界的访问。 例如，如果 `arr`少于10个元素，我们仍然可以允许 `arr[10]`，只需返回一个默认值或通过扩大数组来实现 `arr[10]=e`。
- 允许使用错误数量的参数进行函数调用。 多余的参数可以被忽略。
  参数过少可以用语言选择的默认值填充。

这些选择是语言设计的问题。 赋予可能是错误的含义通常是不明智的 - 它掩盖了错误，并使得在一些无意义的计算之后程序运行变得更加困难。 另一方面，当提供时，这种“更动态”的特性被程序员使用，所以显然有人发现它们有用。

对于我们在这里的目的，我们只将其视为与静态与动态类型不同的问题。 我们不是在程序运行之前或运行时阻止某些X（例如，使用过多参数调用函数），而是改变语言语义，以便我们根本不阻止X - 我们允许它并扩展我们的评估规则以给予它语义。

静态检查的优点和缺点

现在我们知道什么是静态和动态类型，让我们进入几十年的争论中，哪个更好。 我们知道静态类型检查可以尽早捕捉到许多错误，完备性确保某些类型的错误不会保留，而不完全性意味着一些完全正常的程序被拒绝。 我们不会明确回答静态类型是否可取（如果没有其他原因，它取决于您要检查的内容），但是

我们将考虑七个具体的主张，并对每个主张都考虑支持和反对静态类型的有效论点。

1. 静态类型还是动态类型更方便?

认为动态类型更方便的论点源于能够混合使用不同类型的数据，如数字、字符串和对，而无需声明新的类型定义或在模式匹配中“混乱”代码。 例如，如果我们想要一个返回数字或字符串的函数，我们可以直接返回一个数字或字符串，调用者可以根据需要使用动态类型谓词。在Racket中，我们可以这样写：

```
(define (f y) (if (> y 0) (+ y y) "hi"))
(let ([ans (f x)]) (if (number? ans) (number->string ans) ans))
```

相比之下，类似的ML代码需要使用数据类型，其中包含在f中的构造函数和模式匹配来使用结果：

```
datatype t = Int of int | String of string
fun f y = if y > 0 then Int(y+y) else String "hi"
val _ = case f x of Int i => Int.toString i | String s => s
```

另一方面，静态类型使得假设数据具有某种类型更加方便，知道这种假设不会被违反，否则会导致后续错误。 对于Racket函数来说，要确保一些数据是一个数字，它必须在代码中插入一个明确的动态检查，这样做更加繁琐和难以阅读。 相应的ML代码没有这样的尴尬。

```
(define (立方 x)
  (如果 (不 (是数字? x))
      (error "立方函数期望一个数字")
      (* x x x)))
(立方 7)

fun 立方 x = x * x * x
val _ = 立方 7
```

注意，在Racket代码中没有检查的情况下，实际错误会在乘法的主体中出现，这可能会让不知道立方是使用乘法实现的调用者感到困惑。

2. 静态类型是否阻止有用的程序?

动态类型不会拒绝那些完全合理的程序。 例如，下面的Racket代码将'((7 . 7) . (#t . #t))绑定到pair_of_pairs而没有问题，但是相应的ML代码无法通过类型检查，因为ML类型系统无法给 f一个类型。

```
(定义 (f g) (cons (g 7) (g #t)))
```

[5]这是ML的一个限制。有一些具有更多表达形式的多态语言可以对这样的代码进行类型检查。但由于不可判定性，总会有限制。

```
(定义 pair_of_pairs (f (lambda (x) (cons x x))))

fun f g = (g 7, g true) (* 无法通过类型检查 *)
val pair_of_pairs = f (fn x => (x,x))
```

当然，我们可以编写一个ML程序来产生((7,7),(true,true))，但我们可能需要“绕过类型系统”而不是按照我们想要的方式来做。

另一方面，动态类型的灵活性来自于给每个值打上标签。 在ML和其他静态类型语言中，当我们想要时，我们可以通过使用数据类型和显式标签来做同样的事情。 在极端情况下，如果你想像Racket一样在ML中编程，你可以使用一个数据类型来表示“TheOne Racket Type”，并在每个地方插入显式标签和模式匹配。 虽然这种编程风格在任何地方都很痛苦，但它证明了一个观点，即在Racket中我们无法做到的事情在ML中都可以做到。（我们已经在上面简要讨论过这个问题。）

```
数据类型 tort = Int of int
              | String of string
              | Pair of tort * tort
              | Fun of tort -> tort
              | Bool of bool
              | ...
fun f g = (case g of Fun g' => Pair(g' (Int 7), g' (Bool true)))
val pair_of_pairs = f (Fun (fn x => Pair(x,x)))
```

也许更简单的支持静态类型的论点是，现代类型系统足够表达，以至于它们很少妨碍你。你有多经常尝试编写像 f这样的函数，在ML中无法通过类型检查？

3. 静态类型的早期错误检测重要吗?

支持静态类型的一个明确论点是，它能够在你静态检查（非正式地称为“编译”）代码时尽早捕获错误。 软件开发的一个众所周知的真理是，如果在开发者还在思考代码时就发现了错误，那么修复起来会更容易。考虑这个Racket程序：

```
(定义 (pow x)
   (lambda (y)
      (如果 (= y 0)
          1
          (* x (幂 x (- y 1))))))
```

虽然算法看起来是正确的，但是这个程序有一个错误：pow期望柯里化的参数，但是递归调用通过pow传递了两个参数，而不是通过柯里化。 这个错误直到测试pow时，y不等于0时才被发现。 等价的ML程序根本无法通过类型检查：

```
fun pow x y = (* 无法通过类型检查 *)
  如果 y = 0
  那么 1
  否则 x * pow (x,y-1)
```

因为静态检查器可以捕捉已知类型的错误，专业的程序员可以利用这个知识将注意力放在其他地方。 当编写大部分代码时，程序员可能对元组和柯里化的区别非常懒散，因为他们知道类型检查器会在后面给出一系列可以快速纠正的错误。 这样做可以释放出精力来专注于其他任务，比如数组边界推理或更高级的算法问题。

动态类型支持者会认为静态检查通常只能捕捉到你在测试中也能发现的错误。 由于你仍然需要测试你的程序，因此在运行测试之前捕捉到一些错误的附加价值就降低了。 毕竟，下面的程序不作为指数函数工作（它们使用了错误的算术运算），ML的类型系统无法检测到这一点，而测试可以捕捉到这个错误，也可以捕捉到上面的柯里化错误。

```
(定义 (pow x) ; 错误的算法
  (lambda (y)
    (如果 (= y 0)
        1
        (+ x ((pow x) (- y 1))))))

fun pow x y = (* 错误的算法 *)
   if y = 0
   then 1
   否则 x + pow x (y - 1)
```

4. 静态类型还是动态类型导致更好的性能?

静态类型可以导致更快的代码，因为它不需要在运行时执行类型检查。 事实上，很大一部分性能优势可能是由于根本不存储类型标签，这需要更多的空间并减慢构造函数的速度。 在ML中，只有在程序员使用数据类型构造函数时才有运行时标签。

动态类型有三个合理的反驳观点。 首先，在大多数软件中，这种低级性能并不重要。 其次，动态类型语言的实现可以尝试优化掉不必要的类型检查。 例如，在(let ([x (+ y y)]) (* x 4))中，乘法不需要检查x和4是否为数字，而加法只需要检查y一次。

虽然没有优化器能够从每个程序中删除所有不必要的测试（不可判定性再次发作），但在实践中，对于性能至关重要的程序部分来说，这可能足够容易。 第三，如果静态类型语言的程序员必须绕过类型系统的限制，那么这些变通方法可能会削弱所谓的性能优势。 毕竟，使用数据类型的ML程序也有标签。

5. 静态类型还是动态类型使代码重用更容易?

可以说，动态类型使重用库函数更容易。 毕竟，如果你用cons单元格构建了很多不同类型的数据，你可以继续使用car、cdr、cadr等来获取其中的部分，而不是为每个数据结构定义很多不同的getter函数。 另一方面，这可能会掩盖错误。
例如，假设你不小心将一个列表传递给一个期望树的函数。 如果cdr在它们两者上都起作用，你可能会得到错误的答案或在以后引发一个神秘的错误，而使用不同类型的列表和树可以更早地捕捉到错误。

这是一个非常有趣的设计问题，比静态与动态类型更一般化。 通常情况下，重用已有的库或数据结构是很好的，特别是因为你可以重用所有的函数。

有时候，这会使得将在概念上真正不同的事物分离变得太困难，所以最好定义一个新的类型。 这样，静态类型检查器或动态类型测试可以在你将错误的东西放在错误的位置时捕捉到。

6. 静态类型还是动态类型对原型设计更好?

在软件项目的早期阶段，你正在开发一个原型，同时你也在改变对软件将要做什么以及实现方法的看法。

动态类型通常被认为对原型设计更好，因为当变量、函数和数据结构的类型还在变化时，你不需要花费精力去定义它们。 此外，你可能知道你的程序的一部分目前还没有意义（在静态类型语言中无法通过类型检查），但你仍然想运行你的程序的其他部分（例如，测试刚刚编写的部分）

静态类型的支持者可能会反驳说，在软件设计中，尽早记录类型是非常重要的，即使它们不清楚并且会发生变化。 此外，注释掉代码或添加类似于模式匹配分支的存根，如_ => raise Unimplemented通常很容易，并且可以记录程序中已知不工作的部分。

7. 静态类型还是动态类型对代码演进更好?

软件工程中花费了大量的精力来维护工作中的程序，通过修复错误，添加新功能，并且通常通过改变代码来使其演进。

动态类型有时在代码演进方面更加方便，因为我们可以更改代码以更加宽容（接受更多类型的参数），而无需更改任何现有的代码客户端。 例如，考虑将这个简单的函数更改为：

```
(定义 (f x) (* 2 x))
```

到这个版本，它可以处理数字或字符串：

```
(定义 (f x)
  (如果 (number? x)
      (* 2 x)
       (string-append x x) ))
```

没有现有的调用者，可以使用 f与数字，可以告诉这个变化是如何进行的，但是新的调用者可以传入字符串甚至是不知道该值是数字还是字符串的值。 如果我们在ML中进行类似的更改，那么所有现有的调用者都将无法通过类型检查，因为它们都必须将其参数包装在 Int构造函数中，并在函数结果上使用模式匹配：

```
fun f x = 2 * x
```

```
数据类型 t = Int of int | String of string
fun f x =
  case f x of
    Int i     => Int (2 * i)
  | String s => String (s ^ s)
```

另一方面，静态类型检查在演化代码时非常有用，可以捕捉到演化引入的错误。 当我们更改函数的类型时，所有的调用者都不再通过类型检查，这意味着类型检查器给我们提供了一个宝贵的“待办事项列表”，列出了所有需要更改的调用点。 根据这个论点，演化代码的最安全方式是更改规范正在更改的任何函数的类型，这是在类型中捕获尽可能多的规范的一个论点。

在ML中，一个特别好的例子是当你需要向数据类型添加一个新的构造函数时。 如果你没有使用通配符模式，那么你将会收到所有使用该数据类型的case表达式的警告。

尽管“类型检查器的待办事项列表”非常有价值，但是在所有项目都得到解决之前，程序将无法运行，或者如前面所述，您可以使用注释或存根来删除尚未演化的部分。

可选：eval和quote

（这个简短的描述只是介绍了使用eval进行编程的表面。鼓励有兴趣的学生自行学习更多。）

在某种意义上，可以说Racket是一种解释型语言：它具有一个原始的eval函数，可以在运行时接受一个程序的表示并对其进行评估。例如，这个程序，虽然不是很好的风格，因为有更简单的方法来实现它的目的，可能会打印一些内容，取决于x的值。

```
(定义 (make-some-code y)
   (if y
       (列表 'begin (列表 'print "hi") (列表 '+ 4 2))
       (列表 '+ 5 3)))
(定义 (f x)
  (评估 (make-some-code x)))
```

Racket函数make-some-code很奇怪：它不会打印或执行加法。 它只是返回一个包含符号、字符串和数字的列表。 例如，如果调用 #t，它会返回

```
'(begin (print "hi") (+ 4 2))
```

这只是一个包含三个元素的列表，其中第一个元素是符号 begin。
这只是Racket的数据。 但是，如果我们查看这些数据，它看起来就像我们可以运行的Racket程序。嵌套的列表在一起是一个完全有效的Racket表达式的表示，如果求值，将打印"hi"并且结果为6。

原始的评估函数接受这样的表示，并在运行时对其进行求值。我们可以执行任何计算来生成传递给评估函数的数据。 作为一个简单的例子，我们可以将两个列表连接在一起，如(list '+ 2)和(list 3 4)。 如果我们用结果'(+ 2 3 4)，即一个包含4个元素的列表，调用评估函数，则评估函数返回9。

许多语言都有 eval，但也有许多语言没有，对于如何使用它的适当习惯是一个有争议的问题。 大多数人都同意它往往被滥用，但它也是一个非常强大的结构，有时候正是你想要的。

一个基于编译器的语言实现（注意我们没有说“编译语言”）能处理 eval吗?
嗯，它需要在运行时具有编译器或解释器，因为它无法预先知道可能会传递给 eval的内容。 基于解释器的语言实现也需要在运行时具有解释器或编译器，但是，当然，它已经需要用来评估“常规程序”。

在像Javascript和Ruby这样的语言中，我们没有Racket语法的便利，其中程序和列表看起来非常相似， eval可以接受一个看起来完全像Racket语法的列表表示。
相反，在这些语言中， eval接受一个字符串，并通过首先解析它然后运行它来解释它作为具体语法。无论语言如何，如果给出一个格式错误的程序或引发错误的程序， eval都会引发错误。

在Racket中，写make-some-code的方式是痛苦且不必要的。相反，有一个特殊的形式 quote将其下的所有内容视为符号、数字、列表等，而不是要调用的函数。所以我们可以这样写：

```
(定义 (make-some-code y)
  (if y
      (引用 (开始 (打印 "嗨") (+ 4 2)))
      (引用 (+ 5 3)))))
```

有趣的是， eval和 quote是相反的：对于任何表达式 e，我们应该有(eval (quote e))作为一种糟糕但等效的写法 e。

通常 quote太强大了 - 我们想引用大多数东西，但在我们正在构建的大部分语法中，评估一些代码是方便的。 Racket有quasiquote和unquote来实现这一点（如果感兴趣，请参阅手册），而Racket的语言前辈们几十年来一直具备这个功能。 在现代脚本语言中，经常看到类似的功能：在字符串中嵌入表达式评估的能力（然后可能或可能不会调用 eval），就像使用Racket的quote表达式构建要用于 eval的内容一样。 这个功能在脚本语言中有时被称为插值，但它只是*quasiquoting*。

CSE341：程序设计语言 2016年春季
第7单元总结

标准描述：这个总结大致涵盖了课堂和复习部分的内容。 当回顾材料时，以叙述方式阅读材料并将整个单元的材料放在一个文档中可能会有所帮助。 请报告这些笔记中的错误，甚至是打字错误。 这个总结不能完全替代上课、阅读相关代码等。

目录

Ruby物流

课程网站提供了Ruby及其REPL（称为 `irb`）的安装和基本使用说明，因此这里不再重复。 请注意，为了保持一致性，我们将要求使用Ruby版本2.x.y（对于任何x和y），尽管这是为了作业目的 - 我们将讨论的概念并不依赖于一个确切的版本，当然。

有很多免费的Ruby文档可在 `http://ruby-doc.org` 和 `http://www` 上找到。
`ruby-lang.org/en/documentation/`也是我们推荐的。 我们还推荐《*Programming Ruby 1.9 & 2.0*, *The Pragmatic Programmers' Guide*》，尽管这本书不是免费的。 由于在线文档非常好，其他课程材料可能不会详细描述讲座和作业中使用的每个语言特性，尽管我们的目标也不是让你刻意寻找。 总的来说，在一些初始背景之后学习新的语言特性和库是一项重要的技能，以指引你朝正确的方向发展。

Ruby特性对于PL课程最有趣

Ruby是一种大型、现代的编程语言，具有各种特性，使其受欢迎。 其中一些特性对于编程语言特性和语义的课程很有用，而其他一些特性则不太有用，尽管它们在日常编程中可能非常有用。 我们的重点将放在面向对象编程、动态类型、块（几乎是闭包）和混入上。 我们在这里简要描述这些特性和一些其他区别Ruby的东西 - 如果你还没有见过面向对象的编程语言，那么在学习更多Ruby之前，这个概述可能没有意义。

- Ruby是一种纯面向对象的语言，这意味着语言中的所有值都是对象。 例如，在Java中，一些不是对象的值是*n*ull、13、*tr*u*e*和4.0。 在Ruby中，每个表达式都会求值为一个对象。

- Ruby是基于类的：每个对象都是一个类的实例。 一个对象的类决定了它有哪些方法。（所有的代码都在方法中，方法类似于函数，它们接受参数并返回结果。）你可以在一个对象上调用一个方法，例如，obj.m(3,4)会将变量obj求值为一个对象，并调用它的m方法，参数为3和4。 并非所有的面向对象语言都是基于类的；例如，J*a*v*a*S*c*ript。

- Ruby有混入：下一个课程单元将描述混入，它在多重继承（如C++）和接口（如Java）之间取得了合理的折衷。每个Ruby类都有一个超类，但它可以包含任意数量的混入，与接口不同的是，混入可以定义方法（而不仅仅是要求它们的存在）。

- Ruby是动态类型的：就像R*a*c*k*e*t*允许使用任何参数调用任何函数一样，Ruby允许在任何对象上使用任何参数调用任何方法。 如果接收者（我们调用方法的对象）没有定义该方法，我们会得到一个动态错误。

- Ruby有许多动态特性：除了动态类型之外，Ruby还允许在程序执行时向对象添加和删除实例变量（在许多面向对象语言中称为字段），并且允许在程序执行时向类添加和删除方法。

- Ruby具有方便的反射：各种内置方法使得在运行时轻松发现对象的属性成为可能。 例如，每个对象都有一个返回对象类的方法 `class`，以及一个返回对象方法数组的方法 `methods`。

- Ruby有块和闭包：块几乎像闭包一样，在Ruby库中广泛用于方便的高阶编程。 事实上，在Ruby中很少使用显式循环，因为像A*rr*ay这样的集合类定义了许多有用的迭代器。 当你需要时，Ruby还具有完全强大的闭包。

