# PL第二节小结

构建新类型的概念方法

编程语言有像int、bool、unit之类的基础类型,以及在定义中包含其他类型的复合类型。我们已经看到了在ML中创建复合类型的方法，即通过使用元组类型、列表类型和选项类型。我们很快即将学习新的方法去构造更为灵活的复合类型以及为新类型命名。要创建复合类型实际上只有三个基本的构建模块。任何像样的编程语言都以某种方式提供了这些构建块：

(注：作为一个行话，你不需要知道诸如“each-of 类型”，“one-of 类型”，以及“self-reference 类型”，这些都不是标准的—他们只是思考概念的一个方法。通常人们只是用特定的语言结构像“元组”一样,PL研究者们使用术语“乘类型”，“加类型”，“递归类型”，为什么是乘和加？他其实涉及到布尔逻辑运算0是假1是真，and 像是乘or像是加)

*• “Each-of”: 复合类型t描述的值包含 t1,t2,...和tn*

*• “One-of”: 复合类型t描述的值为 t1,t2,...或tn*

*• “Self-reference”: 复合类型t可以在定义中引用自己，一边描述列表和树等递归数据结构*

each-of 类型是大多数程序员最熟悉的一种，元组就是一个例子：int \* bool描述这个值包含一个int和bool.带有字段的Java类也是一种each-of类型。

one-of类型也是一种很常见的类型，但不幸的是在许多编程课程的介绍中没有太多的强调。int option 就是一个例子：这种类型的值要么包含int要不不包含。在ML中对于一个包含int或bool类型的值，我们需要数据类型绑定，这是我们这一节的重点。在Java等具有类的面向对象的语言中，one-of类型是通过子类化方式实现，但这是本课程后面的主题。

Self-reference（自引用类型）允许类型描述递归数据结构。这在与each-of 和one-of组合使用时很有用。例如，int 列表描述值要么什么都不包含，要么包含一个int和int 列表。在任何编程语言中，整数列表都可以用or、and和self-reference来描述，因为这就是整数列表的含义。

自然地，因为复合类型可以嵌套，我们可以有任意嵌套的each-of，one-of和self-reference。例如，( int \* bool ) list list \* ( int option ) list \* bool 这种类型。

记录(Records): 另一种处理“Each-of”类型的方法

记录类型是一种“each-of”类型，其中每一个组成部分都是一个命名字段。例如，{foo : int, bar : int\*bool, baz : bool\*int} 描述有三个字段为foo,bar,和baz的记录。这只是一种新的类型，就像我们学习元组类型一样。

一个记录的表达式创建了一个记录值，例如，表达式{bar = (1+2,true andalso true), foo = 3+4, baz = (false,9) } 将会计算为 {bar = (3,true), foo = 7, baz = (false,9)}，它的类型是 {foo : int, bar : int\*bool, baz : bool\*int} 因为字段的顺序不重要。通常来说一个记录表达式的语法是：{f1 = e1, ..., fn = en}，其中，像往常一样，每一个ei 可以是任何表达式。这里每个f可以是任何字段名(每一个应该不同)。字段名基本上是任何字母或数字序列。

在ML中，我们不需要声明我们想要的具有特定类型字段名称和字段类型的记录，我们只需要写下一个记录表达式，检查器就会给它正确的类型。记录表达式的类型检查规则并不奇怪：对每个表达式进行类型检查得到类型ti,然后构建具有所有正确类型的正确字段的记录类型。因为字段的顺序并不重要，为了保证打印的一致性，REPL总是按照字母表顺序排列并打印它们。

记录表达式的求值规则类似：将每个表达式求值为一个值并创建对应的记录值。

现在我们知道如何去构建记录值，我们需要一种方法来获取到它们的片段。我们使用#foo e,其中foo是一个字段名，类型检查要求e是一个记录类型并且有一个叫做foo的字段，如果这个字段具有类型t，那它就是#foo e 的类型。求值将e求值为一个记录值，然后生成foo字段的内容。

按名vs. 按位，语法糖，以及元组的真相

记录和元组非常相似。他们都是“each-of”的构造，允许任意多的元素。唯一的区别是记录是“按名”而元组是“按位”。这意味着对于记录我们创建它并通过使用字段名获取他们的每一部分，所以在一个记录表达式中我们写下的字段顺序并不重要。但元组没有字段名，所以我们使用位置（first,second,third,...）去区分元素。

在设计语言结构或决定选择哪一种时，按名和按位是一个经典的决定，他们在某些情况下更方便。作为一个粗略的指导，对于较少的元素按位，但对于较大的复合类型，对于记住在哪个位置是很困难的。

Java的方法参数（以及我们到目前为止描述过的ML函数参数）实际上采取了一个混合的方法：方法体使用变量名指向不同的参数，但是调用者是按位传递参数的。也有其他的语言调用者按名传递参数。

尽管有按名按位之分，记录和元组还是如此相似，以至于我们可以完全按照记录来定义元组。如下的方式：

*• 当你写（e1,...,en）,其实 {1=e1,...,n=en}的另一种写法，即一个元组表达式是一个具有字段名为1，2，...，n 的记录。*

*• 类型 t1 \* ... \* tn 也只是{ 1:t1，...，n:tn}的另一种写法*

*• 注意#1 e,#2 e等现在也已经是正确的含义：获取名为1，2等字段的内容。*

事实上，这是ML定义元组的真正方式：元组就是记录。也就是说，元组的所有语法都只是使用记录的一种方便的方式。REPL总是在可能的地方使用元组语法，所以当你求值{2=1+2, 1=3+4} 总是会打印结果为 (7,3)。使用元组的语法是更好的方式，但我们不需要给元组自己的语义：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。

