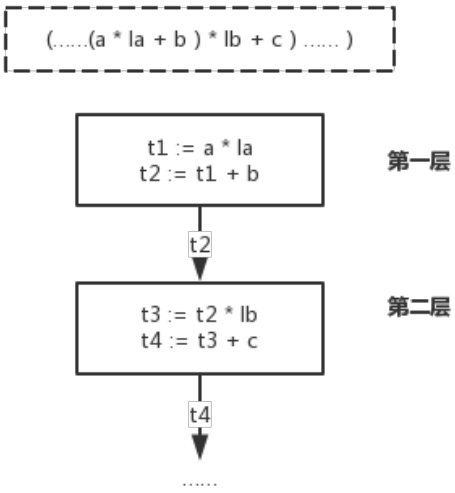
史文翰 No.2014211218 Cla.2014211304 shiwenhan@bupt.edu.cn

对于书中 P271 翻译方案优化的一些思考

一、 不做优化

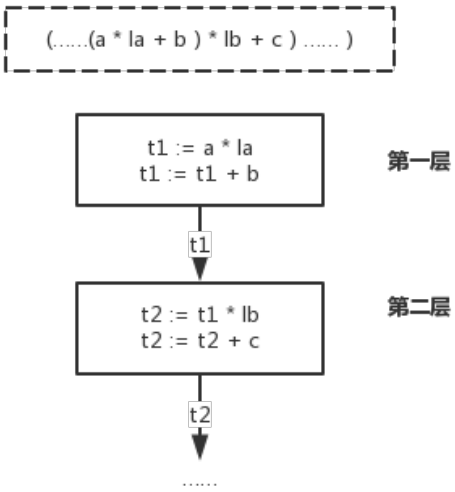
显然，如果不做优化，我们需要另外一个中间变量表达乘法结果。
三地址表达式可以如下表述（假设 la、lb……为乘积因子，a，b，c……自身为加法因子）：



在每一层递归后，我们引入了两个中间变量，如果递归 k 层，一共引入 2k 个中间变量。

二、 书上方式优化

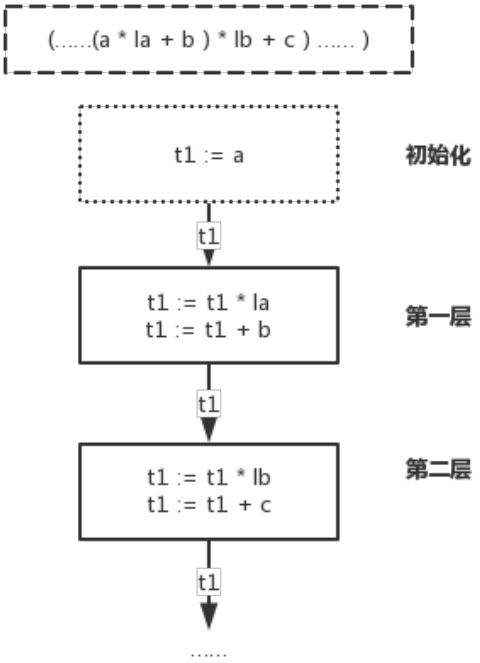
书上的方式省略了作为加法结果的中间变量，更改后的递归过程可以如下表述：



在每一层递归后，我们仅引入一个中间变量，如果递归 k 层，一共引入 k 个中间变量。

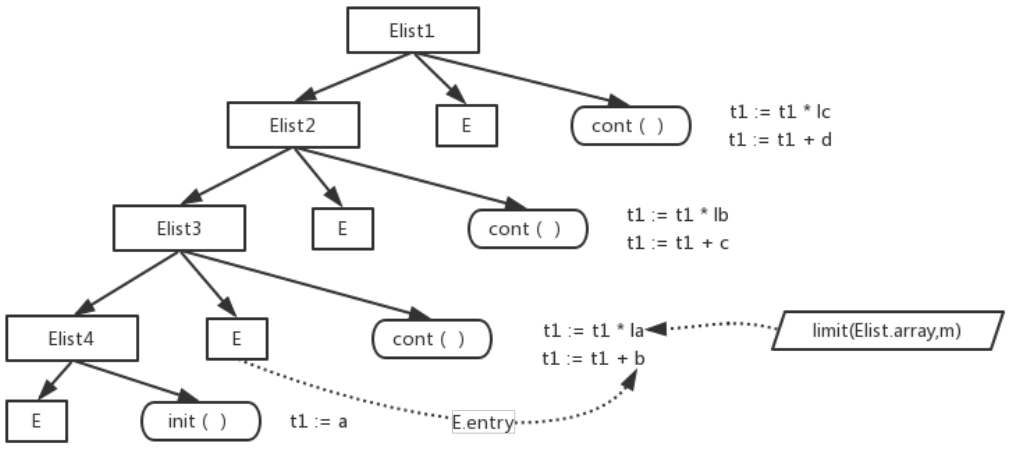
三、 左递归优化

注意到产生式（5）具有左递归的特点，（4）式作为自底向上分析中的左递归的入口，也就是从（4）式开始不断向上一层一层地递归，达到的效果可用下图表示：



在这里，我们仅考虑求解数组变址，也就是仅涉及到[]中的内容，显然变址的传递是综合属性的。当然，**其他属性如 array 的传递方式与书上相同**，保持不变。

我们重新定义（4）和（5）的翻译方案，**仅考虑 entry 属性**（略去其他属性有关的翻译方案），考虑三层以上的递归（三维以上的数组），可作分析树（省略不必要信息）如下：



我们引入两个翻译过程函数，一个是 init（），另一个是 cont（）。前者作为 t1 的产生函数，后者为 t1 的传递函数，它们的具体定义如下：

```
# Elist -> E
def init():
    t = newTemp() # 在左递归入口处声明递归中间变量t1
    Elist.ndim = 1
    Elist.entry = t # 向根节点Elist传递t1
    outcode(t ':=' E.entry) # 输出 t1 := a

# Elist -> Elist1,E
def cont():
    m = Elist1.ndim + 1
    # 直接利用Elist1输出乘法表达式，limit函数提供 t1 := t1 * p的p
    outcode(Elist1.entry ':=' Elist1.entry '*' limit(Elist.array, m))
    # 直接利用Elist1输出加法表达式，E.entry提供 t1 := t1 + k的k
    outcode(Elist1.entry ':=' Elist1.entry '+' E.entry)
    Elist.entry = Elist1.entry # 继续向根节点传递t1
    Elist.ndim = m
```

每一层虚拟综合属性的右侧为输出的三地址表达式，从分析树中不难看出，即使层数不断增多，其中间变量一直维持 t1 一个中间变量就可以完成数组变址的求解工作。在每一层，三地址表达式分为加法式和乘法式。乘法式的乘法因子由 limit 输出（当然，此时 array 信息已经传递完毕，作为 limit 函数的参数输入），加法式的加法因子由同层的 E.entry 提供（由下层向上传递，下层的细节被本层的 E.entry 属性屏蔽）。

值得注意的是，我们移走了 cont（）原来的 t = newTemp，并且直接利用 Elist1.entry 作为信息传递，从编程角度来讲，这是移除了递归函数 cont（）中的临时变量创建过程。其次，可以从文法中看出，利用 Elist -> E 这个产生式作为递归入口并创建 t1 是合适的（它在一定程度上暗示了 Elist 递归的开始）。如果选择 E->L 或者 L->id 都不是最佳的创建时机，而且可能直接引入其他的冗余信息（例如，只声明一个简单变量，却引入了一个 t1）。