- Ruby是一种脚本语言：对于什么使一种语言成为脚本语言没有明确的定义。 这意味着该语言的设计目标是使编写短程序变得容易，提供方便的文件和字符串操作（我们不会讨论这些主题），并且对性能没有太多关注。 像许多脚本语言一样，Ruby不要求在使用变量之前声明它们，并且通常有多种表达相同的意思的方式。

- Ruby在W*e*b应用程序中很受欢迎：Ruby *on* R*ai*l*s*框架是开发现代网站服务器端的热门选择。

回想一下，ML、Racket和Ruby三者结合起来涵盖了函数式与面向对象、静态与动态类型的四种组合中的三种。

我们的重点将放在Ruby的面向对象特性上，而不是它作为脚本语言的好处。 我们也不会讨论它对构建Web应用程序的支持，这是它目前如此受欢迎的主要原因。 作为一种面向对象的语言，Ruby与Smalltalk有很多共同之处，Smalltalk是一种自1980年以来基本没有改变的语言。 Ruby确实有一些不错的附加功能，比如mixin。

Ruby也是一种庞大的语言，对语法持有“为什么不”的态度。 ML和Racket（以及Smalltalk）相当严格地遵循某些传统的编程语言原则，比如定义一个小而强大的语言，然后程序员可以使用它来构建大型库。 Ruby经常持相反的观点。 例如，有很多不同的方法来编写if表达式。

基于类的面向对象编程规则

在学习特定Ruby结构的语法和语义之前，列举描述像Ruby和Smalltalk这样的语言的“规则”是有帮助的。Ruby中的一切都以面向对象编程来描述，我们简称为OOP，如下所示：

1. 所有值（通常是表达式的结果）都是对对象的引用。
2. 给定一个对象，代码通过调用它的方法与之通信。 调用方法的同义词是发送消息。 （在处理这样的消息时，对象可能会向其他对象发送其他消息，从而进行任意复杂的计算。）
3. 每个对象都有自己的私有状态。 只有对象的方法可以直接访问或更新这个状态。
4. 每个对象都是一个类的实例。
5. 对象的类决定了对象的行为。 类包含方法定义，规定了对象如何处理接收到的方法调用。

虽然这些规则在其他面向对象编程语言如Java或C#中也大部分成立，但Ruby对它们做出了更完整的承诺。 例如，在Java和C#中，一些值如数字不是对象（违反规则1），并且有办法使对象状态公开可见（违反规则3）。

对象，类，方法，变量等

（还可以参考与讲座材料一起发布的示例程序，这里没有重复的全部内容。）

类和方法定义

由于每个对象都有一个类，我们需要定义类，然后创建它们的实例（类C的对象是C的一个实例）。 （Ruby还在其语言和标准库中预定义了许多类。）创建一个带有方法 `m1`， `m2`，...， `mn`的类 `Foo`的基本语法（我们将逐步添加功能）可以是：

```
类Foo
   def m1
     ...
   结束

   def m2 (x,y)
     ...
```

```
    结束

    ...

    def mn z
      ...
    end
end
```

类名必须大写。 它们包括方法定义。 一个方法可以接受任意数量的参数，包括0个，并且我们为每个参数都有一个变量。 在上面的示例中， m1不接受任何参数， m2接受两个参数， mn接受一个参数。 这里没有显示方法体。 与ML和Racket函数一样，方法隐式返回其最后一个表达式。 与Java/C#/C++一样，当有帮助时，可以使用显式的 return语句立即返回。 （在方法末尾有一个返回是不好的风格，因为它可以在那里是隐式的。）

方法参数可以有默认值，这样调用者可以传递较少的实际参数，剩下的参数会用默认值填充。 如果一个方法参数有默认值，那么它右边的所有参数也必须有默认值。一个例子是：

```
    def myMethod（x，y，z = 0，w = “嗨”）
      ...
    结束
```

调用方法

方法调用e0.m(e1, ..., en)对对象 e0, e1, ..., en进行求值。 然后它调用 e0的结果（根据 e0的结果的类确定）中的方法 m，并将 e1, ..., en的结果作为参数传递。 至于语法，括号是可选的。特别是，零参数调用通常写作 e0.m，尽管e0.m()也可以。

要在当前执行方法的同一对象上调用另一个方法，可以写作self.m(...)或者 m(...)。 （Java/C#/C++的工作方式相同，只是它们使用关键字 this而不是 self。）

在面向对象编程中，方法调用的另一个常用名称是消息发送。 因此，我们可以说e0.m e1发送了 e0的消息 m，并且参数是 e1的结果。 这个术语是“更面向对象”的- 作为客户端，我们不关心接收者（消息的接收者）是如何实现的（例如，使用一个名为m的方法），只要它能处理这个消息就可以。 作为一般术语，在调用e0.m args时，我们称之为evaluating e0的结果（接收消息的对象）。

实例变量

一个对象有一个类，该类定义了它的方法。 它还有实例变量，用于存储值（即对象）。 许多语言（例如Java）使用术语字段来表示相同的概念。与Java/C#/C++不同，我们的类定义不指示类的实例将具有哪些实例变量。 要向对象添加一个实例变量，只需对其进行赋值：如果实例变量不存在，则会创建它。 所有实例变量都以 @开头，例如 @foo，以区别于方法局部变量。

每个对象都有自己的实例变量。 实例变量是可变的。 一个表达式（在方法体中）可以使用类似于 @foo的表达式读取实例变量，并使用类似于@foo = newValue的表达式写入实例变量。 实例变量对于对象是私有的。 没有直接访问其他对象的实例变量的方法。 所以， @foo引用当前对象的 @foo实例变量，即 self.@foo，但 self.@foo实际上不是合法的语法。

Ruby还有类变量（类似于Java的静态字段）。 它们被写作 @@foo。 类变量对于对象不是私有的。 相反，它们被所有类的实例共享，但仍然不能直接从不同类的对象访问。

构造一个对象

要创建类 Foo的新实例，你可以写Foo.new(...)其中 (...)包含一些参数（注意，与所有方法调用一样，括号是可选的，当参数为零或一个时，最好省略括号）。 调用 Foo.new将创建类 Foo的新实例，并在 Foo.new返回之前，使用传递给 Foo.new的所有参数调用新对象的 initialize方法。 也就是说， initialize方法是特殊的，起到了其他面向对象语言中构造函数的作用。

初始化的典型行为是创建和初始化实例变量。 事实上，初始化的正常方法是始终创建相同的实例变量，并且类中的其他方法不会创建实例变量。 但是Ruby不要求这样做，有时违反这些约定可能是有用的。 因此，类的不同实例可以有不同的实例变量。

表达式和局部变量

Ruby中的大多数表达式实际上是方法调用。 甚至e1 + e2只是e1.+ e2的语法糖，即调用 e1的结果上的 +方法，参数为 e2的结果。 另一个例子是puts e，它打印 e的结果（在调用其 to_s方法将其转换为字符串之后）然后换行。 事实证明 puts是所有对象的方法（它在 Object类中定义，所有类都是 Object的子类 - 我们稍后讨论子类），所以puts e只是self.puts e。

并非每个表达式都是一个方法调用。 最常见的其他表达式是某种条件形式。有各种不同的编写条件语句的方式；请参考讲座材料中发布的示例代码。 如下所讨论的，循环表达式在Ruby代码中很少见。

与实例变量一样，方法内部的局部变量不需要声明：在方法中第一次对x赋值将创建该变量。 变量的作用域是整个方法体。 在使用尚未定义的局部变量时会发生运行时错误。 （相比之下，使用尚未定义的实例变量不会导致运行时错误。而是返回 nil对象，这将在下面更详细地讨论。）

类常量和类方法

类常量与类变量非常相似（参见上文），只是（1）以大写字母开头而不是 @@，（2）不应该对其进行变异，（3）它是公开可见的。 在类 C的实例之外，可以使用 C::Foo语法访问类 C的常量 Foo。 一个例子是 Math::PI。[1]

类方法与普通方法（称为实例方法以区别于类方法）相似除了（1）它没有访问类的任何实例变量或实例方法的权限和（2）你可以在类的外部调用它 C在C.method_name args中定义。 有多种定义类方法的方式；最常见的是有点难以解释的语法：

```
def self.method_name args
  ...
结束
```

在Java和C#中，类方法被称为静态方法。

[1] 实际上， Math是一个模块，而不是一个类，所以这不是一个严格的例子，但是模块也可以有常量。

可见性和获取器/设置器

如上所述，实例变量对于对象是私有的：只有使用该对象作为接收者的方法调用才能读取或写入字段。 因此，语法是 `@foo`并且self对象是隐含的。 注意，即使是同一个类的其他实例也无法访问实例变量。 这非常面向对象：你只能通过发送消息与另一个对象进行交互。

方法可以有不同的可见性。 默认是 公共的，这意味着任何对象都可以调用该方法。
还有私有的，就像实例变量一样，只允许对象本身从其他方法中调用该方法。 中间是受保护的：受保护的方法可以被同一类的任何对象或该类的任何子类的对象调用。

有多种方法可以指定方法的可见性。 也许最简单的方法是在类定义中，可以在方法定义之间放置公共的、私有的或受保护的。 从上到下阅读，最近指定的可见性适用于下一个可见性指定之前的所有方法。 在类中的第一个方法之前有一个隐式的公共的。

为了使实例变量的内容可用和/或可变，我们可以轻松地定义getter和setter方法，按照惯例，我们可以给它们与实例变量相同的名称。例如：

```
def foo
  @foo
end

def foo= x
  @foo = x
end
```

如果这些方法是公共的，现在任何代码都可以通过调用 `foo`或`foo=`来间接访问实例变量 `@foo`。 如果只有同一个类（或子类）的其他对象才能访问实例变量，那么有时候将这些方法设为 `protected`是有意义的。

作为一种可爱的语法糖，在调用以 `=`字符结尾的方法时，你可以在 `=`之前加上空格。 因此，你可以写成`e.foo = bar`而不是`e.foo= bar`。

getter/setter方法的优点是它仍然是一个实现细节，这些方法被实现为获取和设置实例变量。 我们或者子类的实现者以后可以更改这个决定，而客户端不会知道。 我们还可以省略setter方法，以确保实例变量不被修改，除非是对象的一个方法。

作为一个不是真正的“setter方法”的例子，一个类可以定义：

```
def celsius_temp= x
  @kelvin_temp = x + 273.15
end
```

客户端可能会想象这个类有一个`@celsius_temp`实例变量，但实际上它（可能）没有。这是一个很好的抽象，允许实现进行更改。

由于getter和setter方法非常常见，所以有更短的语法来定义它们。 例如，要为实例变量 `@x`、 `@y`和 `@z`定义getter，并为 `@x`定义setter，类定义只需包含：

```
attr_reader :y, :z # 定义getter
attr_accessor :x # 定义getter和setter
```

最后一个语法细节：如果一个方法 m是私有的，你只能以 m或 m(args)的形式调用它。 像 x.m或x.m(args)这样的调用会违反可见性规则。 像 self.m或self.m(args)这样的调用不会违反可见性，但仍然不被允许。

一些语法、语义和作用域需要适应

Ruby有一些怪癖，通常方便快速编写有用的程序，但可能需要一些时间来适应。以下是一些例子；你肯定会发现更多。

- 条件表达式有几种形式，包括e1 if e2（都在一行上），只有在e2为真时才评估 e1（即，从右到左读取）。
- 换行符通常是有意义的。 例如，你可以写

```
if e1
  e2
else
  e3
end
```

  但如果你想将这一切放在一行上，那么你需要写if e1 then e2 else e3 end。 然而，注意，缩进从不重要（只是一种风格问题）。

- 条件可以对任何对象进行操作，并将每个对象视为“真”，只有两个例外： false 和 nil。
- 如上所述，你可以定义一个以 =结尾的方法，例如：

```
def foo= x
  @blah = x * 2
end
```

  正如预期的那样，你可以写 e.foo=(17)来将 e的 @blah实例变量更改为34。 更好的是，你可以调整括号和间距来写e.foo = 17。 这只是语法糖。 它“感觉”像一个赋值语句，但实际上是一个方法调用。 从风格上讲，你可以这样做来改变对象的状态（比如设置一个字段）的方法。

- 在Java/C#/C++中你写 this，在Ruby中你写 self。
- 记住，变量（局部、实例或类）通过赋值自动创建，所以如果你在赋值中拼写错误的变量，你最终只会创建一个不同的变量。

一切都是对象

一切都是对象，包括数字、布尔值和 nil（在Java中通常用作 null）。例如，-42.abs的结果是 42，因为 Fixnum类定义了计算绝对值的方法 abs，而-42是 Fixnum的一个实例。 （当然，这是一个愚蠢的表达式，但是 x.abs在 x当前持有 -42的情况下是合理的。）

所有对象都有一个 nil?方法， nil的类定义为返回 true，而其他类定义为返回 false。就像在ML和Racket中一样，每个表达式都会产生一个结果，但是当没有特定的结果时，nil是首选的风格（就像ML的()和Racket的void-object一样）。 尽管如此，方法通常会返回self，以便可以将后续的方法调用放在一起。

例如，如果 foo方法返回 self，则可以写成x.foo(14).bar("hi")而不是 当没有特定的结果时， nil是首选的风格（就像ML的 ()和Racket的void-object一样）。 例如，如果foo方法返回 self，则可以写成x.foo(14).bar("hi")而不是

```
x.foo(14)
x.bar（“嗨”）
```

有很多方法来支持反射- 在程序执行期间了解对象及其定义的方法。 例如，方法 methods返回一个对象上定义的方法的名称数组，而方法 class返回对象的类。[2] 这种反射有时在编写灵活的代码时很有用。 它在REPL或调试时也很有用。

顶层

您可以在显式类定义之外定义方法、变量等。 这些方法隐式地添加到类 Object中，使它们可以在任何对象的方法中使用。 因此，所有方法实际上都是某个类的一部分。[3]

程序运行时，顶层表达式按顺序进行评估。 因此，Ruby不需要指定一个特殊名称的主类和方法（如 main），您只需创建一个对象并在顶层调用其方法即可。

类定义是动态的

Ruby程序（或REPL的用户）可以在运行时更改类定义。
自然地，这会影响到类的所有用户。 令人惊讶的是，它甚至会影响到已经创建的类的实例。 也就是说，如果你创建了一个Foo的实例，然后在Foo中添加或删除方法，那么已经创建的对象会“看到”其行为的变化。 毕竟，每个对象都有一个类，当前的类定义决定了对象的行为。

这通常是可疑的风格，因为它破坏了抽象，但它使语言定义更简单：定义类和改变类的定义只是像其他操作一样的运行时操作。 它肯定会破坏程序：如果我改变或删除数字上的 +方法，我不会指望很多程序继续正确工作。 向现有类添加方法可能很有用，特别是如果类的设计者没有考虑到一个有用的辅助方法。

添加或更改方法的语法非常简单：只需给出一个包含方法定义的类定义，该类已经定义。 方法定义可以替换先前定义的方法（具有相同名称的方法名称），也可以添加到类中（如果先前不存在该方法）。

鸭子类型

鸭子类型是指表达式“如果它走起来像鸭子，叫起来像鸭子，那么它就是鸭子”，尽管更好的结论可能是“那么就没有理由担心它可能不是鸭子”。

[2]这个类本身只是另一个对象。 是的，甚至类也是对象。
[3]这并不完全正确，因为模块不是类。

在Ruby中，这指的是一个对象的类（例如，“Duck”）传递给一个方法的概念并不重要，只要对象能够响应它所期望的所有消息（例如，“走到x”或“现在叫”）。

例如，考虑这个方法：

```
def mirror_update pt
  pt.x = pt.x * -1
end
```

自然而然地，我们可以将其视为一个必须接受特定类的实例 Point（此处未显示）的方法，因为它使用了其中定义的方法 x和 x=。而且， x的getter方法必须返回一个数字，因为 pt.x的结果会被发送给带有 -1参数的 *消息。

但是，这个方法更具有普遍的实用性。 并不需要 pt是 Point的实例，只要它具有方法 x和 x=即可。

此外， x和 x=方法不一定是实例变量 @x的getter和setter。

更一般地说， x方法不需要返回一个数字。 它只需要返回一个能够响应带有参数 -1的 *消息的对象。

鸭子类型可以使代码更具可重用性，允许客户端创建“假鸭子”并仍然使用您的代码。
在Ruby中，鸭子类型基本上是“免费提供”的，只要你不明确检查参数是否是特定类的实例，使用像instance_of?或is_a?这样的方法（在我们介绍子类化时下面讨论）。

鸭子类型有缺点。 对于如何使用方法的最宽松的规范最终描述了方法的整个实现，特别是它向哪些对象发送了什么消息。 如果我们的规范都揭示了这一点，那么几乎没有任何实现的变体是等价的。 例如，如果我们知道i是一个数字（并忽略客户端在数字类中重新定义方法），那么我们可以用i*2或2*i替换i+i。 但是，如果我们只是假设i可以接收自身作为参数的+消息，那么我们不能进行这些替换，因为i可能没有*方法（破坏i*2），或者它可能不是2期望作为*参数的对象类型（破坏2*i）。

数组

在Ruby程序中，Array类非常常用，而且通常会有特殊的语法与之一起使用。Array的实例具有其他编程语言中数组的所有用途，而且更加灵活和动态。 与Java/C#/C等语言中的数组相比，Ruby中的数组更加灵活和动态，错误操作更少。 不过，它们可能不够高效，但这通常不是Ruby中方便编程的问题。简而言之，所有Ruby程序员都熟悉Ruby数组，因为它们是任何对象集合的标准选择。

一般来说，数组是从数字（索引）到对象的映射。 语法[e1,e2,e3,e4]创建一个包含四个对象的新数组：e1的结果在索引0，e2的结果在索引1，依此类推。（注意索引从0开始。）还有其他创建数组的方法。 例如，Array.new(x)创建一个长度为x的数组，每个索引最初都映射到nil。 我们还可以将块（见下文块的实际含义）传递给Array.new方法来初始化数组元素。 例如，Array.new(x) { 0 }创建一个长度为x的数组，所有元素都初始化为0，Array.new(5) {|i| -i }创建数组[0,-1,-2,-3,-4]。