这是我们将要看到的**语法糖**许多例子中的第一个。我们说“元组只是字段名为1,2,...,n的记录的语法糖。”。它是*语法的*因为我们可以用等价的记录语法来描述关于元组的一切。它也是糖因为它让语言更加甜美。术语*语法糖*被广泛使用。语法糖是一种很好的方法，在使程序语言更小的情况下保留关键思想，同时让程序员更方便的书写。确实，在家庭作业1中我们使用元组的时候根本不知道记录的存在即是元组也是记录。

数据类型绑定：我们自己的“One-of”类型

我们现在介绍 *数据类型绑定 datatype bindings,* 是我们继变量绑定和函数绑定后的第三种绑定。我们用一个有点傻但是简单的例子开始，因为它将帮助我们了解数据类型绑定的许多同方面。我们可以这么写：

datatype mytype = TwoInts of int \* int

| Str of string

| Pizza

粗略地说，这定义了一种新的类型，它的值是int\*int 或者一个string或者什么也不是。任何的值也会被标记为信息让我们知道它是哪个变量：这些标记，我们将会称为构造器的是TwoInts,Str和Pizza。可以用2个构造函数来标记同一类型的基础数据；事实上，即使我们的示例为每个变量使用不同的类型，这也是常见的。

更精确的说，这个例子向环境中添加了4个事情：

*• 一个我们现在可以向其他类型一样使用的新类型 mytype*

*• 三个构造器 TwoInts, Str, Pizza*

一个构造器是2个不同的东西。首先，它要么是一个创造新类型值的函数，要么它实际上是一个新类型的值。在我们的例子中，TwoInts 是一个类型为 int\*int -> mytype的函数， Str 是一个 string -> mytype的函数，Pizza是一个类型为mytype的值。其次，我们在后面的的描述中会在case表达式中使用构造器。

所以我们知道如何去构建mytype类型的值：用右边类型的表达式（或者仅仅是Pizza这样的值）调用构造器（它们是函数）。这些函数调用的结果是“知道他们是哪个变量”（它们存储一个“标签”）的值，并将底层函数据传递给构造函数，REPL这样表示这些值TwoInts(3,4) or Str "hi".

剩下的是获取这些值的方法。。。

ML如何不提供对数据类型值的访问

给一个类型为 mytype的值，我们如何访问存储在其中的数据？首先，我们需要找到是哪一个变量，因为mytype是TwoInts,Str,或Pizza生成的，这影响了可用的数据。一旦我们知道我们有的是什么变量，我们可以访问该变量所携带的部分。

回想一下我们是如何为list 和option实现这一功能的：我们有函数（null 或 isSome）来测试我们有哪种变量，我们有函数（hd，tl,或valOf）来获取内容，当所给参数是错误类型的时候会抛出异常。

ML对数据类型绑定也可以用同样的方法。例如，它可以将我们上面定义的数据类型添加到环境函数isTwoInts,isStr,和isPizza（都是mytype->bool类型）。并且它也可以添加像mytype->int \* int 类型的getTwoInts，以及mytype->string 类型的getStr函数，这可能会抛出异常。

但是ML不采取这种方法，它做的更好。你可以用更好的东西编写这些函数，尽管这样做通常是糟糕的风格。事实上，在学习更好的东西之后，我们将不再像以前那样使用list和option的函数——我们只是开始使用这些函数，以便我们一次可以学习一个内容。

ML怎样提供访问数据类型值的方法：Case 表达式

更好的方式是使用case表达式。这里是一个对于我们数据类型绑定例子的一个简单例子：

fun f x = (\* f has type mytype -> int \*)

case x of

Pizza => 3

| TwoInts(i1,i2) => i1 + i2

| Str s => String.size s

在某种意义上，一个case表达式像是一个更强大的if-then-else表达式：像是一个条件表达式，它计算它的2个子表达式：第一个表达式在case和of关键字之间，第二个表达式在匹配的第一个分支中的表达式，但是没有2个分支（一个表示true,一个表示false）,对于我们数据类型的每个变量我们只会有一个分支（我们将在下面进一步一般化）。像是条件表达式，每个分支表达式必须有同样的类型（在上面的例子中是int），因为类型检查不知道哪个分支将会被我们使用。

每个分支都有这种形式 **p => e**，其中**p**是模式（pattern）而 **e**是一个表达式，我们使用 | 符号来把每个分支分开。模式看起来像是表达式，但不要把它们认作表达式。相反，它们是用来*匹配*case的第一个表达式的结果（**case**后面的那部分）。这就是为什么求值一个case表达式被叫做模式匹配（*pattern-matching*）。

在这一课中，我们让模式匹配简单些：每一个模式使用一个不同的构造器以及模式匹配选择一个和case关键字后面给定的表达式“正确的那一个”的分支。这个分支求值的结果就是全部的答案；其他的分支不会被求值，例如，如果把TwoInts(7,9) 传给f,则第二个分支将会被选择。