获取和设置数组元素的语法与许多其他编程语言类似：表达式 a[i]获取数组 a中索引为 i的元素，a[i] = e将相同的数组元素设置为e

索引。 正如你可能猜到的，在Ruby中，当我们使用这种语法时，实际上是在 Array类上调用方法。

以下是一些简单的方式，Ruby数组比你预期的更具动态性和更少的错误：

- 像通常在动态类型语言中一样，数组可以容纳不同类的对象，例如[14, "hi", false, 34]。
- 负数数组索引从数组的末尾解释。 因此， a[-1]检索数组 a中的最后一个元素， a[-2]检索倒数第二个元素，依此类推。
- 没有数组边界错误。 对于表达式 a[i]，如果 a中的元素少于 i+1个，则结果将为 nil。 设置这样的索引更有趣：对于 a[i]=e，如果 a中的元素少于 i+1个，则数组将动态增长以容纳 i+1个元素，其中最后一个元素将是 e的结果，在旧的最后一个元素和新的最后一个元素之间有正确数量的 nil对象。
- 标准库中定义了许多数组的方法和操作。 如果你需要在数组上执行的操作是通用的，请查阅文档，因为它肯定已经提供了。 作为两个例子， +运算符被定义为数组的连接（一个新数组，其中左操作数的所有元素在右操作数的所有元素之前），而 |运算符类似于 +运算符，只是它从结果中删除所有重复的元素。

除了所有常规的数组用法之外，Ruby数组还经常用于其他语言中我们会使用元组、栈或队列的地方。 元组是最直接的用法。 毕竟，鉴于动态类型和对效率的较少关注，没有理由为元组和数组使用单独的结构。 例如，对于一个三元组，只需使用一个包含3个元素的数组。

对于堆栈， Array类定义了方便的方法 push和 pop。 前者接受一个参数，将数组增加一个索引，并将参数放置在新的最后一个索引位置。 后者将数组减小一个索引，并返回原来最后一个索引位置的元素。 总之，这正是后进先出的行为，这定义了堆栈的行为。 （如何在实际增加和减小元素的底层存储方面实现这一点，只关系到 Array的实现。）

对于队列，我们可以使用 push添加元素，如上所述，并使用 shift方法出队列元素。 shift方法返回数组索引0处的对象，将其从数组中删除，并将所有其他元素向下移动一个索引，即，之前在索引1处的对象现在在索引0处，依此类推。虽然对于简单队列不需要， Array还有一个 unshift方法，类似于 push，但它将新对象放在索引0处，并将所有其他对象向上移动1个索引（增加数组大小1个）。

数组比这里描述的更加灵活。 例如，有一些操作可以用其他数组的元素替换数组元素的任意序列，即使其他数组的长度与被替换的序列不同（从而改变数组的长度）。

总的来说，数组大小的灵活处理（增长和缩小）与将数组视为从数字索引到对象的映射是不同于其他一些编程语言的，但与之一致。

到目前为止，我们还没有展示使用数组的所有内容进行一些计算的操作，例如映射元素以创建一个新数组，或计算它们的总和。 这是因为Ruby中用于此类计算的习惯用法使用块，我们将在下一节介绍。

传递块

虽然Ruby有while循环和for循环，与Java类似，但大多数Ruby代码不使用它们。 相反，许多类具有接受块的方法。 这些块几乎是闭包。 例如，整数有一个times方法，它接受一个块并执行它所想象的次数。例如，

```
x.times { puts “嗨” }
```

如果在环境中将x绑定到3，则打印"hi" 3次。

块是闭包，因为它们可以引用块定义时作用域中的变量。 例如，在此程序执行后，y将绑定到10：

```
y = 7
[4,6,8].each { y += 1 }
```

这里 [4,6,8]是一个有3个元素的数组。 数组有一个方法each，它接受一个块并对每个元素执行一次。 通常，我们希望将块传递给每个数组元素。 我们可以这样做，例如对数组的元素求和并在每个点打印出运行总和：

```
sum = 0
[4,6,8].each { |x|
  sum += x
  puts sum
}
```

令人惊讶的是，块不是对象。 您不能将它们作为“常规”参数传递给方法。 相反，任何方法都可以传递0个或1个块，与其他参数分开。 如上面的示例所示，块只需放在方法调用的右侧。 它也在任何其他“常规”参数之后。
例如， inject方法类似于我们在ML中学习的 fold函数，我们可以将初始累加器作为常规参数传递给它：

```
sum = [4,6,8].inject(0) { |acc,elt| acc + elt }
```

（事实证明，初始累加器是可选的。如果省略，该方法将使用索引为0的数组元素作为初始累加器。）

除了这里显示的大括号语法，你还可以使用 do来编写块，而不是 {和 end而是
}。对于超过一行的块，这通常被认为是更好的风格。

在调用一个带有块的方法时，你应该知道在调用时将传递给块的参数数量。 对于 Array中的 each方法，答案是1，但正如第一个例子所示，如果你不需要它们，可以通过省略 |...|来忽略参数。

许多集合，包括数组，在功能编程中有各种块接受方法，包括 map。 作为另一个例子， select方法类似于我们调用的函数filter。 其他有用的迭代器包括 any?。（如果块对集合的任何元素返回true，则返回true），all?。（如果块对集合的每个元素返回true，则返回true），还有其他几个。

使用块

虽然许多块的用途涉及调用标准库中的方法，但您也可以定义自己的
接受块的方法。 （大型标准库使这种情况相对较少。）

您可以将块传递给任何方法。 方法体使用 yield关键字调用块。 例如，此代码会打印 "hi"3次：

```
def foo x
  if x
    yield
  else
    yield
    yield
  end
end
foo true { puts "hi" }
foo false { puts "hi" }
```

要将参数传递给一个块，你需要在 yield之后放置参数，例如yield 7或yield(8,"str")。

使用这种方法，一个方法可能期望一个块是隐含的；只是它的主体可能会使用 yield。 如果使用了 yield但没有传递块，将会导致错误。 当块和 yield在参数数量上不一致时的行为是相对灵活的，这里没有详细描述。
一个方法可以使用block_given?原始的方法可以用来判断调用者是否提供了一个块。 你不太可能经常使用这个方法：如果需要一个块，通常假设它已经给出，并且如果没有给出，则使用yield失败。在可能或可能不需要一个块的情况下，通常其他常规参数决定是否应该存在一个块。 如果没有，则适用block_given?。

这是一个递归方法，它在块返回一个true结果之前调用块的次数（使用递增的数字）计数。

```
def count i
    if yield i
      1
    else
      1 + (count(i+1) {|x| yield x})
    end
  end
```

奇怪的是，没有直接的方法将调用者的块作为被调用者的块参数传递。 但是，我们可以创建一个新的块{|x | yield x}并且在其体内的yield的词法作用域会做正确的事情。 如果块实际上是我们可以作为对象传递的函数闭包，那么这将是不必要的函数包装。

Proc类

块不完全是闭包，因为它们不是对象。 我们不能将它们存储在字段中，将它们作为常规方法参数传递，将它们赋值给变量，将它们放入数组中等等（请注意，在ML和Racket中，我们可以使用闭包进行等效操作）。因此，我们说块不是“一等值”，因为一等值是可以像语言中的其他任何东西一样传递和存储的东西。

然而，Ruby也有“真正”的闭包：类 Prochas的实例是闭包。 在Proc中的方法 call是将闭包应用于参数的方式，例如 x.call（对于无参数）或x.call(3,4)。要将块转换为Proc，可以编写lambda { ... }其中{ ... }是任何块。 有趣的是，lambda不是关键字。 它只是类 Object中的一个方法（每个类都是 Object的子类，因此lambda在任何地方都可用），它将传递给它的块转换为 Proc。 您也可以定义自己的方法来执行此操作；请参阅文档以了解如何执行此操作的语法。

通常，我们只需要块，例如将块传递给计算数组的这些示例：

```
a = [3,5,7,9]
b = a.map {|x| x + 1}
i = b.count {|x| x >= 6}
```

但是假设我们想要创建一个块的数组，即一个每个元素都是我们可以用一个值“调用”的东西的数组。 在Ruby中你不能这样做，因为数组保存的是对象，而块不是对象。
所以这是一个错误：

```
c = a.map {|x| {|y| x >= y} } # 错误，语法错误
```

但是我们可以使用 lambda来创建一个 Proc实例的数组：

```
c = a.map {|x| lambda {|y| x >= y} }
```

现在我们可以向 c数组的元素发送 call消息：

```
c[2].call 17
j = c.count {|x| x.call(5) }
```

Ruby的设计与ML和Racket形成了有趣的对比，后者只提供完全闭包作为自然选择。 在Ruby中，块比 Proc对象更方便使用，在大多数情况下足够使用，但是程序员在需要时仍然可以使用 Proc对象。 是将块与闭包区分开来，并通过一个功能较弱的构造使更常见的情况更容易，还是只有一个完全强大的通用特性更好？

哈希和范围

Hash和Range类是两个标准库类，也非常常见，但可能比数组稍微少见一些。 与数组类似，它们有特殊的内置语法。 它们也与数组类似，并支持许多相同的迭代器方法，这有助于我们强化“如何迭代”与“迭代时要做什么”的概念分离。

哈希类似于数组，但映射不是从数值索引到对象，而是从（任意）对象到对象。 因此，映射是从（任意）对象到对象的集合。 如果a映射到b，我们称a为键，b为值。 因此，哈希是一个将一组键（哈希中的所有键都是不同的）映射到值的集合，其中键和值只是对象。 我们可以使用以下语法创建一个哈希：

```
{"SML" => 7, "Racket" => 12, "Ruby" => 42}
```

正如你所预期的那样，这将创建一个具有字符串键的哈希。 通常（并且更加高效）使用Ruby的符号作为哈希键，如下所示：

```
{:sml => 7, :racket => 12, :ruby => 42}
```

我们可以使用与数组相同的语法在哈希中获取和设置值，其中键可以是任何东西，例如：

```
h1["a"] = "找到了A"
h1[false] = "找到了false"
h1["a"]
h1[false]
h1[42]
```

哈希上定义了许多方法。 有用的方法包括 keys（返回所有键的数组）， values（类似于values），和 delete（给定一个键，从哈希中删除它及其值）。 哈希也支持与数组相同的迭代器，例如 each和 inject，但某些迭代器需要键和值作为参数，请参考文档。

一个范围表示一系列连续的数字（或其他东西，但我们将重点关注数字）。 例如， 1..100表示整数1、2、3、...、100。 我们可以使用类似Array.new(100){|i| i}的数组，但范围的表示更高效，并且，如 1..100所示，有更方便的语法来创建它们。 虽然通常有更好的迭代器可用，但类似于(0..n).each{|i| e}的方法调用在其他编程语言中很像从0到n的for循环。

值得强调的是，鸭子类型让我们可以在许多地方使用范围，而我们可能自然地期望使用数组。 例如，考虑这个方法，它计算 a中平方小于50的元素数量：

```
def foo a
  a.count {|x| x*x < 50}
end
```

我们可能自然地期望 foo接受数组，并且像foo [3,5,7,9]这样的调用按预期工作。 但我们可以将具有期望一个参数的块的 count方法的任何对象传递给 foo。 所以我们也可以执行foo (2..10)，它的结果是6。

子类和继承

基本思想和术语

子类是基于类的面向对象编程的一个重要特性。如果类 C是类 D的子类，那么 C的每个实例也是 D的实例。类 C的定义继承了类 D的方法，也就是说，它们也是类 C的定义的一部分。此外，类 C可以通过定义新的方法来扩展，这些方法是类 C拥有但类 D没有的。它还可以通过改变继承的方法的定义来覆盖方法。 在Ruby中，这与Java类似。在Java中，子类也继承了超类的字段定义，但在Ruby中，字段（即实例变量）不是类定义的一部分，因为每个对象实例都会创建自己的实例变量。

Ruby中的每个类除了 Object之外都有一个超类。[4]这些类形成了一棵树，其中每个节点都是一个类，父节点是它的超类。 类 Object是树的根节点。 在基于类的语言中，这是

[4]实际上， Object的超类是BasicObject，BasicObject没有超类，但这不是一个重要的细节，所以我们将忽略它。

称为类层次结构。 根据子类的定义，一个类在树中（即，它和根节点之间的所有节点，包括它自己）拥有所有祖先的方法，可以被覆盖。

一些*Ruby*特点

- Ruby类定义使用`class C < D ...` 指定一个超类 `end`用于定义一个新类 `C`，其超类为 `D`。 省略`< D`意味着`< Object`，这是我们迄今为止的例子所做的。
- Ruby的内置反射方法可以帮助您探索类层次结构。 每个对象都有一个类方法，返回对象的类。 一致地，尽管一开始可能会让人困惑，但在Ruby中，类本身也是一个对象（毕竟，每个值都是一个对象）。 一个类的类是 `Class`。 这个类定义了一个方法 `superclass`，返回超类。

- 每个对象也有方法 `is_a?`和`instance_of?`。方法is_a?接受一个类（例如，`x.is_a? Integer`），如果接收者是 `Integer`或任何（传递性）子类的实例，则返回trueof `Integer`，即，如果它在类层次结构中低于 `Integer`。方法instance_of?类似，但只有当接收者是该类的实例时才返回true，而不是子类。 （请注意，在Java中，原始 `instanceof`类似于Ruby的 `is_a?`。）

使用像 `is_a?`这样的方法和 `instanceof`是“不太面向对象”的，因此通常不是首选的风格。
它们与鸭子类型相冲突。

第一个例子：点和彩色点

这里是描述简单的二维点和一个子类（只是用字符串表示颜色）的简单类的定义。

```
class Point
  attr_accessor :x, :y
  def initialize(x,y)
    @x = x
    @y = y
  end
  def distFromOrigin
    Math.sqrt(@x * @x   + @y * @y)
  end
  def distFromOrigin2
    Math.sqrt(x * x + y * y)
  end
end
class ColorPoint < Point
  attr_accessor :color
  def initialize(x,y,c="clear")
    super(x,y)
    @color = c
  end
end
```

有很多种方式我们可以定义这些类。 我们在这里的设计选择包括：

- 我们使 `@x`, `@y`, 和 `@color`实例变量可变，具有公共的getter和setter方法。
- 一个 `ColorPoint`的默认“color”是 `"clear"`。

- 出于教学目的，我们以两种不同的方式实现到原点的距离。
  方法distFromOrigin直接访问实例变量，而方法distFromOrigin2使用了
  self的getter方法。 根据 Point的定义，两者将产生相同的结果。

在 ColorPoint的 initialize方法中使用了 super关键字，它允许重写的方法调用超类中同名的方法。 这在构造Ruby对象时不是必需的，但通常是希望的。

为什么使用子类化?

我们现在考虑使用类 Point的子类来定义有颜色的点的风格，如上所示。 事实证明，在这种情况下，这是良好的面向对象编程风格。 定义 ColorPoint是一个良好的风格，因为它允许我们重用 Point的大部分工作，并且将任何 ColorPoint的实例视为“是一个” Point是有意义的。

但是，有几种值得探索的替代方案，因为在面向对象的程序中，子类化经常被过度使用，所以在程序设计时考虑替代方案是否比子类化更好是值得的。

首先，在Ruby中，我们可以使用新方法扩展和修改类。 因此，我们可以通过替换 Point类的 initialize方法并为 @color添加getter/setter方法来简单地更改它。 只有当每个 Point对象，包括 Point的所有其他子类的实例，都应该有一个颜色，或者至少有一个颜色不会对我们的程序造成任何问题时，这才是合适的。 通常修改类不是一种模块化的改变 - 只有在您知道它不会对使用该类的程序产生负面影响时才应该这样做。

其次，我们可以从头开始定义 ColorPoint，将代码从 Point复制过来（或重新输入）。在动态类型语言中，语义上的差异（而不是风格上的差异）很小：如果发送了 is_a?消息给ColorPoint的实例，它们将返回false。如果传入参数 Point，它们将以相同的方式工作。 在像Java/C#/C++这样的语言中，超类对静态类型有影响。 不子类化 Point的一个优点是，对 Point的任何后续更改都不会影响 ColorPoint——通常在基于类的面向对象编程中，我们必须担心如何影响子类的类的更改。

第三，我们可以将 ColorPoint作为 Object的子类，但它包含一个实例变量，称为@pt，保存一个 Point的实例。 然后，它需要定义 Point中定义的所有方法，将消息转发给 @pt中的对象。 这里有两个例子，省略了所有其他方法（x=, y, y=,distFromOrigin,distFromOrigin2）：

```
def initialize(x,y,c="清除")
  @pt = Point.new(x,y)
  @color = c
end
def x
  @pt.x # 将消息转发给@pt对象
end
```

这种方法风格不好，因为子类化更简短，我们希望将 ColorPoint视为“是一个” Point。 但是一般来说，许多面向对象语言的程序员过度使用子类化。 在将一个新的数据类型作为现有数据类型的单独子部分包含的情况下，这种实例变量的方法更好。

覆盖和动态分派

现在让我们考虑一个不同的 Point子类，用于三维点：

```
class ThreeDPoint < Point
  attr_accessor :z
  def initialize(x,y,z)
    super(x,y)
    @z = z
  end
  def distFromOrigin
    d = super
    Math.sqrt(d * d + @z * @z)
  end
  def distFromOrigin2
    d = super
    Math.sqrt(d * d + z * z)
  end
end
```

在这里，代码重用的优势仅限于继承方法 x, x=, y和 y=，以及通过super使用其他 Point中的方法。 请注意，除了覆盖 initialize之外，我们还对distFromOrigin和distFromOrigin2使用了覆盖。

计算机科学家们几十年来一直在争论这种子类化是否是良好的风格。 一方面，它确实让我们重用了相当多的代码。 另一方面，有人可能会认为一个ThreeDPoint不是概念上的（二维） Point，因此在某些代码期望后者时传递前者可能是不合适的。 其他人说一个ThreeDPoint是一个 Point，因为你可以将其视为在 z等于0的平面上的投影。 我们不会解决这个传奇的争论，但你应该明白，尽管它让你在超类中重用了一些代码，但通常子类化是不好/令人困惑的风格。

如果我们在Point类中有一个名为distance的方法，它接受另一个（行为类似于）Point的对象作为参数，并计算参数和self之间的距离，那么反对子类化的论点就更加有力。
如果ThreeDPoint想要用一个接受另一个（行为类似于）ThreeDPoint的对象作为参数的方法来覆盖这个方法，那么ThreeDPoint的实例将不会像Point的实例那样行为：当传递一个Point的实例时，它们的distance方法将失败。