这种方式处理了使用one-of类型时候的“变量检查”部分，但模式匹配同时也处理了“获取基础数据”的部分。因为TwoInts “携带” 2个值，它的模式可以（就目前来说，必须）使用2个变量（（i1,i2））。作为匹配的一部分，相对应的值（在我们的例子中是7和9）的部分在环境中被绑定到i1和i2用来计算相应的右边的部分（i1+i2）。这意味着，模式匹配像是一个let表达式：它在一个局部作用域中绑定变量。类型检查器知道这些变量的类型，因为他们被在模式中使用创建的构造器数据类型绑定中特别指定了。

为什么在测试变量以及提取内容上case表达式比函数更好呢？

*• 我们永远不会从错误的变量中提取数据，也就是说我们不会得到hd[]这样的异常。*

*• 如果case表达式中忘记了一个变量，那么类型检查器会给出一个警告信息，这意味着对case表达式求值有可能会到一个不匹配的分支，这种情况下会抛出异常，如果你没有这样的警告，那这种情况就不会发生。*

*• 如果一个case表达式对一个变量使用了2次，类型检查器将会给出一个错误信息，因为其中一个分支永远不会被使用到。*

*• 如果你还是想使用null或者hd这样的函数，你可以很容易的自己编写他们。*

*• 模式匹配要比我们目前所提到的更加通用与强大，我们在下面会指出模式匹配的全部真相。*

One-of 类型的有用示例

让我们现在思考下几个关于“one-of”类型的有用例子，因为目前为止我们考虑的是一个愚蠢的例子。

首先，它们对枚举一组固定选项的集合很有用-而且这笔使用小整数要好的多，且更具风格，例如：

datatype suit = Club | Diamond | Heart | Spade

很多语言都支持这种枚举，包括JAVA和C，但是ML采取了更进一步的方法，允许变量携带参数，我们可以这样做：

datatype rank = Jack | Queen | King | Ace | Num of int

然后我们可以用一个each-of的类型将这两个结合起来：suit \* rank

当你在不同的情况下有不同的数据，one-of 也是非常有用的。例如，假设你想通过学生的id号来识别学生，但是有些情况下学生没有（有可能他们是新生），那么你就要使用他的全名来代替（有名，中间名，姓），这个数据类型绑定直接抓住了这个思想：

datatype id = StudentNum of int

| Name of string \* (string option) \* string

不幸的是，这种情况通常会反映出程序员对“one-of”类型的理解不深刻，而坚持使用“each-of”类型，这就相当于把锯子当锤子使（它也许起作用，但这是错误的），考虑到他们的错误代码如下：

(\* If student\_num is -1, then use the other fields, otherwise ignore other fields \*)

{student\_num : int, first : string, middle : string option, last : string}

这种方法要求所有的代码遵循注释中的规则，没有来自类型检查器的帮助。这同样也非常浪费空间，每一条记录中都有从未使用过的字段。

另一方面，如果我们想要存储每个学生的ID（如果有的话）和他的姓名信息时，each-of 类型是相当正确的方法：

{ student\_num : int option,

first : string,

middle : string option,

last : string }

我们的最后一个例子是一个算术表达式的数据定义，它包括常量、负数、相加、相乘。

datatype exp = Constant of int

| Negate of exp

| Add of exp \* exp

| Multiply of exp \* exp

多亏了self-reference，这个数据定义实际上描述的是一个树，该树的叶子节点是整数，中间节点是有一个子节点的负数或是有两个子节点的加法或是有两个子节点的乘法。我们可以写这样一个函数，传入exp并计算：

fun eval e =

case e of

Constant i => i

| Negate e2 => ~ (eval e2)

| Add(e1,e2) => (eval e1) + (eval e2)

| Multiply(e1,e2) => (eval e1) \* (eval e2)

所以，下面这个函数调用的计算结果是15：

eval (Add (Constant 19, Negate (Constant 4)))

注意构造函数只是我们用其他表达式（通常是由其他构造函数构建出来的值）来调用的函数。

我们可以在exp类型的值上写许多函数，大部分都要用到类似的模式匹配和递归，下面是一些其他可以处理exp参数的函数：

*• 一个表达式中最大的常量*

*• 一个表达式中的所有常量的列表*

*• 一个布尔值表示表达式中是否至少有一次乘法*

*• 表达式中加法表达式的数量*

下面是最后一个函数的写法

fun number\_of\_adds e =

case e of

Constant i => 0

| Negate e2 => number\_of\_adds e2

| Add(e1,e2) => 1 + number\_of\_adds e1 + number\_of\_adds e2

| Multiply(e1,e2) => number\_of\_adds e1 + number\_of\_adds e2

数据绑定和Case表达式目前为止的小结

就目前我们所知道的数据类型和模式匹配我们可以总结如下：

datatype t = C1 of t1 | C2 of t2 | ... | Cn of tn

这个绑定引入了一个新的类型t，并且每一个Ci构造器都是一个类型为ti -> t的函数，对于不携带任何东西的变量，忽略了“of ti”,这样的构造函数只有类型t.要获得t中的部分，我们使用case表达式：

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

一个case表达式将e计算至v, 找到第一个匹配v的pi，然后计算ei作为整个表达式的计算结果，目前为止，模式看起来像Ci(x1,...,xn) ,Ci是一个类型为t1 \* ... \* tn -> t 的构造器（或者仅仅就是Ci, 如果Ci 什么也不携带的话），这样的一个模式匹配了一个来自Ci(v1,...,vn)