现在我们考虑一个更加有趣的Point的子类。 这个类的实例PolarPoint的行为与Point的实例等效，除了initialize方法的参数之外，但实例使用极坐标（半径和角度）的内部表示：

```
class PolarPoint < Point
  def initialize(r,theta)
    @r = r
    @theta = theta
  end
  def x
    @r * Math.cos(@theta)
  end
  def y
    @r * Math.sin(@theta)
  end
```

```
  def x= a
    b = y # 避免多次调用y方法
    @theta = Math.atan(b / a)
    @r = Math.sqrt(a*a + b*b)
    self
  end
  def y= b
    a = y # 避免多次调用y方法
    @theta = Math.atan(b / a)
    @r = Math.sqrt(a*a + b*b)
    self
  end
  def distFromOrigin
    @r
  end
  # distFromOrigin2 已经可以工作了!!
end
```

注意，PolarPoint的实例没有实例变量@x和@y，但是该类重写了x，x=，y和y=方法，以便客户端无法判断实现是否不同（除了浮点数的四舍五入）：他们可以互换使用Point和PolarPoint的实例。 在Java中，类似的例子仍然会有来自超类的字段，但是不会使用它们。 PolarPoint相对于Point的优势，虽然仅仅是为了举例，是distFromOrigin更简单和更高效。

这个例子的关键点是子类没有覆盖distFromOrigin2，但是继承的方法仍然正常工作。 为了理解这一点，考虑超类中的定义：

```
  def distFromOrigin2
    Math.sqrt(x * x + y * y)
  end
```

与distFromOrigin的定义不同，这个方法使用其他方法调用作为乘法的参数。 回想一下，这只是语法糖的一种形式：

```
  def distFromOrigin2
    Math.sqrt(self.x() * self.x() + self.y() * self.y())
  end
```

在超类中，这似乎是一个不必要的复杂化，因为 self.x()只是一个返回 @x的方法，而 Point的方法可以直接访问 @x，就像distFromOrigin一样。

然而，在 Point的子类中覆盖方法 x和 y会改变distFromOrigin2在子类实例中的行为。 给定一个 PolarPo int实例，它的distFromOrigin2方法是用上面的代码定义的，但是当调用时， self.x和 self.y将调用在 PolarPoint中定义的方法，而不是在 Point中定义的方法。

这个语义有很多名字，包括动态分派、迟绑定和虚拟方法调用。
在函数式编程中没有任何东西可以与之相比，因为 self在环境中的处理方式是特殊的，我们将在下面更详细地讨论。

方法查找的精确定义

这个讨论的目的是仔细考虑面向对象语言构造的语义，特别是对方法的调用，就像我们仔细考虑函数式语言构造的语义一样，特别是对闭包的调用。 正如我们将看到的，关键的区别特征是在调用方法时， self在环境中绑定到什么。 正确的定义是我们所称之为动态分派。

我们将逐步建立起的基本问题是，在存在覆盖的情况下，给定一个调用 e0.m(e1,e2,...en)，我们如何“查找”我们所调用的方法定义 m的规则，这是一个非平凡的问题。 但首先，让我们注意到，通常关于如何“查找”某个东西的这类问题对于编程语言的语义来说是至关重要的。 例如，在ML和Racket中，查找变量的规则导致了词法作用域和函数闭包的正确处理。 而在Racket中，我们有三种不同的let表达式，正是因为它们在某些子表达式中对变量的查找有不同的语义。

在Ruby中，方法和块中的局部变量的变量查找规则与ML和Racket中的规则并没有太大的不同，尽管有些奇怪的地方是变量在使用之前并没有被声明。 但是我们还需要考虑如何“查找”实例变量、类变量和方法。 在所有情况下，答案取决于绑定到self的对象 - self被特殊对待。

在任何环境中，self映射到某个对象，我们将其视为“当前对象” - 当前执行方法的对象。 要查找实例变量@x，我们使用绑定到self的对象 - 每个对象都有自己的状态，我们使用self的状态。 要查找类变量@@x，我们只需使用绑定到self.class的对象的状态。 查找方法m的方法调用更加复杂...

在基于类的面向对象语言（如Ruby）中，评估方法调用e0.m(e1,...,en)的规则是：

- 将e0、e1、...、en评估为值，即对象obj0、obj1、...、objn。
- 获取obj0的类。 每个对象在运行时“知道它的类”。 将类视为obj0的一部分状态。
- 假设obj0的类是A。 如果m在A中定义，则调用该方法。 否则，使用A的超类递归地查看是否定义了m。 如果A及其任何超类都没有定义m，则引发“方法丢失”错误。 （实际上，在Ruby中，规则是调用一个名为method_missing的方法，任何类都可以定义该方法，因此我们再次从A及其超类开始查找。但大多数类都不定义method_missing，它在Object中的定义会引发我们期望的错误。）
- 我们现在找到了调用的方法。 如果方法有形式参数（即，参数名称或参数） x1， x2，... ， xn，那么评估主体的环境将映射 x1到 obj1，x2到 obj2等等。但是还有一件事是面向对象编程的本质，在函数式编程中没有真正的类似物：我们总是在环境中有 self。 在评估方法主体时， self绑定到 obj0，即消息的“接收者”。

如上所述，调用方中的 self的绑定是指“晚绑定”，“动态分派”和“虚方法调用”的同义词。这是Ruby和其他面向对象编程语言的语义的核心。 这意味着当 m的主体在 self上调用一个方法时（例如，self.someMethod 34或只是someMethod 34），我们使用 obj0的类来解析 someMethod，而不一定是我们正在执行的方法的类。 这就是为什么上面描述的 PolarPoint类的工作方式。

关于这个语义有几个重要的评论：

- Ruby的混入使查找规则变得更加复杂，所以上述规则实际上通过
  忽略混入来简化。 当我们学习混入时，我们将相应地修改方法查找的语义。

- 这个语义比ML/Racket函数调用要复杂得多。 如果你先学习了面向对象编程和动态分派，这可能看起来并不是那样，因为它们似乎是许多入门编程课程的重点。 但实际上它更加复杂：我们必须将self的概念与语言中的其他内容区分对待。 复杂并不一定意味着它是次等的或者优越的；它只是意味着语言定义中有更多需要描述的细节。

  这个语义显然对许多人来说是有用的。

- Java和C#有更加复杂的方法查找规则。 它们确实具有这里描述的动态分派，因此学习Ruby应该有助于理解这些语言中的方法查找语义。 但是它们也具有静态重载，即类可以有多个具有相同名称但接受不同类型（或数量）参数的方法。 因此，我们不仅需要找到一个具有正确名称的方法，还需要找到一个与调用处的参数类型匹配的方法。 此外，可能有多个方法匹配，并且语言规范中有一长串复杂的规则来找到最佳匹配（或者如果没有最佳匹配则给出类型错误）。 在这些语言中，只有当一个方法的参数具有相同的类型和数量时，它才会覆盖另一个方法。 在Ruby中，这些都不会出现，“相同的方法名称”总是意味着覆盖，并且我们没有静态类型系统。 在C++中，还有更多的可能性：我们有静态重载和不支持动态分派的不同形式的方法。

动态分派与闭包

为了理解动态分派与我们用于函数调用的词法作用域的区别，考虑下面这段简单的ML代码，它定义了两个互相递归的函数：

```
fun even x = if x=0 then true else odd (x-1)
and odd  x = if x=0 then false else even (x-1)
```

这创建了两个闭包，它们的环境中都有另一个闭包。 如果我们稍后用其他东西（例如）来隐藏 even闭包，

```
fun even x = false
```

那不会改变 odd的行为。 当 odd在定义 odd的环境中查找 even时，它将得到上面第一行的函数。 这对于仅仅通过查找定义的位置来理解 odd的工作方式是“好”的。 另一方面，假设我们写了一个更好的版本的 even，如下所示：

```
fun even x = (x mod 2) = 0
```

现在我们的奇数不再从这个优化的实现中受益。

在面向对象编程中，我们可以使用（滥用？）子类化、重写和动态分派来改变奇数的行为，通过重写 even：

```
类 A
  def even x
     if x==0 then true else odd(x-1) end
  end
```

```
  def odd x
     if x==0 then false else even(x-1) end
  end
end
class B < A
  def even x # 改变了B的odd!
     x % 2 == 0
  end
end
```

现在(B.new.odd 17)将会更快地执行，因为 odd对 even的调用将解析为 B中的方法 - 这都是因为环境中 self绑定的结果。 虽然在上面的简短示例中这当然很方便，但它也有真正的缺点。 我们无法查看一个类（A）并知道对该代码的调用将如何行为。 在子类中，如果有人重写了 even并且不知道这将改变 odd的行为，那该怎么办？ 基本上，对可能被重写的方法的调用需要非常谨慎地考虑。 很可能最好的方法是拥有无法被重写的私有方法，以避免问题。 然而，重写和动态分派是区分面向对象编程和函数式编程的最重要的特点。

在Racket中手动实现动态调度

现在让我们考虑使用Racket编写对象和动态调度的代码，只使用对和函数。[5]这有两个目的：

- 它演示了一个语言的语义（如消息发送等原语在语言中的工作方式）通常可以在另一种语言中通过一些辅助函数来模拟（模拟相同的行为）的习惯用法。 这可以帮助您成为在可能没有您习惯的功能的不同语言中更好的程序员。

- 它以较低级的方式来理解动态调度的“工作原理”，通过看看我们如何在另一种语言中手动执行它。 面向对象语言的解释器必须对语言中的程序进行类似的自动评估。

还要注意，我们之前进行了类似的练习，以更好地理解闭包：我们展示了如何使用对象和接口或使用函数指针和显式环境在Java或C中获得闭包的效果。

我们的方法与Ruby（或Java）实际上在以下几个方面有所不同：

- 我们的对象只包含一个字段列表和一个方法列表。 这不是“基于类”的，其中一个对象将具有一个字段列表和一个类名，然后类将具有方法列表。
  我们本可以用这种方式来实现。
- 真正的实现更高效。 它们使用更好的数据结构（基于数组或哈希表）来存储字段和方法，而不是简单的关联列表。

尽管如此，实现动态分派的关键思想仍然得以体现。 顺便说一下，我们明智地选择在Racket而不是ML中进行这个实现，因为类型会妨碍我们。在ML中，我们可能会

[5]虽然我们没有研究过，Racket有类和对象，所以你实际上不会想在Racket中这样做。关键是通过手动编码相同的思想来理解动态分派。

最终使用“一个大的数据类型”来给所有对象和它们的所有字段赋予相同的类型，这基本上是在ML中以Racket的方式编程的一种尴尬的方式。（相反，类型化的面向对象编程语言通常对ML风格的编程也不友好，除非它们为泛型类型和闭包添加单独的构造。）

我们的对象只有字段和方法：

```
(结构体 obj (fields methods))
```

我们将使用字段来保存一个不可变列表，其中包含可变对，每个元素对都是一个符号（字段名称）和一个值（当前字段内容）。 有了这个，我们可以定义帮助函数 get和 set，它们接受一个对象和一个字段名称，适当地返回或修改字段。 请注意，这些只是普通的Racket函数，没有特殊的功能或语言扩展。 我们确实需要定义自己的函数，称为assoc-m，因为Racket的 assoc期望一个不可变的对的不可变列表。

```
(define (assoc-m v xs)
  (cond [(null? xs) #f]
        [(equal? v (mcar (car xs))) (car xs)]
        [#t (assoc-m v (cdr xs))]))

(define (get obj fld)
  (let ([pr (assoc-m fld (obj-fields obj))])
    (if pr
        (mcdr pr)
        (error "字段未找到"))))

(define (set obj fld v)
  (let ([pr (assoc-m fld (obj-fields obj))])
    (if pr
        (set-mcdr! pr v)
        (error "字段未找到"))))
```

更有趣的是调用一个方法。 方法字段也将是一个关联列表，将方法名称映射到函数（不需要变异，因为我们比Ruby更不动态）。使动态分派工作的关键是，这些函数都将接受一个额外的显式参数，该参数在具有内置动态分派支持的语言中是隐式的。 这个参数将是“self”，我们的Racket帮助函数用于发送消息，只需传入正确的对象：

```
(define (send obj msg . args)
  (let ([pr (assoc msg (obj-methods obj))])
    (if pr
        ((cdr pr) obj args)
        (error "方法未找到" msg))))
```

请注意，我们用于方法的函数会传递“整个”对象 obj，该对象将用于绑定到 self的任何发送。（上面的代码使用Racket对可变参数函数的支持，因为这很方便 - 如果有必要，我们可以避免使用它。在这里， send可以接受大于或等于2的任意数量的参数。第一个参数绑定到 obj，第二个参数绑定到 msg，而其他所有参数都被放在一个列表中（按顺序），该列表绑定到 args。因此，我们期望(cdr pr)是一个接受两个参数的函数：我们将 obj作为第一个参数传递，将列表 args作为第二个参数传递。）

现在我们可以定义 make-point，它只是一个产生点对象的Racket函数：

```
(定义 (make-point _x _y)
  (对象
   (列表 (mcons 'x _x)
         (mcons 'y _y))
   (列表 (cons 'get-x (lambda (self args) (get self 'x)))
         (cons 'get-y (lambda (self args) (get self 'y)))
         (cons 'set-x (lambda (self args) (set self 'x (car args))))
         (cons 'set-y (lambda (self args) (set self 'y (car args))))
         (cons 'distToOrigin
               (lambda (self args)
                    (让 ([a (send self 'get-x)]
                         [b (send self 'get-y)])
                      (sqrt (+ (* a a) (* b b)))))))))
```

注意每个方法都有一个第一个参数，我们刚好称之为 self，在这里在Racket中没有特殊含义。然后我们使用 self作为 get、 set和 send的参数。 如果我们有其他对象想要发送消息或访问字段，我们只需将该对象放入 args列表中传递给我们的辅助函数。 一般来说，每个函数的第二个参数是我们面向对象思维中的“真实参数”列表。

通过使用 get， set和 send函数，我们定义了点，并且使用起来就像面向对象编程一样：

```
(定义 p1 (创建点 4 0))
(send p1 'get-x)           ; 4
(send p1 'get-y)           ; 0
(send p1 'distToOrigin) ; 4
(send p1 'set-y 3)
(send p1 'distToOrigin) ; 5
```

现在让我们模拟子类化...

我们的对象编码不使用类，但我们仍然可以创建一个重用定义点所使用的代码的东西。 这是用于创建具有颜色字段和获取器/设置器方法的点的代码。 关键思想是通过构造函数使用 make-point创建一个点对象，然后通过创建一个具有额外字段和方法的新对象来扩展该对象：

```
(定义 (创建彩色点 _x _y _c)
 (让 ([pt (创建点 _x _y)])
(对象
 (cons (mcons 'color _c)
       (对象字段 pt))
 (连接 (列表
          (cons 'get-color (lambda (self args) (get self 'color)))
          (cons 'set-color (lambda (self args) (set self 'color (car args)))))
       (对象方法 pt)))))
```

我们可以像使用从 make-point返回的“对象”一样使用从make-color-point返回的“对象”，
此外，我们还可以使用字段 color和方法 get-color和 set-color。

面向对象编程的基本区别特征是动态分派。 我们的对象编码“正确地实现了动态分派”，但我们的示例尚未展示出来。为了展示，我们需要在“超类”中定义一个“方法”

来调用由“子类”定义/覆盖的方法。就像我们在Ruby中所做的那样，让我们通过添加新的字段和覆盖 get-x 、 get-y、 set-x和 set-y方法来定义极坐标点。 关于下面的代码的一些细节：

- 与颜色点一样，我们的“构造函数”使用“超类”构造函数。
- 与Java中一样，我们的极坐标点对象仍然具有x和y字段，但我们从不使用它们。
- 为了简单起见，我们只需通过将替换项放在方法列表中的早期位置来覆盖方法。 被覆盖的方法。 这是因为 assoc在列表中返回第一个匹配的对。

最重要的是，对于极坐标点，distToOrigin“方法”仍然有效，因为其体中的方法调用将使用在make-polar-point的定义中列出的 'get-x和 'get-y的过程，就像动态分派所需的那样。 正确的行为是由我们的 send 函数将整个对象作为第一个参数传递而得到的。

```
(定义 (make-polar-point _r _th)
  (让 ([pt (make-point #f #f)])
    (对象
     (附加 (列表 (mcons 'r _r)
                 (mcons 'theta _th))
           (对象字段 pt))
     (连接
      (列表
       (cons 'set-r-theta
             (lambda (self args)
               (开始
                 (设置 self 'r (获取参数的第一个元素))
                 (设置 self 'theta (获取参数的第二个元素)))))
       (cons 'get-x (lambda (self 参数)
                      (让 ([r (获取 self 'r)]
                           [theta (获取 self 'theta)])
                        (* r (cos theta)))))
       (cons 'get-y (lambda (self 参数)
                      (让 ([r (获取 self 'r)]
                           [theta (获取 self 'theta)])
                        (* r (sin theta)))))
       (cons 'set-x (lambda (self args)
                      (让* ([a       (获取参数的第一个元素)]
                            [b     (发送自身 '获取-y)]
                            [θ (atan (/ b a))]
                            [r     (sqrt (+ (* a a) (* b b)))])
                        (发送自身 '设置-r-θ r θ))))
       (cons '设置-y (lambda (self args)
                      (让* ([b       (获取参数的第一个元素)]
                            [a     (发送自身 '获取-x)]
                            [θ (atan (/ b a))]
                            [r     (sqrt (+ (* a a) (* b b)))])
                        (发送自身 '设置-r-θ r θ)))))
      (对象方法 pt)))))
```

我们可以创建一个极坐标点对象，并像这样发送一些消息：

```
(定义 p3 (制造极坐标点 4 3.1415926535))
(发送 p3 '获取-x) ; 4
(发送 p3 '获取-y) ; 0 (或者略有舍入误差)
(发送 p3 '到原点的距离) ; 4 (或者略有舍入误差)
(发送 p3 '设置-y 3)
(发送 p3 '到原点的距离) ; 5 (或者略有舍入误差)
```

CSE341：程序设计语言 2016年春季
第8单元总结

标准描述：这个总结大致涵盖了课堂和复习部分的内容。 当回顾材料时，以叙述方式阅读材料并将整个单元的材料放在一个文档中可能会有所帮助。 请报告这些笔记中的错误，甚至是打字错误。 这个总结不能完全替代上课、阅读相关代码等。

目录

面向对象编程与函数分解

我们可以使用实现小型表达式语言的操作来比较过程（函数）分解和面向对象分解的经典示例。 在函数式编程中，我们通常将程序分解为执行某些操作的函数。 在面向对象编程中，我们通常将程序分解为为某种数据提供行为的类。

我们展示了这两种方法在完全相反的方式上基本上表达了相同的思想，哪种方式“更好”要么是品味的问题，要么取决于软件在将来如何被改变或扩展。 然后我们考虑了两种方法如何处理多个参数的操作，在许多面向对象的语言中，这需要一种称为双重（多重）分派的技术，以保持面向对象的风格。

基本设置

下面的问题是一个常见的编程模式的典型例子，而且不巧的是，在课程中我们已经考虑过几次。假设我们有：

- 用于算术等小“语言”的表达式
- 表达式的不同变体，如整数值、否定表达式和加法表达式
- 对表达式的不同操作，如对其进行评估、转换为字符串，或确定它们是否包含常数零

这个问题导致一个概念性的矩阵（二维网格），每个变体和操作组合都有一个条目：

	评估	toString	是否有零
整数			
加法			
否定			

无论你使用什么编程语言或者如何解决这个编程问题，你都需要指明网格中每个条目的正确行为。 某些方法或语言可能更容易指定默认值，但你仍然需要为每个条目做出决策。

函数式方法

在函数式语言中，标准的风格是按照以下方式进行操作：

- 为表达式定义一个数据类型，每个变体都有一个构造函数。 （在动态类型的语言中，我们可能不会在程序中给数据类型命名，但我们仍然是按照概念来思考。同样，在没有直接支持构造函数的语言中，我们可能会使用类似列表的东西，但我们仍然是按照定义一种构造每个数据变体的方式来思考。）
- 为每个操作定义一个函数。
- 在每个函数中，为数据的每个变体设置一个分支（例如，通过模式匹配）。 如果有许多变体的默认情况，我们可以使用类似通配符模式的东西来避免列举所有的分支。

请注意，这种方法实际上只是过程分解：将问题分解为与每个操作相对应的过程。

这段ML代码展示了我们示例的方法：请注意，我们在一个地方定义了所有类型的数据，然后通过每一列实现了表中的九个条目，每一列对应一个函数：

异常 BadResult 的字符串

数据类型 exp =
    Int   int的
  负数| 加法
          exp的乘法

fun eval e =
    情况 e of

```
        Int _       => e
      | 取反 e1 => (case eval e1 of
                      Int i => Int (~i)
                    | _ => raise BadResult "非整数的否定")
      | Add(e1,e2) => (case (eval e1, eval e2) of
                      (Int i, Int j) => Int (i+j)
                    | _ => raise BadResult "非整数的加法")

fun toString e =
    case e of
        Int i       => Int.toString i
      | 取反 e1 => "-(" ^ (toString e1) ^ ")"
      | 加(e1,e2) => "(" ^ (toString e1) ^ " + " ^ (toString e2) ^ ")"

fun hasZero e =
    case e of
        Int i       => i=0
      | 取反 e1 => hasZero e1
      | Add(e1,e2) => (hasZero e1) 或者 (hasZero e2)
```

面向对象的方法

在面向对象的语言中，标准的风格是按照以下方式进行：

- 为表达式定义一个类，每个操作都有一个抽象方法。 (在动态类型的语言中，我们可能实际上不会在程序中列出抽象方法，但我们仍然在思考这个概念。同样，在具有鸭子类型的语言中，我们可能实际上不会使用超类，但我们仍然在思考如何定义我们需要支持的操作。)

- 为每个数据变体定义一个子类。
- 在每个子类中，为每个操作定义一个方法。 如果有许多变体的默认值，我们可以在超类中使用一个方法定义，通过继承来避免列举所有的分支。

请注意，这种方法是一种面向数据的分解方法：将问题分解为与每个数据变体对应的类。

这是Ruby代码，为了清晰起见，不同类型的表达式都是Exp类的子类。 在静态类型的语言中，这是必需的，超类必须声明每个Exp子类定义的方法 - 将所有操作都列在一个地方。 请注意，我们通过每一行为表格定义了九个条目的类。

```
类Exp
  # 可以在这里放置默认实现或辅助方法
end
class Int < Exp
  attr_reader :i
  def initialize i
    @i = i
  end
  def eval
    self
```

```
  结束
  def toString
    @i.to_s
  end
  def hasZero
    i==0
  end
end
class Negate < Exp
  attr_reader :e
  def initialize e
    @e = e
  end
  def eval
    Int.new(-e.eval.i) # 如果e.eval没有i方法（不是Int），则报错
  end
  def toString
    "-(" + e.toString + ")"
  end
  def hasZero
    e.hasZero
  end
end
class Add < Exp
  attr_reader :e1, :e2
  def initialize(e1,e2)
    @e1 = e1
    @e2 = e2
  end
  def eval
    Int.new(e1.eval.i + e2.eval.i) # 如果e1.eval或e2.eval没有i方法，则报错
  end
  def toString
    "(" + e1.toString + " + " + e2.toString + ")"
  end
  def hasZero
    e1.hasZero || e2.hasZero
  end
end
```

笑话的结尾

所以我们已经看到，函数分解将程序分解为执行某些操作的函数，而面向对象分解将程序分解为为某种数据提供行为的类。 这两者完全相反，以至于它们是相同的 - 只是决定是按列还是按行布局我们的程序。理解这种对称性在概念化软件或决定如何分解问题时非常有价值。 此外，各种软件工具和集成开发环境可以帮助您以与源代码分解不同的方式查看程序。 例如，一个面向对象编程语言的工具，显示所有覆盖某个超类的方法 `foo`的工具，即使代码是按行组织的，也会以列的方式显示给您。

那么，哪个更好？ 这通常是个人偏好的问题，无论是按行还是按列布局，都取决于个人感觉是否“更自然”。

所以你有权利发表自己的观点。 最常见的观点可能取决于软件的性质。 对于我们的表达问题，功能性方法可能更受欢迎：将求值的情况与操作的情况放在一起更“自然”。 对于实现图形用户界面等问题，面向对象的方法可能更受欢迎：将某种数据类型（如菜单栏）的操作放在一起（如背景颜色、高度和鼠标点击事件），而不是将某种数据类型（如菜单栏、文本框、滑块条等）的情况放在一起。选择还可能取决于所使用的编程语言，库的组织方式等。

通过新操作或变体扩展代码

如果我们稍后通过添加新的数据变体或新的操作来扩展我们的程序，那么在“行”和“列”之间的选择就变得不那么主观了。

考虑一下函数式方法。 添加新操作很容易：我们可以实现一个新函数而不需要编辑任何现有的代码。 例如，这个函数创建一个新的表达式，其结果与其参数相同，但没有负常数：

```
fun noNegConstants e =
    case e of
        Int i        => 如果 i < 0 则 Negate (Int(~i)) 否则 e
      | Negate e1    => 取反(去除负常数 e1)
      | 加(e1,e2) => 加(去除负常数 e1, 去除负常数 e2)
```

另一方面，添加一个新的数据变体，例如Mult of exp * exp则不那么愉快。 我们需要返回并更改所有的函数以添加一个新的情况。 在静态类型的语言中，我们确实得到了一些帮助：在添加 Mult构造函数之后，如果我们的原始代码没有使用通配符模式，那么类型检查器将在我们需要为 Mult添加一个情况的地方给出一个非穷尽的模式匹配警告。

再次，面向对象的方法完全相反。 添加一个新的变体很容易：我们可以实现一个新的子类而不编辑任何现有的代码。 例如，这个Ruby类为我们的语言添加了乘法表达式：

```
class Mult < Exp
  attr_reader :e1, :e2
  def initialize(e1,e2)
    @e1 = e1
    @e2 = e2
  end
  def eval
    Int.new(e1.eval.i * e2.eval.i) # 如果e1.eval或e2.eval没有i方法则报错
  end
  def toString
    "(" + e1.toString + " * " + e2.toString + ")"
  end
  def hasZero
    e1.hasZero || e2.hasZero
  end
end
```

另一方面，添加一个新的操作，比如noNegConstants，就不那么愉快了。我们需要回去

并更改所有的类以添加一个新的方法。 在静态类型的语言中，我们确实得到了一些帮助：在 Expsuperclass 中声明所有子类都应该有一个noNegConstants方法后，类型检查器 将为任何需要实现该方法的类给出错误提示。（这种静态类型是使用抽象方法和抽象类来实现的，这将在后面讨论。）

为可扩展性进行规划

如上所示，函数分解允许添加新操作，面向对象分解允许添加新变体，而无需修改现有代码，也无需明确规划 - 编程风格“就是这样工作的”。如果您提前计划并使用某种笨拙的编程技术（如果经常使用，似乎不那么笨拙），函数分解可以支持新变体，面向对象分解可以支持新操作。

我们不在此处详细考虑这些技术，您也不需要负责学习它们。 对于面向对象编程，“访问者模式”是一种常见方法。 这种模式通常使用双重分派来实现，下面将介绍其他目的的覆盖。 对于函数式编程，我们可以定义我们的数据类型具有“其他”可能性，并且我们的操作接受一个可以处理“其他数据”的函数。以下是在SML中的想法：

```
数据类型'a ext_exp =
    Int    int的
  | 'a ext_exp的否定
  | 加       'a ext_exp * 'a ext_exp的
  | 'a的其他扩展表达式

fun eval_ext (f,e) = (* 注意我们传递一个函数来处理扩展 *)
  case e of
    Int i        => i
  | 否定 e1       => 0 - (eval_ext (f,e1))
  | Add(e1,e2)   => (eval_ext (f,e1)) + (eval_ext (f,e2))
  | OtherExtExp e => f e
```

通过这种方法，我们可以通过实例化 'a为exp * exp来创建一个支持乘法的扩展，通过将函数(fn (x,y) => eval _ext(f,e1) * eval_ext(f,e2))传递给 eval_ext，并使用OtherExtExp(e1,e2)来乘以 e1和 e2。 这种方法可以支持不同的扩展，但不能很好地支持将两个单独创建的扩展组合在一起。

请注意，将原始数据类型包装在新数据类型中是行不通的，就像这样：

```
数据类型myexp_wrong =
         OldExp of exp
       | MyMult of myexp_wrong * myexp_wrong
```

这种方法不允许，例如，一个 Add的子表达式是一个 MyMult.

对可扩展性的思考

很明显，如果你期望有新的操作，那么函数式方法更自然；如果你期望有新的数据变体，那么面向对象的方法更自然。问题是（1）未来往往难以预测；我们可能不知道可能有哪些扩展，以及（2）两种扩展形式都可能发生。 像Scala这样的新语言旨在很好地支持这两种扩展形式；我们仍然在获得实际经验，看它的工作效果如何，因为这是一个根本上困难的问题。

更一般地说，制作既健壮又可扩展的软件是有价值但困难的。 可扩展性使原始代码更难开发，更难在局部推理，并且更难改变。

（不破坏扩展）。 事实上，语言通常提供构造来 *prevent*extensibility。
ML的模块可以隐藏数据类型，这样可以防止在模块外部定义新的操作。
Java的 `final`修饰符可以防止子类。

使用功能分解的二进制方法

到目前为止，我们考虑的操作只使用了一个类型的一个值，该类型具有多个数据变体： `eval`,`toString`, has `Zero`和`noNegConstants`都对一个表达式进行操作。 当我们有需要两个（二进制）或更多（n-ary）变体作为参数的操作时，通常会有更多的情况。 使用功能分解，所有这些情况仍然在一个函数中一起处理。 如下所示，面向对象的方法更加繁琐。

为了举例，假设我们在表达式语言中添加了字符串值和有理数值。
进一步假设我们将 `Add`表达式的含义更改为以下内容：

- 如果参数是整数或有理数，则进行适当的算术运算。
- 如果其中一个参数是字符串，则将另一个参数转换为字符串（除非它已经是一个字符串），并返回字符串的连接。

因此，将 `Negate`或 `Mult`的子表达式评估为 `String`或 `Rational`是错误的，但是 `Add`的子表达式可以是我们语言中的任何类型的值：int、string或rational。

SML代码中有一个有趣的变化是在 `eval`的 `Add`情况下。 现在我们必须考虑9个（即$3 * 3$）子情况，每个子情况对应于通过评估子表达式产生的值的组合。 为了使这一点更明确，并且更像下面考虑的面向对象版本，我们可以将这些情况移到一个辅助函数 `add_values`中，如下所示：

```
fun eval e =
   case e of
      ...
     | Add(e1,e2)   => add_values (eval e1, eval e2)
      ...

fun add_values (v1,v2) =
    case (v1,v2) of
        (Int i,   Int j)          => Int (i+j)
      | (Int i,   String s)       => String(Int.toString i ^ s)
      | (Int i,   Rational(j,k)) => Rational(i*k+j,k)
      | (String s,   Int i)       => String(s ^ Int.toString i) (* 不可交换 *)
      | (String s1, String s2)    => String(s1 ^ s2)
      | (字符串 s，有理数(i,j)) => 字符串(s ^ Int.toString i ^ "/" ^ Int.toString j)
      | (有理数 _,        Int _)     => add_values(v2,v1) (* 可交换：避免重复 *)
      | (Rational(i,j), String s) => String(Int.toString i ^ "/" ^ Int.toString j ^ s)
      | (Rational(a,b), Rational(c,d)) => Rational(a*d+b*c,b*d)
      | _ => raise BadResult "传递给add_values的非值"
```

注意 `add_values`在这个二维网格中定义了所有9个条目，用于在我们的语言中添加值 - 这是一个与我们之前考虑的不同类型的矩阵，因为行和列是变体。

	整数	字符串	有理数
整数			
字符串			
有理数			

虽然案例的数量可能很大，但这是问题的固有特性。 如果许多案例的工作方式相同，我们可以使用通配符模式和/或辅助函数来避免冗余。 冗余的一个常见来源是交换律，即值的顺序不重要。 在上面的例子中，只有一种这样的情况：添加有理数和整数与添加整数和有理数是相同的。注意我们如何利用这种冗余通过调用add_values(v2,v1)使一个案例使用另一个案例。

面向对象编程中的二进制方法：双重分派

现在我们转向以面向对象的方式支持字符串、有理数和增强的Add的评估规则。 因为Ruby有内置的名为String和Rational的类，所以我们将扩展我们的代码，使用名为MyString和MyRational的类，但显然这不是重点。 第一步是添加这些类并让它们实现所有现有的方法，就像我们之前添加Mult时所做的那样。 然后，“只剩下”修改Add类的eval方法，该方法之前假设递归结果将是Int的实例，因此具有一个getter方法i：

```
def eval
  Int.new(e1.eval.i + e2.eval.i) # 如果e1.eval或e2.eval没有i方法则报错
end
```

现在我们可以用类似于我们在ML中的 add_values辅助函数的代码来替换这个方法体，但是这样的辅助函数不符合面向对象的风格。 相反，我们期望 add_values是在表示我们语言中的值的类中的一个方法：一个 Int， MyRational或 MyString应该“知道如何将自己与另一个值相加”。所以在 Add中，我们写道：

```
def eval
  e1.eval.add_values e2.eval
end
```

这是一个很好的开始，现在我们有了在类 Int， MyRational和 MyString中的 add_values方法的义务。通过将 add_values方法放在 Int， MyString和 MyRational类中，我们可以根据对象的类（即 e1.eval返回的接收者）使用动态分派来将我们的工作分成三个部分，即在 Add的 eval方法中的 add_values调用的接收者。 但是然后这三个类中的每一个都需要处理九种情况中的三种，根据第二个参数的类别。 一种方法是，在这些方法中放弃面向对象的风格（！），并使用运行时对类进行测试以包括这三种情况。 Ruby代码将如下所示：

```
类 Int
 ...
 定义 add_values v
    如果 v.is_a? Int
       ...
    否则如果 v.is_a? MyRational
       ...
    否则
```

```
            ...
        结束
    结束
结束
类 MyRational
    ...
    定义 add_values v
        如果 v.is_a? Int
            ...
        否则如果 v.is_a? MyRational
            ...
        否则
            ...
        结束
    结束
结束
类 MyString
    ...
    定义 add_values v
        如果 v.is_a? Int
            ...
        否则如果 v.is_a? MyRational
            ...
        否则
            ...
        结束
    结束
结束
```

虽然这种方法可行，但它并不是真正的面向对象编程。 相反，它是面向对象分解（对第一个参数进行动态调度）和函数分解（使用 is_a?来确定每个方法中的情况）的混合。 这并不一定有什么问题 - 它可能比我们即将展示的更简单易懂 - 但它放弃了面向对象编程的可扩展性优势，而且并不是“完全”的面向对象编程。

下面是如何思考“完全”的面向对象编程方法：我们三个 add_values方法内部的问题是我们需要“知道”参数 v的类。 在面向对象编程中，策略是用 v上的方法调用来替代“需要知道类”的操作。 所以我们应该“告诉 v”执行加法，传递 self。 而且，我们可以“告诉 v” self是什么类，因为 add_values方法知道：在 Int中， self是一个 Int，例如。 我们“告诉 v类”的方式是为每种类型的参数调用不同的方法。

这种技术被称为双重分派。 这是我们示例的代码，后面是额外的解释：

```
类 Int
  ... # 与 add_values 无关的其他方法
  def add_values v # 第一次分派
    v.addInt self
  end
  def addInt v # 第二次分派：v 是 Int
    Int.new(v.i + i)
  end
  def addString v # 第二次分派：v 是 MyString
    MyString.new(v.s + i.to_s)
```

```
  结束
  def addRational v # 第二次分派：v 是 MyRational
    MyRational.new(v.i+v.j*i,v.j)
  end
end
class MyString
  ... # 与 add_values 无关的其他方法
  def add_values v # 第一次分派
    v.addString self
  end
  def addInt v # 第二次分派：v 是 Int
    MyString.new(v.i.to_s + s)
  end
  def addString v # 第二次分派：v 是 MyString
    MyString.new(v.s + s)
  end
  def addRational v # 第二次分派：v 是 MyRational
    MyString.new(v.i.to_s + "/" + v.j.to_s + s)
  end
end
class MyRational
  ... # 与add_values无关的其他方法
  def add_values v # 第一个分派
    v.addRational self
  end
  def addInt v # 第二个分派
    v.addRational self # 重用可交换操作的计算
  end
  def addString v # 第二个分派：v是MyString
    MyString.new(v.s + i.to_s + "/" + j.to_s)
  end
  def addRational v # 第二个分派：v是MyRational
    a,b,c,d = i,j,v.i,v.j
    MyRational.new(a*d+b*c,b*d)
  end
end
```