的值，并且把每一个xi绑定到vi去计算对应的ei。

类型同义

在继续讨论数据类型之前，让我们把它和另一个有用的绑定比较，同时引入一个新的类型名称。类型同义是为一个已存在的类型创建另一个名字并且2者之间是完全可以互相替换的。例如，如果我们写：

type foo = int

那么我们可以在任何写int的地方写foo，反之亦然。所以给定一个类型为foo -> foo的函数，我们可以用3调用该函数，并将结果加4。REPL视情况有时会打印foo,有时会打印int；

细节不重要，取决于语言的实现。对于int这样的类型，这样的一种同义词并不是很有用（尽管稍后我们将学习的ML模块系统基于这个特性之上）。

但是对于许多的复杂类型，创建同义类型将会非常方便，下面是2个我们之前创建过的例子：

type card = suit \* rank

type name\_record = { student\_num : int option,

first : string,

middle : string option,

last : string }

只要记住，他们是完全可以互相替换的，例如作业中要求函数类型为 card -> int,如果REPL报告你的答案有suit \* rank -> int，这是没问题的，因为它两完全一样。

相比之下，数据类型绑定引入了一个与现有类型完全不相同的类型。它创建生成这种新类型值的构造函数。所以，举例来说，除非我们为suit引入一个同义类型,否则和suit相同的类型只有suit。

Lists和Options也是数据类型

因为数据类型的定义可以是递归的，我们可以使用它们来创建我们自己的某种类型的lists.

这种绑定对整数链表非常有效：

datatype my\_int\_list = Empty

| Cons of int \* my\_int\_list

我们可以使用Empty和Cons构造函数来创建my\_int\_list类型的值，还可以使用case表达式去使用这种类型的值：

val one\_two\_three = Cons(1,Cons(2,Cons(3,Empty)))

fun append\_mylist (xs,ys) =

case xs of

Empty => ys

| Cons(x,xs’) => Cons(x, append\_mylist(xs’,ys))

事实证明，内置的（即预定义了一些特殊的语法支持）lists和option也是数据类型。就风格而言，最好是使用广泛被使用的内置特性，而不是你自己所创造的。

更为重要的是，使用模式匹配而不是我们之前说的null、hd、tl、isSome这样的函数来访问list和option的值是很好的风格。

对于option来说，你只要知道SOME和NONE是构造函数，我们用来创建值以及在模式匹配中访问值，下面是后者的一个简短的例子：

fun inc\_or\_zero intoption =

case intoption of

NONE => 0

| SOME i => i+1

列表的情况与之类似，只是有一些方便的语法特性：[]是一个不携带任何参数的构造器，::是携带2个参数的构造器，但::还是不常见的，因为它是一个中缀操作符（它在两个操作数中间），创建及模式匹配如下：

fun sum\_list xs =

case xs of

[] => 0

| x::xs’ => x + sum\_list xs’

fun append (xs,ys) =

case xs of

[] => ys

| x::xs’ => x :: append(xs’,ys)

注意这里的x和xs’只是通过模式匹配引入的局部变量，我们可以使用我们想要的任何名称，

我们甚至可以用hd和tl——这样做只是隐藏了在外部环境预定义的函数。

为什么在访问lists和options优先使用模式匹配而不是使用像null、hd函数的原因通常来说和数据类型绑定的原因相同：你不能忘记Case表达式，你不能错误的应用函数，等等。但如果这种方法较差，那为什么ML环境要预定义这些函数呢？部分原因是因为它们适用于作为其他函数的参数传递，这是课程下一部分的主要话题。

多态数据类型

除了[]和::的奇怪语法之外，内置lists、options和我们示例中的数据类型绑定唯一的区别是内置的两个是多态的（*polymorphic*）——他们可以被用于携带任何类型的值，正如我们之前所见的int list, int list list, (bool \* int ) list 等等。你也可以对你自己的数据绑定这样做。事实上，它非常适用于构建“泛型 generic”数据结构。我们不会在此处专注这个特性，这也没什么复杂的，例如，option就是这样在环境中被预定义的：

datatype ’a option = NONE | SOME of ’a

这种绑定并没有引入一个option类型，它使得如果t是一个类型，那么t option 也是一个类型。你也可以定义有多种类型的多态数据类型，例如，这是一个中间节点是’a类型的值，叶子节点是’b类型的值的一棵二叉树。

datatype (’a,’b) tree = Node of ’a *\* (’a,’b) tree \** (’a,’b) tree

| Leaf of ’b

我们还有像( int \* int ) 树（每个节点和叶子都有一个int值），( string \* bool ) 类型的树（每个节点都有一个string类型的值，每个叶子都有一个bool类型的值），对于普通的数据类型和多态数据类型，你使用构造器和模式匹配的方式是一样的。