在理解所有方法调用的工作原理之前，请注意我们现在有了9种不同的加法情况：

- Int中的addInt方法用于当加法的左操作数是Int（在v中）且右操作数是Int（在self中）时。
- Int中的addString方法用于当加法的左操作数是MyString（在v中）且右操作数是Int（在self中）时。
- 在Int中，addRational方法用于当加法的左操作数是MyRational（在v中）且右操作数是Int（在self中）时。
- 在MyString中，addInt方法用于当加法的左操作数是Int（在v中）且右操作数是MyString（在self中）时。
- 在MyString中，addString方法用于当加法的左操作数是MyString（在v中）且右操作数是MyString（在self中）时。

- 在MyString中，addRational方法用于当加法的左操作数是MyRational（在v中）且右操作数是MyString（在self中）时。
- 在MyRational中，addInt方法用于当加法的左操作数是Int（在v中）且右操作数是MyRational（在self中）时。
- 在MyRational中，addString方法用于当加法的左操作数是MyString（在v中）且右操作数是MyRational（在self中）时。
- MyRational中的addRational方法是用于当加法的左操作数是MyRational（在v中）而右操作数是MyRational（在self中）时。

正如我们在面向对象编程中所期望的那样，我们的9个情况与ML代码相比“分散”开来。 现在我们需要了解动态分派是如何在所有9个情况中选择正确的代码的。 从Add中的eval方法开始，我们有e1.eval.add_values e2.eval。 有3个add_values方法，动态分派将根据e1.eval返回的值的类来选择一个。 这是“第一次分派”。假设e1.eval是一个Int。 然后下一个调用将是v.addInt self，其中self是e1.eval，v是e2.eval。 再次感谢动态分派，查找的方法将是9个情况中的正确情况。 这是“第二次分派”。所有其他情况都类似地工作。

理解双重分派可能是一种令人费解的练习，它加强了动态分派的工作原理，这是面向对象编程与其他编程语言的关键区别。 这并不一定直观，但在Ruby/Java中，以面向对象的方式支持二进制操作（如加法）是必须的。

笔记：

- 下一节将讨论具有多方法的面向对象编程语言，不需要手动进行双重分派。
- 像Java这样的静态类型语言不会妨碍双重分派的使用。 事实上，需要声明方法参数和返回类型，并在超类中指示所有子类实现的方法，可以更容易理解正在发生的事情。我们的示例的完整Java实现已发布在课程材料中。 （在Java中，常常为不同类型的参数重用方法名称。因此，我们可以使用add而不是addInt，addString和addRational，但这可能比有帮助更令人困惑，尤其是在初学双重分派时。）

多方法

并非所有面向对象编程语言都需要繁琐的双重调度模式来实现二元操作的完全面向对象风格。 具有多方法的语言，也被称为多重调度，提供了更直观的解决方案。 在这样的语言中，类I*n*t，MyStri*n*g和MyRati*o*nal可以分别定义三个名为add_values的方法（因此程序中将有九个名为add_values的方法）。 每个add_values方法都会指定其参数所期望的类。 然后e1.eval.add_values e2.eval将通过考虑e1.eval的结果的类和e2.eval的结果的类，在运行时选择其中的9个方法之一。

这是一种强大且不同的语义，与我们对面向对象编程的研究不同。在我们对Ruby的研究中（Java/C#/C++的工作方式相同），方法查找规则涉及接收者的运行时类（我们正在调用其方法的对象），而不是参数的运行时类。 多重调度是通过考虑多个对象的类并使用所有这些信息来选择要调用的方法的“更动态的调度”。

Ruby不支持多方法，因为Ruby致力于在每个类中只有一个具有特定名称的方法。 任何对象都可以传递给这个方法。 因此，在同一个类中没有办法有3个 `add_values`方法，并且没有办法根据参数指示使用哪个方法。

Java和C++也没有多方法。 在这些语言中，你可以在一个类中有多个同名方法，并且方法调用的语义确实使用参数的类型来选择调用哪个方法。 但它使用的是参数的类型，这些类型在编译时确定，而不是在运行时确定结果的类。 这种语义被称为静态重载。 它被认为是有用和方便的，但它不是多方法，并且不能避免在我们的示例中需要双重分派。

C#具有与Java和C++相同的静态重载，但是在语言的4.0版本中，可以通过在正确的位置使用类型“dynamic”来实现多方法的效果。 我们在这里不讨论细节，但这是一个将语言特性结合起来实现有用目标的好例子。

许多面向对象编程语言多年来一直有多方法 - 这不是一个新的想法。 也许最著名的具有多方法的现代语言是Clojure。

多重继承

我们已经看到面向对象编程的本质是继承，覆盖和动态dispatch。 我们所有的例子都是具有1个（直接）超类的类。 但是，如果继承如此有用和重要，为什么不允许使用更多在其他地方定义的代码，比如另一个类。 我们现在开始讨论3个相关但不同的想法：

- 多重继承：具有多重继承的语言允许一个类扩展多个其他类。这是最强大的选项，但是会出现一些语义问题，而其他想法避免了这些问题。 Java和Ruby没有多重继承; C++有。
- 混入：Ruby允许一个类有一个直接超类，但任意数量的混入。 因为一个混入只是一堆方法，许多语义问题都消失了。 混入不能解决所有需要多重继承的情况，但它们有一些很好的用途。 特别是，优雅使用混入通常涉及混入方法调用它们假定在包括混入的所有类中定义的方法。 Ruby的标准库很好地利用了这种技术，你的代码也可以。
- *Java/C#*风格的接口：*Java/C#*类只有一个直接的超类，但可以“实现”任意数量的接口。 因为接口不提供行为 - 它们只要求某些方法存在 - 大部分语义问题都消失了。 接口基本上是关于类型检查的，我们将在本单元后面更详细地学习，所以在像Ruby这样的语言中几乎没有使用它们的理由。 *C++*没有接口，因为继承一个具有所有“抽象”方法（或者在*C++*中称为“纯虚”方法）的类可以实现相同的功能，如下面更详细地描述。

要理解为什么多重继承有潜在的用处，考虑两个经典的例子：

- 考虑一个 `Point2D`类，它有子类 `Point3D`（添加一个z维度）和 `ColorPoint`（添加一个颜色属性）。 要创建一个`ColorPoint3D`类，似乎很自然地有两个直接的超类， `Point3D`和 `ColorPoint`，所以我们从两者继承。
- 考虑一个人类，它有子类艺术家和牛仔。 要创建一个艺术家牛仔（既是艺术家又是牛仔），似乎自然而然地有两个直接的超类。 然而，请注意，艺术家类和牛仔类都有一个名为“draw”的方法，但它们的行为非常不同（创建图片与制造枪支）。

在没有多重继承的情况下，你会在这些示例中复制代码。 例如，彩色三维点可以是三维点的子类，并从彩色点复制代码，或者它可以是彩色点的子类，并从三维点复制代码。

如果我们有多重继承，我们必须决定它的含义。 天真地说，新类具有所有超类的方法（在字段也是类定义的一部分的语言中也包括字段）。
然而，如果两个直接超类具有相同的字段或方法，那意味着什么？ 如果字段或方法是从相同的共同祖先继承的，这是否重要？ 在返回我们的示例之前，让我们更详细地解释这些问题。

通过单继承，类层次结构——程序中的所有类以及它们之间的继承关系——形成一个树，其中 A继承 B意味着 A是 B在树中的子类。 通过多继承，类层次结构可能不是一个树。 因此，它可以有“钻石”——四个类，其中一个是两个其他类的（不一定是直接的）子类，它们有一个共同的（不一定是直接的）超类。 通过“直接”我们指的是直接继承（子父关系），而我们可以说“传递”是指更一般的祖先-后代关系。

通过多个超类，我们可能会在从不同类继承的字段/方法上产生冲突。
对于ArtistCowboy对象的 draw方法是一个明显的例子，我们希望在子类中同时拥有这两个方法，或者可能覆盖其中一个或两个。 至少我们需要使用 super来指示所需的超类。 但这不一定是唯一的冲突。

假设Person类有一个pocket字段，艺术家和牛仔使用它来做不同的事情。 那么也许一个ArtistCowboy应该有两个pocket，即使pocket的概念是在共同的祖先Person中创建的。

但是如果你看看我们的ColorPoint3D示例，你会得出相反的结论。 在这里，Point3D和ColorPoint都继承了x和y的概念，但我们肯定不希望ColorPoint3D有两个x方法或两个@x字段。

这些问题是多重继承语言（最著名的是C++）需要相当复杂的规则来处理子类化、方法查找和字段访问的一些原因。 例如，C++有（至少）两种不同的创建子类的形式。 一种总是从所有超类复制所有字段。另一种只复制最初由相同的共同祖先声明的字段的一个副本。 （这种解决方案在Ruby中不起作用，因为实例变量不是类定义的一部分。）

混合

Ruby有混合，它们介于多重继承（见上文）和接口（见下文）之间。
它们为包含它们的类提供实际代码，但它们本身不是类，所以你不能创建它们的实例。 Ruby并不是发明混合的。 它的标准库很好地利用了它们。
混合的近义词是特征，但我们将坚持Ruby所称的混合。

要定义一个Ruby混合，我们使用关键字模块而不是类。 （模块不仅仅用作混合，因此选择了奇怪的词。）
例如，这是一个用于向类添加颜色方法的混合的例子：

```
模块颜色
  属性访问器：颜色
  定义变暗
    self.color = "dark " + self.color
  end
end
```

这个混合定义了三个方法，颜色，颜色=和变暗。 通过使用include关键字和混合的名称，类定义可以包含这些方法。例如：

```
类 ColorPt < Pt
    包括 Color
结束
```

这定义了一个子类 Pt，它还具有 Color定义的三个方法。 这样的类还可以定义/覆盖其他方法；在这里，我们选择不添加任何额外的内容。这不一定是一个好的风格，有几个原因。 首先，我们从 Pt继承的 initia lize方法没有创建 @color字段，所以我们依赖于客户端在调用 color之前调用 color=，否则将返回 nil 。 因此，覆盖initialize可能是一个好主意。 其次，使用实例变量的混入在风格上是有问题的。

正如你在Ruby中所期望的那样，它们使用的实例变量将成为混入所包含的对象的一部分。因此，如果与类（或另一个混入）定义的意图分开的实例变量存在名称冲突，两个独立的代码片段将改变相同的数据。 毕竟，混入是“非常简单”的-它们只是定义了一组可以包含在类中的方法。

现在我们有了混入，我们也必须重新考虑我们的方法查找规则。 我们必须选择一些东西，这就是Ruby选择的方式：如果 obj是类 C的一个实例，并且我们向 obj发送消息 m，

- 首先在类 C中查找 m的定义。
- 接下来查找 C中包含的混入。 后面的混入会覆盖前面的混入。
- 接下来查找 C的超类。
- 接下来查找 C的超类的混入。
- 接下来查找 C的超超类。
- 等等。

许多优雅的混入使用以下听起来奇怪的方法：它们定义调用 self上的其他未由混入定义的方法的方法。 相反，混入假设所有包含混入的类都定义了这个方法。 例如，考虑这个允许我们“加倍”任何定义了 +的类的实例的混入：

```
  模块双倍器
  def double
    self + self # 使用self的+消息，而不是在Doubler中定义
  end
end
```

如果我们在某个类C中包含Doubler并在该类的实例上调用double，我们将调用实例上的+方法，如果未定义则会出错。 但是如果+被定义了，一切都会顺利进行。所以现在我们可以通过定义+并包含Doubler mixin来轻松实现加倍的便利。例如：

```
class AnotherPt
  attr_accessor :x, :y
  include Doubler
  def + other # 添加两个点
    ans = AnotherPt.new
    ans.x = self.x + other.x
```

```
    ans.y = self.y + other.y
    ans
  end
end
```

现在，AnotherPt的实例具有我们想要的double方法。 我们甚至可以将double添加到已经存在的类中：

```
class String
  include Doubler
end
```

当然，这个例子有点傻，因为 double方法非常简单，一遍又一遍地复制它不会那么麻烦。

在Ruby中，使用了两个名为 Enumerable和 Comparable的混入。 Comparable提供了 =, !=, >, >=, <和 <=等方法，所有这些方法都假设类定义了 <=>。 而 <=>需要做的是，如果左边的参数小于右边的参数，则返回一个负数；如果它们相等，则返回0；如果左边的参数大于右边的参数，则返回一个正数。 现在一个类不必定义所有这些比较方法 - 它只需要定义 <=>并包含 Comparable即可。 考虑以下比较姓名的例子：

```
class Name
  attr_accessor :first, :middle, :last
  include Comparable
  def initialize(first,last,middle="")
    @first = first
    @last = last
    @middle = middle
  end
  def <=> other
    l = @last <=> other.last # <=> defined on strings
    return l if l != 0
    f = @first <=> other.first
    return f if f != 0
    @middle <=> other.middle
  end
end
```

在 Comparable中定义方法很容易，但我们肯定不希望为每个想要进行比较的类重复这项工作。 例如， >方法只是：

```
def > other
  (self <=> other) > 0
end
```

许多有用的以块为参数的方法，用于迭代某些数据结构的模块是 Enumerable。例如any?， map， count和 inject。 它们都是在假设类具有方法 each的情况下编写的。 因此，一个类可以定义 each，包含 Enumerable混入，并拥有所有这些方便的方法。 因此，例如， Array类只需定义 each并包含 Enumerable。 这是我们可能定义的一个范围类的另一个示例：[1]

[1]我们不会实际定义这个，因为Ruby已经有非常强大的范围类。

```
class MyRange
  include Enumerable
  def initialize(low,high)
    @low = low
    @high = high
  end
  def each
    i=@low
    while i <= @high
      yield i
      i=i+1
    end
  end
end
```

现在我们可以像这样编写代码MyRange.new(4,8).inject {|x,y| x+y} orMyRange.new(5,12).count {|i| i.odd?}. 请注意， Enumerable中的 map方法总是返回一个 Array的实例。 毕竟，它不知道如何生成任何类的实例，但它知道如何生成一个包含由 each产生的每个元素的数组。 我们可以像这样在 Enumerable混入中定义它：

```
 def map
   arr = []
   each {|x| arr.push x }
   arr
 end
```

混合类不如多重继承强大，因为我们必须事先决定将一个类制作成什么，将一个混合类制作成什么。 给定艺术家和牛仔类，我们仍然没有自然的方法来创建一个艺术家牛仔。 而且不清楚我们可能希望以混合类的方式定义艺术家、牛仔或两者之一。

Java/C#风格的接口

在Java或C#中，一个类只能有一个直接的超类，但可以实现任意数量的接口。
接口只是一系列方法以及每个方法的参数类型和返回类型。 类类型只有在实际上提供了（直接或通过继承）所有它声称实现的接口的所有方法时才进行类型检查。 接口是一种类型，所以如果一个类C实现了接口I，那么我们可以将C的实例传递给期望类型为I的参数的方法，例如。 接口更接近于“鸭子类型”的概念，而不仅仅使用类作为类型（在Java和C#中，每个类也是一种类型），但只有在类定义明确声明实现接口时，类才具有某个接口类型。 我们将在本单元后面更详细地讨论面向对象的类型检查。

因为接口实际上不定义方法 - 它们只是给方法命名并给出类型 - 所以不会出现上述关于多重继承的问题。 如果两个接口有一个方法名冲突，也没关系 - 一个类仍然可以实现它们两个。 如果两个接口在方法的类型上有分歧，那么没有类可能同时实现它们，但类型检查器会捕捉到这个问题。 因为接口不定义方法，所以不能像混入一样使用它们。

在动态类型语言中，实际上没有太多理由使用接口。[2] 我们已经可以传递任何对象并在任何对象上调用任何方法。

[2] 可能唯一的用途是改变Ruby的is a?的含义。以便合并接口，但我们可以更直接地使用反射来查找对象的方法。

将对象传递给任何方法并在任何对象上调用任何方法都是可以的。 我们需要在头脑中（最好是在必要时的注释中）跟踪哪些对象可以响应哪些消息。 动态类型的本质不是写下这些东西。

底线：在静态类型的语言（如Java和C#）中，实现接口不会继承代码，它纯粹与类型检查有关。这使得这些语言中的类型系统更加灵活。 所以Ruby不需要接口。

抽象方法

通常，一个类定义有调用其他方法的方法，这些方法实际上在类中并没有定义。 创建这样一个类的实例并使用方法，以至于出现“方法丢失”错误将是一个错误。 那么为什么要定义这样一个类呢？ 因为这个类的整个目的是被子类化，并且不同的子类以不同的方式定义缺失的方法，依靠动态分派使得超类中的代码调用子类中的代码。 在Ruby中，这个功能完全正常 - 你可以有注释表明某些类只用于子类化的目的。

在静态类型语言中，情况更有趣。 在这些语言中，类型-检查的目的是防止“方法丢失”错误，因此在使用这种技术时，我们需要指示不能创建超类的实例。 在Java/C#中，这些类被称为“抽象类”。我们还需要给出（非抽象）子类必须提供的方法的类型。 这些是“抽象方法”。由于这些语言中的子类型化，我们可以有具有超类类型的表达式，并且知道在运行时对象实际上将是其中一个子类。 此外，类型检查确保对象的类已实现所有抽象方法，因此可以安全地调用这些方法。 在C++中，抽象方法被称为“纯虚方法”，并且具有相同的目的。

抽象方法和高阶函数之间存在有趣的类比。 在这两种情况下，语言支持一种编程模式，其中一些代码以灵活和可重用的方式传递给其他代码。在面向对象编程中，不同的子类可以以不同的方式实现抽象方法，并且通过动态分派，超类中的代码可以使用这些不同的实现。 对于高阶函数，如果一个函数将另一个函数作为参数，不同的调用者可以提供不同的实现，然后在函数体中使用这些实现。

具有抽象方法和多重继承的语言（例如，C++）不需要接口。 相反，我们可以只使用仅包含抽象（纯虚）方法的类，就像它们是接口一样，并且让类实现这些“接口”只需对这些类进行子类化。 这种子类化并不继承任何代码，因为抽象方法不定义方法。 通过多重继承，我们不会“浪费”我们的一个超类来实现这种模式。

子类型化简介

我们之前研究了函数式程序的静态类型，特别是ML的类型系统。 ML使用其类型系统来防止错误，例如将数字视为函数。 ML的类型系统的一个关键特点（不会拒绝太多无错误的程序，程序员可能会编写的程序）是参数多态，也称为泛型。

因此，我们还应该研究面向对象程序的静态类型，例如Java中的类型。如果一切都是对象（在Java中不如在Ruby中那样真实），那么我们的类型系统主要应该防止“方法丢失”错误，即向没有该消息的对象发送消息。 如果对象具有从对象外部访问的字段（例如，在Java中），那么我们还希望防止