对于Each-Of类型的模式匹配：值绑定的真相

目前为止我们对one-of类型使用过模式匹配，其实我们也可以对each-of类型使用。给一个记录类型的值{f1=v1,...,fn=vn}，模式{f1=x1,...,fn=xn}会将xi和vi匹配并绑定在一起。正如你期望的那样，模式中的字段顺序并不重要。之前说到，元组只是记录的语法糖：模式(x1,...,xn) 和{1=x1,...,n=xn}是相同的，并且匹配元组(v1,...,vn)的值和{1=v1,...,n=vn}也是相同的，所以我们可以写这样一个函数，对一个int \* int \* int 的三个部分求和：

fun sum\_triple (triple : int \* int \* int) =

case triple of

(x,y,z) => z + y + x

一个简单的records（以及ML中的字符串拼接操作符）的例子如下：

fun full\_name (r : {first:string,middle:string,last:string}) =

case r of

{first=x,middle=y,last=z} => x ^ " " ^ y ^ " " ^z

然而，对于一个只有一个分支的cas表达式是非常糟糕的风格——它看起来很奇怪，因为这种表达式的目的是区分多种case，所以，当我们知道单个模式一定会匹配而我们使用模式匹配只是为了提取值的情况下，我们应该如何使用模式匹配呢？事实证明，您也可以在值绑定中使用模式，下面的方法是一个较好的风格：

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

事实上我们还能做的更好：正如一个模式可以用在值绑定中来绑定变量（例如，x,y,z）到表达式（eg. tripe）的各个部分,我们可以在定义一个函数绑定时使用一个模式，该模式将会通过与调用函数时传入的值进行匹配来引入绑定。因此下面是我们第三个也是最好的一个例子：

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\_tripe应该会引起你的兴趣：它携带一个tripe作为参数，使用模式匹配将这三部分绑定到了3个在函数体内使用的变量，但这看起来就像是一个函数携带了3个类型为int的参数，确实，类型为int \* int \* int -> int 是携带三个参数的函数，还是携带一个tripe类型参数的函数？

事实证明其实我们一直在撒谎：在ML中压根就没有多参数函数这种东西，**ML中的每个函数都只携带一个参数！**每当我们写下一个多参数函数，我们实际上写的是携带一个元组作为参数的单参数函数，并且他使用模式匹配去提取值。这是一个很常见的风格以至于经常会被忘记，在讨论ML代码话题时说“多参数函数”是完全没有问题的。但是就实际的语言定义来说，它确实是个单参数函数：语法糖只是为了扩展第一个单分支case表达式版本的sum\_triple。

这种灵活性有时候是非常有用的。在像C和Java这样的语言中，你不能把一个函数/方法的计算结果立即传入另一个多参数的函数/方法。但是对于只有一个参数即元组的函数来说，这没问题。这里有个愚蠢的例子，我们通过将一个triple向左旋转2次达到向右旋转的目的：

fun rotate\_left (x,y,z) = (y,z,x)

fun rotate\_right triple = rotate\_left(rotate\_left triple))

更通俗的说，你可以计算元组然后将他们传递给函数，即使这个函数的编写者考虑的是多个参数。

那么0参数函数呢？他们也并不存在。这个绑定 fun f () = e 使用单元模式() 去匹配传递unit值 ()的调用，该值是unit 类型的唯一值。unit类型只是一个只有一个构造函数的数据类型，它不接收任何参数并且使用不常见的语法()。基本来说，datatype unit = () 是预定义的。

题外话：类型推断

通过使用模式去获取tuples和records的值而不是通过#foo，你可能已经意识到在函数参数中写下类型已经不再有必要了。事实上，在ML中，将他们去掉是惯例——你总是可以从REPL中来查看函数的类型。我们之前需要他们是因为#foo 并没有给函数的类型检查足够多的信息，因为类型检查器不知道records应该还有哪些字段，但是上面介绍的record/tuple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（否则你的程序将不会通过类型检查）——类型推断仅仅意味着你不必再写下这些类型。

所以上面使用模式匹配而不是#foo和#2的例子都不需要参数类型。写这些不是那么杂乱的版本通常是更好的风格，再次说明最后一种是最好的：

fun sum\_triple triple =

case triple of

(x,y,z) => z + y + x

fun sum\_triple triple =

let val(x,y,z) = triple

in

x + y + z

end

fun sum\_triple (x,y,z) =

x + y + z

这个版本需要一个显示的实参类型：

1. fun sum\_triple (triple : int \* int \* int) =
2. #1 triple + #2 triple + #3 triple

因为类型检查器不能够这样：

fun sum\_triple (triple : int \* int \* int) =

#1 triple + #2 triple + #3 triple

并且推断参数的一定有 int \* int \* int这样的类型，因为它也可能是int \* int \* int \* int

或者 int \* int \* int \* string 或者 int \* int \* int \* bool \* string以及无数的其他类型。如果你不使用#，由于类型推断的便利性，ML几乎不需要显示的类型注释。

事实上，有时候类型推断会揭示出函数比你想象的更通用。考虑下面这段代码，他确实使用了元组(tuple)/记录(record)的一部分:

fun partial\_sum (x,y,z) = x + z

fun partial\_name {first=x, middle=y, last=z} = x ^ " " ^ z

在上面的2种情况中，被推断出来的类型都揭示了y可以是任何类型，所以我们可以这样调用partial\_sum (3,4,5) 或 partial\_sum (3,false,5)。

我们会在以后的章节中讨论*多态函数(polymorphic functions)*以及*类型推断(type inference)*是如何工作的，因为他们是我们课程要讨论的主题。现在的话，只需要停止使用#，停止写下参数类型，如果偶尔因为类型推断看到 ’a或 ’b这样的类型，不要困惑，这正如前面所讨论的那样。。。

题外话：多态类型和相等类型

我们现在鼓励您不要在程序中使用显示的类型注释，但从上面的例子来看，这可能导致出现令人意外的泛型。假设你被要求写一个类型为 int \* int \* int -> int ，行为和partial\_sum一样的函数，但REPL会指出，partial\_sum的类型是 int \* ’a \* int -> int。这是没问题的，因为多态性表明partial\_sum具有更一般的类型。如果你可以采取包含'a，'b，'c等类型的类型，并一致地替换每个这些类型变量，以获得你“想要的”类型，那么你拥有的类型比你想要的更通用。

再举个例子，我们写过的append函数类型为’a list \* ’a list -> ’a list, 因此通过一致性把’a替换成string，我们可以将append的类型视为string list \* string list ->string list。我们可以用任何一种类型替换，不仅仅是string。实际上我们不会真的去做任何类型：这只是第一种思维训练，用来检查一个类型是否比我们所需要的类型更通用。请注意，像’a这样的类型变量必须一致的替换，这意味着append的类型不能比string list \* int list -> string list更为一般化。

你也会看到’’a这样的类型变量，这些被称为***equality types*** (相等类型)，它是ML的一个奇怪的特性，与我们目前学习的无关。基本上，ML的=操作符(用来比较的)适用于许多类型，不止是int，但是它比较的两个操作数必须是相同类型。例如，它既适用于字符串类型，也适用于元组中所有类型都支持相等的元组类型。但不是对每一种类型都有效，比如’’a类型，这种类型的比较只能用***equality types***来代替它。

再次说明，我们后面会继续讨论多态类型和类型推断，这次题外话只是为了避免在Homework2中避免困惑，因为在Homework2中：如果你写了一个函数，REPL给你返回一个比你所需要的更加一般化的类型，这是正常的。同时记住，如上所述，如果REPL使用的类型同义词与你所期望的不一致，也没关系。

嵌套模式

事实证明，模式的定义是递归的：我们在模式中任何位置的变量都可以使用模式来代替，粗略的说，模式匹配的语义是要匹配的值必须与模式有相同的“形状”，并且变量必须要绑定到“正确的部分”。（这是一个非常简短的解释，这就是为什么下面描述了一个精确地定义。）

例如模式a::(b::(c::d)) 匹配至少3个元素的列表，并且将会绑定a作为第一个元素，b是第二个，c是第三个，以及d包含所有其他的元素列表(如果有的话)。而a::(b::(c::[ ]))模式，只会匹配刚好有3个元素的列表。另一种嵌套模式(a,b,c)::d，只能匹配任意非空的三元组列表，将a绑定到头部元素的第一个部分，b绑定到head的第二部分，c绑定到head的第三部分，d绑定到列表的尾部。

一般来说，模式匹配是讨论一个值和一个模式（1）确定模式和值是否匹配 （2）如果匹配，绑定变量到值的正确匹配的部分。以下就是模式匹配的优雅递归定义：

*• 一个变量模式（x）与任意的值（v）匹配，并且引入一个绑定（从x到v）。*

*• 模式C匹配值C，如果C是一个构造函数且不携带任何数据。*

*• 模式C p其中C是构造函数，p是模式，与形式为C v的值匹配（注意构造函数是一样的）如果p与v匹配（即嵌套模式与携带值匹配）。它引入了p匹配v引入的绑定。*

*• 模式 (p1,p2,...,pn) 匹配元组值 (v1,v2,...,vn)，如果 p1匹配v1，p2匹配v2，...，pn匹配vn，它引入了递归匹配引入的所有绑定。*

*• （类似的案例如改形式的记录模式{f1=p1,...,fn=pn} ...）*

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，对于携带多个参数的构造函数C，我们不必写像C(X1,...Xn)这样的模式。我们可以写C x; 这样可以直接将x绑定到C(X1,...Xn) 所携带的值元组(tuple)上，实际上真正发生的是所有的构造函数都携带0或1个参数，但是这1个参数有时候可能本身是个元组(tuple)。所以C(X1,...Xn) 实际上是个嵌套模式，其中(x1,...,xn) 是一个匹配所有有n个部分的元组。其次，更为重要的是我们可以使用嵌套模式来代替使用嵌套的case表达式当我们指向匹配具有特定形式的值。

还有一些其他类型的模式，有时候我们并不需要绑定变量到值上。例如，下面这个计算列表长度的函数：

fun len xs =

case xs of

[] => 0

| x::xs’ => 1 + len xs’

我们不使用这个变量x，在这个例子中，最好不要引入变量，使用通配符\_可以匹配一切(就像一个变量匹配一切)，但不要引入绑定，所以我们应该这样写：

fun len xs =

case xs of

[] => 0

| \_::xs’ => 1 + len xs’