“字段缺失”错误。 还有其他可能的错误，比如用错误数量的参数调用方法。

虽然像Java和C#这样的语言现在有泛型，但面向对象风格最基本的类型系统表达能力的源头是子类型多态，也被称为子类型。 ML没有子类型，尽管这个决定实际上是语言设计的一个方面（例如，它会使类型推断变得复杂）。

使用Java来研究子类型是很自然的，因为它是一种众所周知的面向对象语言，具有具有子类型的类型系统。 但它也相当复杂，使用类和接口来描述具有方法、重写、静态重载等特性的类型。虽然这些特性有优点和缺点，但它们可能会使任何语言中的子类型的基本思想变得复杂。

因此，虽然我们会简要讨论面向对象编程中的子类型，但我们主要会使用一个具有记录（类似于ML中的具有命名字段的内容的东西 - 基本上是具有公共字段、没有方法和没有类名的对象）和函数（类似于ML或Racket中的函数）的小型语言。这将让我们看到子类型应该如何工作 - 以及不应该如何工作。

这种方法的缺点是我们不能使用我们所学的任何语言：ML没有子类型和记录字段是不可变的，Racket和Ruby是动态类型的，而Java对于我们的起点来说太复杂了。 因此，我们将创造一种只有记录、函数、变量、数字、字符串等的语言，并随着学习的进行解释表达式和类型的含义。

一个虚构的记录语言

为了研究子类型背后的基本思想，我们将使用具有可变字段的记录，以及函数和其他表达式。 我们的语法将是ML和Java的混合，以保持示例简短和清晰。
对于记录，我们将有以下表达式来创建记录、获取字段和设置字段：

- 在表达式`{f1=e1, f2=e2, ..., fn=en}`中，每个 `fi`是一个字段名，每个 ei是一个表达式。 语义是将每个 ei评估为值 vi，结果是记录值`{f1=v1, f2=v2, ..., fn=vn}`。 因此，记录值只是一组字段，每个字段都有一个名称和内容。

- 对于表达式 `e.f`，我们将 `e`评估为一个值 `v`。 如果 `v`是一个具有 `f`字段的记录，则结果是 `f`字段的内容。 我们的类型系统将确保 `v`具有一个 `f`字段。

- 对于表达式`e1.f = e2`，我们将 `e1`和 `e2`评估为值 `v1`和 `v2`。 如果 `v1`是一个具有`f`字段的记录，则将 `f`字段的内容更新为 `v2`。 我们的类型系统将确保 `v1`具有一个`f`字段。 与Java类似，我们选择将`e1.f = e2`的结果设为 `v2`，尽管通常我们不使用字段更新的结果。

现在我们需要一个类型系统，其中包含记录类型和每个表达式的类型规则。
就像在ML中，我们将记录类型写为`{f1:t1, f2:t2, ..., fn:tn}`。 例如，{x：实数, y：实数}将描述具有两个名为 x和 y的字段，其内容为实数的记录。 而{foo: {x：实数, y：实数}, bar：字符串, baz：字符串}将描述具有三个字段的记录其中 `foo`字段包含一个类型为{x：实数, y：实数}的（嵌套的）记录。 然后我们按照以下方式对表达式进行类型检查：

- 如果 `e1`具有类型 `t1`， `e2`具有类型 `t2`，...， `en`具有类型 `tn`，则`{f1=e1, f2=e2, ..., fn=en}`具有类型`{f1:t1, f2:t2, ..., fn:tn}`。

- 如果 e具有包含f : t的记录类型，则 e.f具有类型 t（否则 e.f无法通过类型检查）。
- 如果e1具有包含f : t的记录类型，并且e2具有类型t，则e1.f = e2具有类型t（否则e1.f = e2不会通过类型检查）.

假设其他表达式（如变量、函数、算术和函数调用）的“常规”类型规则，像这样的示例将按预期进行类型检查：

```
fun distToOrigin (p:{x:real,y:real}) =
  Math.sqrt(p.x*p.x + p.y*p.y)

val pythag : {x:real,y:real} = {x=3.0, y=4.0}
val five : real = distToOrigin(pythag)
```

特别地，函数distToOriginhas type{x : real, y : real} -> real，我们使用与ML相同的语法来表示函数类型。调用distToOrigin(pythag)传递了正确类型的参数，因此调用通过了类型检查，调用表达式的结果是返回类型 real。

这种类型系统的作用是：当评估时，不会尝试在没有该字段的记录中查找字段。

子类型化的需求

根据我们目前的类型规则，这个程序无法通过类型检查：

```
fun distToOrigin (p:{x:real,y:real}) =
  Math.sqrt(p.x*p.x + p.y*p.y)

val c : {x:real,y:real,color:string} = {x=3.0, y=4.0, color="green"}
val five : real = distToOrigin(c)
```

在调用distToOrigin(c)时，参数的类型是{x:real,y:real,color:string}，函数期望的类型是{x:real, y:real}，违反了函数必须使用期望的参数类型的类型规则。 然而，上面的程序是安全的：运行它不会导致访问不存在的字段。

一个自然的想法是使我们的类型系统更宽松，如下所示：如果某个表达式具有记录类型{f1:t1, . . . , fn:tn }，则该表达式也可以具有删除一些字段的类型。 然后，我们的示例将通过类型检查：由于表达式 c具有类型{x:real,y:real,color:string}，它也可以具有类型{x:real,y:real}，这使得调用可以通过类型检查。 请注意，我们还可以将 c用作类型为{color:string}->int的函数的参数，例如。

让一个具有一种类型的表达式也具有另一种信息较少的类型是子类型的概念。 （可能看起来有些反常，子类型具有更多的信息，但这就是它的工作原理。更不反常的思考方式是，子类型的值比超类型的值“更少”，因为子类型的值有更多的义务，例如具有更多的字段。）

子类型关系

我们现在将子类型添加到我们的虚构语言中，这样就不需要改变我们的任何现有类型规则。 例如，我们将保持函数调用规则不变，仍然要求实际参数的类型

等于函数定义中的函数参数的类型。 为了做到这一点，我们将在我们的类型系统中添加两个东西：

- 一个类型是另一个类型的子类型的概念：我们将写作`t1 <: t2`表示 `t1`是`t2`的子类型。
- 唯一的新类型规则：如果 `e`的类型是 `t1`并且`t1 <: t2`，则 `e`（也）具有类型 `t2`。

现在我们只需要给出`t1 <: t2`的规则，即一个类型何时是另一个类型的子类型。 这种方法是良好的语言工程 - 我们将子类型的概念分离成一个单一的二元关系，可以与类型系统的其余部分分开定义。

一个常见的误解是，如果我们正在定义自己的语言，那么我们可以随意制定类型和子类型规则。 只有当我们忘记我们的类型系统据称在程序运行时防止某些事情发生时，这才是真的。如果我们的目标仍然是防止字段缺失错误，那么我们不能添加任何导致我们停止实现目标的子类型规则。这就是人们说的“子类型不是一种观点”的意思。

对于子类型化，关键的指导原则是可替代性：如果我们允许`t1 <: t2`，那么任何类型为 `t1`的值必须能够以 `t2`的任何方式使用。对于记录类型，这意味着 `t1`应该具有与`t2`相同的所有字段，并且字段类型也相同。

一些好的子类型化规则

不再废话，我们现在可以给出四个子类型化规则，将其添加到我们的语言中，以接受更多的程序而不破坏类型系统。 前两个规则是针对记录类型的，而后两个规则，虽然可能看起来是多余的，但并不会造成任何伤害，并且在任何具有子类型化的语言中都很常见，因为它们与其他规则结合得很好：

- “宽度”子类型化：超类型可以具有具有相同类型的子集字段，即子类型可以具有“额外”字段
- “排列”子类型化：超类型可以具有相同类型的相同字段集，但顺序可以不同。

- 传递性：如果`t1 <: t2`并且`t2 <: t3`，则`t1 <: t3`。
- 自反性：每个类型都是其自身的子类型：`t <: t`。

请注意，宽度子类型化允许我们忽略字段，排列子类型化允许我们重新排序字段（例如，我们可以将`{x:real,y:real}`作为`{y:real,x:real}`的替代）并且具有这些规则的传递性允许我们同时进行这两个操作（例如，我们可以将`{x:real,foo:string,y:real}`作为`{y:real,x:real}`的替代）。

深度子类型化：一个带有变异的坏主意

到目前为止，我们的子类型化规则允许我们删除字段或重新排序字段，但是超类型无法拥有与子类型中的字段类型不同的字段。 例如，考虑这个例子，它将一个“球体”传递给一个期望“圆形”的函数。请注意，圆和球体都有一个 `center`字段，它本身保存了一个记录。

```
fun circleY (c:{center:{x:real,y:real}, r:real}) =
   c.center.y
```

```
val sphere:{center:{x:real,y:real,z:real}, r:real}) = {center={x=3.0,y=4.0,z=0.0}, r=1.0}
val _ = circleY(sphere)
```

circleY的类型是{center:{x:real,y:real}, r:real}->real而sphere的类型是{center:{x:real,y:real,z:real}, r:real}，所以调用circleY(sphere)只有在类型检查通过时才能进行

```
{center:{x:real,y:real,z:real}, r:real} <: {center:{x:real,y:real}, r:real}
```

到目前为止，这种子类型关系不成立：我们可以删除 center字段，删除 r字段，或重新排序这些字段，但我们不能“进入字段类型进行子类型检查。”

由于我们可能希望上面的程序进行类型检查，因为评估它不会出错，也许我们应该添加另一条子类型规则来处理这种情况。 记录的自然规则是“深度”子类型：

- “深度”子类型：如果ta <: tb，则{f1:t1,...,f:ta,...,fn:tn} <: {f1:t1,...,f:tb,...,fn:tn}。

这个规则允许我们在字段 center上使用宽度子类型来展示

```
{center:{x:real,y:real,z:real}, r:real} <: {center:{x:real,y:real}, r:real}
```

所以上面的程序现在可以进行类型检查。

不幸的是，这个规则破坏了我们的类型系统，允许我们不想允许的程序进行类型检查！这可能不直观，程序员经常犯这种错误 - 认为深度子类型应该被允许。下面是一个例子：

```
fun setToOrigin (c:{center:{x:real,y:real}, r:real})=
   c.center = {x=0.0, y=0.0}

val sphere:{center:{x:real,y:real,z:real}, r:real}) = {center={x=3.0,y=4.0,z=0.0}, r=1.0}
val _ = setToOrigin(sphere)
val _ = sphere.center.z
```

这个程序的类型检查方式与之前类似：调用setToOrigin(sphere)时，参数的类型为{center:{x:real,y:real,z:real}, r:real}并将其作为{center:{x:real,y:real}, r:real}使用。 但是，当我们运行这个程序时会发生什么？ setToOrigin会改变其参数，使 center字段保存一个没有z字段的记录！因此，最后一行的sphere.center.z将无法工作：它试图读取一个不存在的字段。

这个故事的寓意很简单，但经常被忽视：在具有记录（或对象）的语言中，具有字段的getter和setter，深度子类型是不安全的-你不能在子类型和超类型中为字段拥有不同的类型。

然而，请注意，如果一个字段是不可设置的（即，它是不可变的），那么深度子类型规则是正确的并且，就像我们在circleY中看到的那样，是有用的。 所以这又是一个例子，说明没有变异会使编程更容易。 在这种情况下，它允许更多的子类型，这使我们能够更多地重用代码。

另一种看待这个问题的方式是，给定（1）设置一个字段，（2）让深度子类型改变一个字段的类型，以及（3）让类型系统实际上防止字段缺失错误，你可以拥有其中的任意两个。

Java/C#数组子类型的问题

现在我们知道如果记录字段是可变的，深度子类型是不正确的，我们可以质疑Java和C#如何处理数组的子类型。 就子类型而言，数组与记录非常相似，只是字段名是数字，而且所有字段都具有相同的类型。 （由于e1[e2]计算要访问的索引，类型系统不限制结果可能是什么索引，所以我们需要所有字段具有相同的类型，以便类型系统知道结果的类型。）所以这段代码在Java中通过类型检查应该非常令人惊讶：

```
类 Point { ... } // 有字段 double x, y
类 ColorPoint 扩展 Point { ... } // 添加字段 String color
...
void m1(Point[] pt_arr) {
  pt_arr[0] = new Point(3,4);
}
String m2(int x) {
  ColorPoint[] cpt_arr = new ColorPoint[x];
  for(int i=0; i < x; i++)
     cpt_arr[i] = new ColorPoint(0,0,"green");
  m1(cpt_arr);
  return cpt_arr[0].color;
}
```

调用m1(cpt_arr)使用子类型化ColorPoint[] <: Point[]，这本质上是深度子类型化
即使数组索引是可变的。 因此，看起来cpt_arr[0].color将读取一个没有该字段的对象的 color
字段。

在Java和C#中实际发生的是赋值pt_arr[0] = new Point(3,4);如果 pt_arr实际上是一个 ColorPoint数组，将引发一个异常。 在Java中，这是一个ArrayStoreException。 存储引发异常的优点是不需要运行时检查其他表达式，如数组读取或对象字段读取。 不变量是类型为ColorPoint[]的对象始终包含类型为 ColorPoint或子类型的对象，而不是像 Point这样的超类型。 由于Java允许在数组上进行深度子类型化，因此无法在静态上保持此不变量。 相反，它在所有数组赋值上进行运行时检查，使用数组元素的“实际”类型和被赋值的“实际”类。 因此，尽管在类型系统中 pt_arr[0]和new Point(3,4)都具有类型 Point，但此赋值可能在运行时失败。

通常情况下，运行时检查意味着类型系统可以防止更少的错误，需要更多的注意和测试，以及在数组更新时执行这些检查的运行时成本。 那么为什么Java和C#被设计成这样呢？ 在这些语言没有泛型之前，这似乎对于灵活性很重要，这样，例如，如果你编写了一个对 Point对象数组进行排序的方法，你可以使用你的方法对 ColorPoint对象数组进行排序。 允许这样做可以使类型系统更简单，更少地“干扰”你，但代价是静态检查更少。 更好的解决方案是在子类型化的情况下结合使用泛型（请参见下一讲的有界多态）或者支持指示一个方法不会更新数组元素，在这种情况下，深度子类型化是正确的。

在Java/C#中的null

当我们谈到Java/C#选择动态检查而不是“自然”类型规则的地方时，更普遍的问题是如何处理常量 null。由于这个值没有字段或方法（实际上，与Ruby中的 nil不同，它甚至不是一个对象），它的类型应该自然地反映出它不能用作方法的接收者或用于获取/设置字段。 相反，Java和C#允许 null具有任何对象类型，就好像它定义了每个方法并且具有每个字段。从静态的角度来看

从检查的角度来看，这完全是相反的。 结果，语言定义必须指示每个字段访问和方法调用都包含一个运行时检查null，导致Java程序员经常遇到的NullPointerException错误。

那么为什么Java和C#要设计成这样呢？ 因为有些情况下拥有null非常方便，比如在创建Foo实例之前初始化一个类型为Foo的字段（例如，如果你正在构建一个循环列表）。 但是，拥有永远不应该为空的字段和变量也是非常常见的，你希望类型检查器能够帮助你维护这个不变量。 许多关于将“不能为null”类型纳入编程语言的提案已经提出，但是在Java或C#中还没有“流行起来”。相比之下，注意ML如何使用option类型来实现类似的目的：类型t option和t不是同一种类型；你必须使用NONE和SOME构造函数来构建一个数据类型，其中的值可能有也可能没有一个t值。

函数子类型化

当一个函数类型是另一个函数类型的子类型时的规则比记录的深度子类型化问题更加不直观，但对于理解如何安全地覆盖面向对象语言中的方法同样重要（见下文）。

当我们谈论函数子类型化时，我们是指在另一个类型的函数的位置上使用一个类型的函数。 例如，如果 f 接受一个类型为 t1->t2的函数 g，我们能否传递一个类型为t3->t4的函数呢？ 如果 t3->t4是 t1->t2的子类型，则允许这样做，因为通常我们可以传递一个预期类型的子类型给函数 f作为参数。 但这不是关于 f 的“函数子类型化”，而是关于函数参数的“常规”子类型化。 “函数子类型化”是决定一个函数类型是否是另一个函数类型的子类型。

为了理解函数子类型化，让我们使用这个高阶函数的例子，它计算二维点 p和调用 f与 p的结果之间的距离：

```
fun distMoved (f : {x:real,y:real}->{x:real,y:real},
               p : {x:real,y:real}) =
   let val p2 : {x:real,y:real} = f p
       val dx : real = p2.x - p.x
       val dy : real = p2.y - p.y
   in Math.sqrt(dx*dx + dy*dy) end
```

distMoved的类型是

```
(({x:实数,y:实数}->{x:实数,y:实数}) * {x:实数,y:实数}) -> 实数
```

因此，不需要子类型化的 distMoved的调用可能如下所示：

```
fun flip p = {x=~p.x, y=~p.y}
val d = distMoved(flip, {x=3.0, y=4.0})
```

上面的调用也可以传入一个带有额外字段的记录，例如{x=3.0,y=4.0,color="green"}，但这只是对distMoved函数的第二个参数进行普通的宽度子类型化。 我们在这里的兴趣是决定哪些具有类型不同于{x:real,y:real}->{x:real,y:real}的函数可以作为distMoved函数的第一个参数传递。

首先，传入一个返回类型“承诺”更多的函数是安全的，即返回类型是所需返回类型的子类型{x:real,y:real}。 例如，这个调用是可以通过类型检查的：

```
fun flipGreen p = {x=~p.x, y=~p.y, color="green"}
val d = distMoved(flipGreen, {x=3.0, y=4.0})
```

flipGreen的类型是

{x:实数,y:实数} -> {x:实数,y:实数,颜色:字符串}

这是安全的，因为 distMoved期望一个{x:实数,y:实数}->{x:实数,y:实数}，而 flipGreen可以替代这种类型的值，因为 flipGreen返回一个也有颜色字段的记录不是一个问题。

一般来说，这里的规则是，如果ta <: tb，则t -> ta <: t -> tb，即，子类型可以有一个返回类型是超类型返回类型的子类型。 为了引入一点术语，我们说返回类型对于函数子类型化是协变的，这意味着返回类型的子类型化工作方式与整体类型的子类型化工作方式“相同”。