根据我们的定义，通配符的模式非常简单：

*• 一个通配符模式（\_）匹配任意值v，但不引入绑定。*

最后，你可以在模式中使用整型常量，例如，模式37匹配值37，并且不引入绑定。

**嵌套模式的有用例子**

使用嵌套模式而不是乱七八糟的一堆混乱的嵌套case表达式的一个优雅示例是“压缩”与“解压缩”列表（本例中有三个）：

fun zip3 list\_triple =

case list\_triple of

([],[],[]) => []

| (hd1::tl1,hd2::tl2,hd3::tl3) => (hd1,hd2,hd3) :: zip3(tl1,tl2,tl3)

| \_ => raise ListLengthMismatc

fun unzip3 lst =

case lst of

[] => ([],[],[])

| (a,b,c)::tl => let val (l1,l2,l3) = unzip3 tl

in

(a::l1,b::l2,c::l3)

end

下面这个例子检测整数列表是否是排过序的：

fun nondecreasing intlist =

case intlist of

[] => true

| \_::[] => true

| head::(neck::rest) => (head <= neck andalso nondecreasing (neck::rest))

有时候通过匹配一对值来比较两个值也很优雅，这个例子中，通过不进行乘法运算来确定乘法的符号，虽然有点简单，但是证明了这个想法：

datatype sgn = P | N | Z

fun multsign (x1,x2) =

let fun sign x = if x=0 then Z else if x>0 then P else N

in

case (sign x1,sign x2) of

(Z,\_) => Z

| (\_,Z) => Z

| (P,P) => P

| (N,N) => P

| \_ => N (\* many say bad style; I am okay with it \*)

end

最后一个case表达式的风格值得讨论：当你在底部加上这样一个“包罗万象”的情况时，你放弃了任何检查，但是你没有遗漏任何情况：毕竟，它匹配任何一种前面的case没有匹配到的情况，所以类型检查器当然不会觉得你遗漏了任何情况。因此，如果你使用这个技巧，你必须格外小心，并且枚举其余情况（在本例中是(N,P) and (P,N)）可能不太容易出错。类型检查器仍然会确定没有遗漏的情况，这是有用且必要的，因为它必须对(Z,\_) 和 (\_,Z)的使用进行推理已确定类型sgn \* sgn 没有遗漏的情况。

***可选:* 函数绑定中的多个用例**

到目前为止，我们已经看到了case表达式中对one-of 这种类型的模式匹配，我们还看到了在值或函数绑定中对 each-of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。但在函数或值绑定中是否有一种方法针对one-of匹配？这看起来似乎是个坏主意因为我们需要多种可能性。但ML在函数定义中有特殊的语法，下面是2个例子，一个是我们自定义的数据类型一个是列表：

datatype exp = Constant of int | Negate of exp | Add of exp \* exp | Multiply 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)

由于品味的问题，你的老师可能不太喜欢这种风格，你必须在正确的地方使用括号，但这在ML程序员中很常见，所以也欢迎您这样做。从语义上说，它只是一个case表达式的单个函数体的语法糖：

fun eval e =

case e of

Constant i => i

| Negate e2 => ~ (eval e2)

| Add(e1,e2) => (eval e1) + (eval e2)

| Multiply(e1,e2) => (eval e1) \* (eval e2)

fun append e =

case e of

([],ys) => ys

| (x::xs’,ys) => x :: append(xs’,ys)

简单来说，语法：

fun f p1 = e1

| f p2 = e2

...

| f pn = en

只是下面的语法糖：

fun f x =

case x of

p1 => e1

| p2 => e2

...

| pn => en

注意append示例使用了嵌套模式：通过将模式（例如，[]或x::xs’）放入到其他模式中,每个分支匹配一对列表。

**异常**

ML有一个内置的异常概念。你可以使用rise原语来引出rise（也叫抛出throw）异常。例如，标准函数库中的hd函数会在调用[]的时候抛出List.Empty异常：

fun hd xs =

case xs of

[] => raise List.Empty

| x::\_ => x

你可以使用异常绑定创建自己的异常类型，异常可以选择性的携带值，这让引发异常的代码提供更多的信息：

exception MyUndesirableCondition

exception MyOtherException of int \* int

异常的种类很像数据结构绑定的构造函数。实际上，它们是函数（如果它们携带值）或者值（如果不带值），它们创建的值类型为exn而不是数据类型的类型。所以Empty,MyUndesirableCondition,MyOtherException(3,9)都是类型为exn的值，而MyOtherException 的类型是 int \* int -> exn。

通常我们只使用异常构造函数作为raise的参数，例如raise MyOtherException(3,9)，但我们可以更普遍的使用它们来创建exn类型的值。例如，这是返回整数列表中最大元素的函数的一个版本，当调用[]时，比起返回一个option或者抛出像List.Empty的特定的异常，这个函数会携带一个exn类型的参数并抛出，因此调用者可以传入其自己选择的异常。（类型检查器会检查ex的类型是否为exn，因为这个类型是raise的参数所要求的。）

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; 此调用向函数传递一个异常值，然后函数不会引发该值。

另一个异常相关的特性是*handling (也就是catching)* 异常，为此，ML有处理的表达式，看起来像是：e1 handle p => e2，其中e1,e2是表达式，p是模式，匹配一个exn。这个语义是指，对e1求值作为结果，但如果e1引发了一个匹配p的异常，那么e2将会被求值并作为整个结果，如果e1引发的异常没有匹配到p，则整个handle表达式也会抛出异常。同样的，如果e2引发了异常，则整个表达式也会抛出异常。

正如case表达式一样，handle表达式也可以有许多分支，每个分支都有一个模式和一个表达式，在语法上用|分隔。

**尾递归和累加器**

这个主题设计到一个新的编程习惯用法，但是没有新的语言结构。它定义了尾递归，描述了它与在ML这样的函数式编程语言中编写高效的递归函数有怎样的关系，并且介绍了如何使用累加器作为一种技术来实现一些函数的尾递归。

要理解尾递归和累加器，看下面这几个将列表中所有元素相加的函数：

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

两个函数的计算结果相同，第二个函数更为复杂些，使用了一个局部辅助函数，它携带了一个额外的acc参数，也就是累加器。在f函数的最基础的case中我们返回acc，最外层调用的acc我们传递的是0，这个值我们在sum1中最基本的case中使用。这部分是常见的：

非累加器风格的基本case称为初始累加器，而累加器风格的基本case只返回累加器。

为什么当sum2明显更为复杂却更受欢迎呢？要回答这个问题，我们需要稍微了解一下函数调用是如何实现的。从概念上说，有一个调用栈（一种可以进行压入push和弹出pop先进后出操作的数据结构），里面存放着每一个已经调用但是还没有完成的函数调用元素。每一个元素中存储像局部变量和函数中尚未求值的部分之类的信息，当一个函数体的求值调用另一个函数时，一个新的元素就被压入了调用栈，并在被调用完成后弹出。

对于sum1来说，每一个对sum1的递归调用都有一个调用堆栈元素（有时候它被叫做堆栈帧），也就是说堆栈将会和这个列表一样大，这是必要的，因为在弹出每个堆栈帧之后，调用者必须“执行主体的其余部分”——即将i加到递归结果中并返回。

根据目前为止的描述，sum2也没有更好：sum2调用f, 然后对每个列表元素递归调用。然而，当f递归调用f之后，调用者没什么可做的了当被调用方返回调用结果后，这种情况就叫做*尾递归*并且像ML这样的函数式语言提供一个基本的优化：当一个调用是一个尾调用的时候，调用者的堆栈帧在调用之前弹出——被调用者的堆栈帧替代了调用者的。这是有道理的：调用者无论如何都会返回被调用者的结果。因此，调用sum2永远不会占用超过1个堆栈帧。

为什么函数式语言的实现包含了这种优化？这样做，递归有时候可以向while循环一样高效，这也不会使调用栈越来越大，这里的“有时候”指的正是指尾调用的时候。这是程序员可以推理的，因为你可以查看代码并识别那些调用是尾调用。

尾调用不必是对同一个函数的调用（f可以调用g），所以它们比while循环总是要调用同一个循环更加灵活。使用累加器是一个常见的将递归函数转变为“尾递归函数”的方法（其中所有递归调用都是尾调用），但并不总是这样。例如，函数通常在处理树状的数据结构的时候其调用堆栈通常会树的深度一样增长，但这在任何语言中都是如此：while循环在处理树的时候不是非常有用。

**关于尾递归的更多例子**

尾递归在处理列表的函数中很常见，但其概念更为通用。例如下面是2个阶乘函数的实现，第二个使用了尾递归辅助函数，因此它只需要一个小的常量调用堆栈空间。

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 结果相同，尽管前面的形式是(4\*(3\*(2\*(1\*1)))),而后面的形式是(((1\*4)\*3)\*2)\*1。我们依赖于乘法的结合律以及乘以1是恒等函数(1\*x=x\*1)这一事实。

前面求和的例子对加法作了类似的假设，一般来说，讲一个非尾递归函数转变为一个尾递归函数通常需要结合性，但许多函数是结合性的。

一个更有趣的例子是这个低效的列表反转函数：

fun rev1 list =

case list of

[] => []

| x::xs => rev1(xs) @ [x]

我们可以立刻识别这不是一个尾递归因为在递归调用后他仍然需要将结果附加到这个列表的头个元素组成的单元素列表中。尽管这是最自然的方式去反转一个列表，效率低下是由于创建了一个深度等同于参数长度的调用堆栈，我们将其称为n。更糟糕的是，完成的总工作量与n的平方成正比，也就是说这是一个二次算法。原因是，将2个列表追加到一起所需要的时间和第一个列表的长度成正比：它必须遍历第一个列表。在所有对rev1的递归调用中，我们调用@时，第一个参数长度为n-1,n-2,...,1,从1到n相加为n(n-1)/2。

就像你在数据结构和算法课程中学到的那样，对于足够大的n，这样的二次算法要比线性算法慢得多。也就是说如果你希望n总是很小，那么对程序员的时间和坚持使用简单的递归算法可能是值得的，幸运的是，在其他情况下，使用累加器会得到一个几乎同样简单的线性算法。

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）尾递归以及（2）我们对每个递归调用只过一个定量的工作，因为::不需要遍历它的任何一个参数。

**尾位置的精确定义**

虽然大多数人依靠直觉来判断“哪些是尾递归”，但我们可以通过递归地定义尾部位置来更精确的说，如果一个调用处于尾部位置，那么他就是尾调用。每一种表达式的定义都有一部分，以下是这几部分：

*• 在fun f(x) = e , e 是一个尾位置。*

*• 如果一个表达式不在尾位置，那么它的任何一个子表达式也不在尾位置。*

*• 如果 if e1 then e2 else e3 在尾位置，那么e2,e3都在尾位置，但e1不在(case表达式同理)*

*• 如果let b1,....bn in e end在尾位置，那么e在尾位置(但绑定中的表达式不在)。*

*• 。。。*