现在让我们考虑传入一个具有不同参数类型的函数。 事实证明，参数类型对于函数子类型化来说是不协变的。 考虑一下对 distMoved的这个例子调用：

```
fun flipIfGreen p = if p.color = "green"
                    then {x=~p.x, y=~p.y}
                    else {x=p.x, y=p.y}
val d = distMoved(flipIfGreen, {x=3.0, y=4.0})
```

flipIfGreen的类型是

{x:实数,y:实数,color:字符串} -> {x:实数,y:实数}

这个程序不应该通过类型检查：如果我们运行它，表达式 p.color将会出现“没有这个字段”的错误，因为传递给flipIfGreen的点没有一个 color字段。 简而言之，ta <: tb，并不意味着ta -> t <: tb -> t。 这将导致在一个“需要更多参数”的函数中使用一个“需要更少参数”的函数。这会破坏类型系统，因为类型规则不会要求提供“更多的东西”。

但事实证明，在需要更少参数的函数中使用“需要更多参数”的函数也可以正常工作。考虑以下对 distMoved的使用示例：

```
fun flipX_Y0 p = {x=~p.x, y=0.0}
val d = distMoved(flipX_Y0, {x=3.0, y=4.0})
```

flipX_Y0的类型是

{x:实数} -> {x:实数,y:实数}

因为 flipX_Y0的参数只需要 x字段。 而对 distMoved的调用没有问题：
distMoved总是使用一个具有 x字段和 y字段的记录来调用其 f参数，这比
flipX_Y0所需的要多。

一般来说，函数子类型的参数类型处理是“反向”的，如下所示：如果tb <: ta，则ta -> t <: tb -> t。 “反向”的技术术语是逆变，意味着参数类型的子类型化与整体类型的子类型化相反。

作为一个最后的例子，函数子类型可以允许参数的逆变和结果的协变：

```
fun flipXMakeGreen p = {x=~p.x, y=0.0, color="绿色"}
val d = distMoved(flipXMakeGreen, {x=3.0, y=4.0})
```

这里flipXMakeGreen的类型是

```
{x:实数} -> {x:实数,y:实数,color:字符串}
```

这是一个子类型

```
{x:实数,y:实数} -> {x:实数,y:实数}
```

因为{x:实数,y:实数} <: {x:实数}(参数的逆变) 和
{x:实数,y:实数,color:字符串} <: {x:实数,y:实数}(结果的协变)。

函数子类型的一般规则是：如果t3 <: t1并且t2 <: t4，那么t1->t2 <: t3->t4。 这个规则，结合自反性（每个类型都是自身的子类型），让我们可以使用逆变的参数，协变的结果，或者两者都使用。

参数逆变是课程中最不直观的概念，但它值得铭记在心，以免忘记。许多非常聪明的人会因为它与方法/函数的调用无关而感到困惑。 相反，它与方法/函数本身有关。 对于将非函数参数传递给函数，我们不需要函数子类型化：我们可以使用其他子类型化规则（例如，记录类型的规则）。 函数子类型化用于高阶函数或将函数本身存储在记录中。 对象类型与具有函数（方法）的记录相关联。

面向对象编程的子类型化

正如承诺的那样，我们现在可以将对子类型化的理解应用到像Java或C#这样的面向对象编程语言中。

对象基本上是一个包含字段（我们在这里假设是可变的）和方法的记录。 我们假设方法的“插槽”是不可变的：如果一个对象的方法 m是用一些代码实现的，那么没有办法将 m变异为引用不同的代码。 （子类的实例可以具有不同的 m代码，但这与变异记录字段不同。）

从这个角度来看，对象的声音子类型化是从记录和函数的声音子类型化中得出的：

- 子类型可以有额外的字段。
- 因为字段是可变的，子类型不能有不同类型的字段。
- 子类型可以有额外的方法。
- 因为方法是不可变的，子类型可以有一个方法的子类型，这意味着子类型中的方法可以具有逆变参数类型和协变结果类型。

话虽如此，Java和C#中的对象类型并不像记录类型和函数类型那样。 例如，我们无法写出一个看起来像这样的类型：

```
{字段：x：实数，y：实数，...
 方法：distToOrigin：（）->实数，...}
```

相反，我们将类名作为类型的重用，如果有一个类 Foo，那么类型 Foo中包含类定义所暗示的所有字段和方法（包括超类）。 正如之前讨论的，我们还有接口，它们更像记录类型，只是不包括字段，并且我们使用接口的名称作为类型。 Java和C#中的子类型化仅包括通过子类关系和类明确指示实现的接口（包括超类实现的接口）所声明的子类型化。

总的来说，这种方法比子类型要求更加严格，但由于它不允许任何东西，它可以可靠地防止“字段丢失”和“方法丢失”错误。特别是：

- 子类可以添加字段，但不能删除它们
- 子类可以添加方法，但不能删除它们
- 子类可以用协变返回类型覆盖方法
- 类可以实现比接口要求更多的方法，或者用协变返回类型实现所需的方法

类和类型是不同的东西！Java和C#故意混淆它们是为了方便起见，但你应该将这些概念分开。 类定义了对象的行为。 子类继承行为，通过扩展和覆盖修改行为。 类型描述了一个对象具有哪些字段和可以响应哪些消息。子类型是一个可替代性的问题，我们想要将其标记为类型错误。 所以，尽量避免说“在超类型中覆盖方法”或“使用子类型将超类的参数传递”。尽管如此，在每个类声明引入一个具有相同名称的类和类型的语言中，这种混淆是可以理解的。

协变自身/这个

作为一个最后微妙的细节和高级点，从类型检查的角度来看，Java的 this（即Ruby的 self）被特殊对待。 在类型检查类 C时，我们知道 this将具有类型 C或子类型，因此假设它具有类型 C是合理的。 在子类型中，例如在覆盖 C中的方法的方法中，我们可以假设this具有子类型。 这些都不会引起任何问题，对于面向对象编程来说是必不可少的。例如，在下面的类B中，只有 this具有类型 B，而不仅仅是 A，方法 m才能进行类型检查。

```
类 A {
  int m(){ return 0; }
}
类 B 扩展 A {
  int x;
  int m(){ return x; }
}
```

但是，如果你回忆起我们在Racket中手动编码对象的方式，编码将 this作为额外的显式参数传递给方法。这将暗示逆变子类型化，意味着子类中的 this不能具有子类型，而在上面的示例中它需要具有子类型。

原来这在某种意义上是特殊的，因为它像是一个额外的参数，但它是一个协变的参数。这怎么可能呢？ 因为它不是一个“正常”的参数，调用者可以选择“任何”正确类型的参数。 方法总是使用一个 this参数来调用，该参数是方法期望的类型的子类型。

这就是为什么在静态类型的语言中手动编写动态分派的原因，即使它们具有子类型：你的类型系统需要特殊的支持来处理 this。

泛型与子类型

我们现在已经学习了子类型多态性，也称为子类型，和参数多态性，也称为泛型类型，或者只是泛型。 所以让我们比较和对比这两种方法，展示每种方法的设计目的。

泛型有什么好处?

有许多使用泛型类型的编程习惯。 我们在这里不考虑所有这些，但让我们重新考虑一下在研究高阶函数时出现的可能最常见的两种习惯用法。

首先，有一些函数可以组合其他函数，比如 compose：

```
val compose: ('b->'c) * ('a->'b) -> ('a->'c)
```

其次，有一些函数可以操作集合/容器，不同的集合/容器可以保存不同类型的值：

```
val length: 'a列表-> int
val map: ('a->'b) ->'a列表->'b列表
val swap: ('a *'b) -> ('b *'a)
```

在所有这些情况下，关键点是，如果我们必须为这些函数选择非通用类型，我们将会得到更少的代码重用。 例如，我们需要一个 swap函数来从一个bool * int生成一个int * bool，并且需要另一个 swap函数来交换一个int * int的位置。通用类型比仅仅说某个参数可以“是任何东西”更有用和准确。例如， swap的类型表明结果的第二个组件与参数的第一个组件具有相同的类型，结果的第一个组件与参数的第二个组件具有相同的类型。 一般来说，我们重用一个类型变量来指示多个东西可以具有任何类型，但必须是相同的类型。

Java中的泛型

自从1990年代创建以来，Java就具有子类型多态性，并且自2004年以来就具有参数多态性。 在Java中使用泛型可能会更加麻烦，因为它没有ML对类型推断和闭包的支持，但是泛型仍然对于相同的编程习惯非常有用。 这里，例如，是一个通用的Pair类，允许两个字段具有任何类型：

```
类Pair <T1, T2> {
  T1 x;
  T2 y;
  Pair(T1 _x, T2 _y){ x = _x; y = _y; }
  Pair<T2,T1> swap() {
     return new Pair<T2,T1>(y,x);
  }
  ...
}
```

注意，类似于ML，“Pair”不是一个类型：类似Pair<String,Integer>是一个类型。 swap方法是面向对象风格的Pair<T1,T2>中的实例方法，返回一个Pair<T2,T1>。我们也可以定义一个静态方法：

```
  static <T1,T2> Pair<T2,T1> swap(Pair<T1,T2> p) {
     return new Pair<T2,T1>(p.y,p.x);
  }
```

出于向后兼容性的原因，前面的段落并不完全正确：Java还有一种类型为Pair的类型，它“忘记”了其字段的类型。对这个“原始”类型进行强制转换会导致编译时警告，你应该明智地遵守：忽视它们可能会导致你意想不到的运行时错误。

子类型是泛型的不良替代品

如果一种语言没有泛型或者程序员对泛型不熟悉，通常会看到以子类型为基础编写的通用代码。 这样做就像用锤子而不是画笔来绘画：
从技术上讲是可能的，但显然是错误的工具。 考虑一下这个Java示例：

```
class LamePair {
  Object x;
  Object y;
  LamePair(Object _x, Object _y){ x=_x; y=_y; }
  LamePair swap() { return new LamePair(y,x); }
  ...
}

String s = (String)(new LamePair("hi",4).y); // 只在运行时捕获错误
```

LamePair类型中的代码可以正常进行类型检查：字段 x和 y的类型为 Object，它是每个类和接口的超类型。 当客户端使用这个类时会出现困难。 将参数传递给构造函数时，子类型的行为符合预期。但是当我们检索字段的内容时，得到一个 Object并不是很有用：我们希望得到我们放回去的值的类型。

子类型化并不是这样工作的：类型系统只知道该字段保存了一个 Object。 因此，我们必须使用一个向下转型，例如 (String)e，这是一个运行时检查，用于验证评估结果 e的类型实际上是 String，或者一般来说，是其子类型。 这样的运行时检查在性能方面具有通常的动态检查成本，但更重要的是，在失败的可能性方面：这不是静态检查的。 实际上，在上面的示例中，向下转型将失败： x字段保存的是一个 String，而不是 y字段。

一般来说，当你使用 Object和向下转型时，你实际上是采用了动态类型的方法：任何对象都可以存储在 Object字段中，所以程序员在没有类型系统的帮助下，需要清楚地知道哪种数据在哪里。

子类型有什么好处?

我们并不是说子类型没有用处：它非常适合允许代码与具有“额外信息”的数据重用。例如，操作点的几何代码应该对彩色点也能正常工作。 在这种情况下，ML代码无法通过类型检查，确实是非常不方便的：

```
fun distToOrigin1 {x=x,y=y} =
```

[3] Java会自动将4转换为一个 Integer对象，其中包含一个4。

```
   Math.sqrt (x*x + y*y)

  (* 无法通过类型检查 *)
(* val five = distToOrigin1 {x=3.0,y=4.0,color="red"} *)
```

一个普遍认可的例子是图形用户界面，其中子类型化效果很好。 大部分图形库的代码对于任何类型的图形元素（“在屏幕上绘制”，“更改背景颜色”，“报告鼠标是否点击它”等）都能正常工作，其中不同的元素如按钮、滑块或文本框可以作为子类型。

泛型是子类型化的不良替代品

在使用泛型而不是子类型化的语言中，你可以使用高阶函数编写自己的代码重用，但对于一个简单的想法来说，这可能会带来很多麻烦。 例如，下面的distToOrigin2使用由调用者传入的getter函数来访问x和y字段，然后下面的两个函数具有不同的类型但相同的主体，只是为了满足类型检查器。

```
fun distToOrigin2(getx,gety,v) =
    let
        val x = getx v
        val y = gety v
    in
        Math.sqrt (x*x + y*y)
    end

fun distToOriginPt (p : {x:real,y:real}) =
    distToOrigin2(fn v => #x v,
                  fn v => #y v,
                  p)

fun distToOriginColorPt (p : {x:real,y:real,color:string}) =
    distToOrigin2(fn v => #x v,
                  fn v => #y v,
                  p)
```

然而，没有子类型，有时候写代码像distToOrigin2如果你想要它
更可重用。

有界多态

正如Java和C#所展示的，静态类型的编程语言也可以有泛型和子类型。 同时拥有这两个特性会有一些复杂性（例如，静态重载和子类型更难定义），但也有好处。 除了支持各自特性的明显好处外，我们还可以结合这些思想来获得更多的代码重用和表达能力。

关键思想是具有有界泛型，其中不仅仅是说“是 T的子类型”或“对于所有类型'a”，我们可以说，“对于所有类型'a，它们是 T的子类型。” 就像泛型一样，我们可以使用'a多次来表示两个事物必须具有相同的类型。 就像子类型化一样，我们可以将'a视为 T的子类型，访问我们所知道的 T具有的任何字段和方法。

我们将使用Java来展示一个例子，希望您只需知道 List<Foo>是表示类型为 Foo的元素列表的语法。

考虑这个 Point类，它有一个 distance方法：

```
类Pt {
    双x，y;
    double distance(Pt pt) { return Math.sqrt((x-pt.x)*(x-pt.x)+(y-pt.y)*(y-pt.y));    }
    Pt(double _x, double _y) { x = _x; y = _y; }
}
```

现在考虑这个静态方法，它接受一个点列表 pts，一个点 center和一个半径 radius，并返回一个新的点列表，其中包含所有在 radius范围内的输入点，即在由 center和 radius定义的圆内的点：

```
静态List<Pt> inCircle（List<Pt> pts，Pt center，双半径）{
    List<Pt> result = new ArrayList<Pt>();
    for(Pt pt : pts)
      if(pt.distance(center) <= radius)
        result.add(pt);
      return result;
}
```

（理解方法体中的代码并不重要。）

这段代码对于 List<Pt>来说完全正常，但是如果 ColorPt是 Pt的子类型（添加了一个 color字段和相关方法），那么我们不能使用List<ColorPt>参数调用上面的 inCircle方法。 因为深度子类型化与可变字段不一致，List<ColorPt>不是 List<Pt>的子类型。 即使它是，当参数类型为List<ColorPt>时，我们希望有一个结果类型为List<ColorPt>的结果。

对于上面的代码，这是正确的：如果参数是一个List<ColorPt>，那么结果也将是一个List<ColorPt>，但我们希望在类型系统中表达出来。 Java的有界多态性让我们能够描述这种情况（语法细节不重要）：

```
静态<T extends Pt> List<T> inCircle（List<T> pts，Pt center，双半径）{
  List<T> result = new ArrayList<T>();
    for(T pt : pts)
      if(pt.distance(center) <= radius)
        result.add(pt);
      return result;
}
```

这个方法在类型 T上是多态的，但 T必须是 Pt的子类型。 这种子类型是必要的，以便方法体可以在类型为 T的对象上调用 distance方法。太棒了！

可选：额外的Java特定有界多态性

虽然上面的第二个版本的 inCircle是理想的，但现在让我们考虑一些变化。 首先，Java确实有足够的动态检查转换，可以使用第一个版本并将结果从 List<Pt>转换为List<ColorPt>。 我们必须使用“原始类型” List来做到这一点，类似于 cps具有类型List<ColorPt>的情况。

```
List<ColorPt> out = (List<ColorPt>)(List) inCircle((List<Pt>)(List)cps, new Pt(0.0,0.0), 1.5);
```

在这种情况下，这些转换是可以的：如果 inCircle被传递一个List<ColorPt>，结果将是一个List<ColorPt>。 但是这样的转换是危险的。 考虑这个与初始非泛型 inCircle方法具有相同类型的方法变体：

```
静态List<Pt> inCircle（List<Pt> pts，Pt center，双半径）{
    List<Pt> result = new ArrayList<Pt>();
    for(Pt pt : pts)
      如果（pt.distance（center）<=半径）
        result.add(pt);
      否则
        result.add(center);
      return result;
}
```

不同之处在于，圆外的任何点都会被 center“替换”在输出中。 现在，如果我们使用一个List<ColorPt> cps调用 inCircle，其中一个点不在圆内，那么结果不是List<ColorPt>——它包含一个 Pt对象！你可能会期望结果的转换为List<ColorPt>会失败，但是出于向后兼容性的原因，Java不是这样工作的：即使这个转换成功。 所以现在我们有一个类型为List<ColorPt>的值，它不是一个 ColorPt对象的列表。 在Java中，相反的情况是，当我们从这个所谓的List<ColorPt>中获取一个值并尝试将其作为 ColorPt使用时，转换将在稍后失败，因为它实际上是一个 Pt。 责任显然在错误的地方，这就是为什么在第一次使用会引发警告的转换是如此有问题的原因。

最后，我们可以讨论在我们的有界多态版本中，中心参数最适合的类型是什么。 上面，
我们选择了 Pt，但我们也可以选择 T：

```
static <T extends Pt> List<T> inCircle(List<T> pts, T center, double radius) {
    List<T> result = new ArrayList<T>();
    for(T pt : pts)
      if(pt.distance(center) <= radius)
        result.add(pt);
      return result;
}
```

事实证明，这个版本允许的调用者比之前的版本少，例如，第一个参数的类型可以是List<ColorPt>，第二个参数的类型可以是 Pt（因此，通过子类型化，也可以是 ColorPt）。对于类型为 T的参数，当第一个参数的类型为List<ColorPt>时，我们要求一个 ColorPt（或子类型）。 另一方面，我们的版本有时需要将 center添加到输出中，这要求参数的类型为 T：

```
static <T extends Pt> List<T> inCircle(List<T> pts, T center, double radius) {
    List<T> result = new ArrayList<T>();
    for(T pt : pts)
      如果（pt.distance（center）<=半径）
        result.add(pt);
      否则
        result.add(center);
      return result;
}
```

在这个最后的版本中，如果 center的类型为 Pt，则调用result.add(center)无法通过类型检查，因为 Pt 可能不是 T的子类型（我们只知道 T是 Pt的子类型）。 实际的错误消息可能有点令人困惑：它报告 List<T >没有 add方法接受 Pt，这是正确的：我们尝试使用的 add方法接受 T。