构造健壮的系统的要点:对小变化不敏感.系统应该是连续的,按层次来逼近待解决的问题,这样就不会在需要改需求的时候重写很多东西,这个思想很重要.

混淆程序和过程?

为什么积分比求导难得多?

求导: ARROW(箭头)方向,父表达式->子表达式,从左自右的 '归约',所以比较简单.

积分: 从右到左,要考虑多得多,是一种复杂的搜索.

约定: 谓词名称以问号结尾.

|  |
| --- |
| # 对EXP求VAR的积分  DERIV(EXP VAR):  switch (GET\_EXP\_TYPE(EXP)):  case CONSTANT: # dc/dy  ret 0  case SAME-VAR: # 即dx/dy  ret 1  case SUM: # EXP = EXP.CAR + EXP.CDR  ret ADD(DERIV(EXP.CAR , VAR), DERIV(EXP.CDR , VAR))  case PRODUCT: # EXP = EXP.CAR \* EXP.CDR  ret ADD(PRODUCT(DERIV(EXP.CAR , VAR) , EXP.CDR) , PRODUCT(DERIV(EXP.CDR , VAR) , EXP.CAR))  # 这个例子封装了求导的三个法则,用到了按愿望思维,还没定义EXP该如何表示  # 双关(pun)思想:想表达类似加减乘除的东西,为什么不用和程序一样的语言?  # 算式需要parse,但是lisp可以提供,每个算式就是一个list,每个car就是算符,cdr是运算数  # 就这么不断取car,cdr......  # 现在定义判断是否为常数的过程:  CONSTANT?(EXP VAR):  ret ATOM?(EXP) & ~EQ?(EXP, VAR)  # 这个ATOM?用于判断表达式是否为原子表达式  # 原子表达式,不可再被细分,是系统中内建的特殊Test,无法取CAR,CDR,不是list  # 用不能被分解的符号表示VAR  # 现在定义判断是变量是否与表达式相同的过程:  CONSTANT?(EXP VAR):  ret ATOM?(EXP) & EQ?(EXP, VAR)  # EQ?也是系统内建的,不要去关心他们的实现!  # 现在定义判断一个EXP是否为和式的过程:  ADD?(EXP):  ret ~ATOM?(EXP) & EQ?(EXP.CAR , ‘+)  # 为什么要在+符号这里用’号?  # 例子: I.SPEAK.ASK(SOMEONE, ‘Say your name !’)  # Returns SOMEONE.NAME  # or  # Returns ‘Say your name !’  # 这个符号是问它(解释器)是不是代表+这个基本过程(运算符)的符号  # 20:00讲了很多关于”引用”的东西 |

|  |
| --- |
| # 现在要定义一个把两个数加起来的过程:  # 先对A1和A2做定义:  def A1 = CADR # 取出某list的CDR的CAR部分,这里讲了为什么保留了CAR和CDR  def A2 = CADDR #取出某list的CDR的CDR的CAR部分  # 比如(+ 1 2), 这其实是一个list,CAR = ‘+ , CDR=(1, (2, nil)), CADR=1, CADDR=2  ADD(A1 , A2):  ret (‘+ , A1 , A2)  # 有了ADD?和ADD,那么就照这两个葫芦画另两个葫芦PRODUCT?和PRODUCT:  PRODUCT?(EXP):  ret ~ATOM?(EXP) & EQ?(EXP.CAR , ‘\*) # 先判断不是原子式, 再看算符是不是乘  def M1 = CADR  def M2 = CADDR  PRODUCT(M1 , M2):  ret (‘ \*, M1 , M2)  # 好啦!现在可以求导啦!  # 接下来就调用DERIV来求导:  def foo = (+ (\* a (\* x x))  (+ (\* b x)  c  )  )  CALL DERIV(foo , ‘x)  # 如果不加引号的话解释器就会引用定义好的过程: +  # 加引号的话就只是一个符号而已, 避免解释器去调用相同名字的过程  # 这个CALL返回的表达式非常糟糕, 没有化简, 只是简单的代换, 所以要像分数那样化简  # 过程的形状,是通过局部规则向低层次展开,这里的返回表达式很好的证明了这一点,过程代表了一系列用于演进过程局部规则,最终会得出答案,答案是遍历树得出来的  # 从现在开始不要简单的把符号和左右值组合成list, 要化简  ADD-REDUCE(A1 , A2):  # 判断A1和A2是否是数字,可以进行合并否, 以下是网课中的代码  ###  if (NUMBER?(A1) & (A1 = 0)):  ret A2  elif (NUMBER?(A2) & (A2 = 0)):  ret A1  elif (NUMBER?(A1) & NUMBER?(A2)):  ret A1 + A2  ##  # 我认为其实只需要一个判断分支就够了:  if (NUMBER?(A1) & NUMBER?(A2)):  ret A1 + A2  else:  ret (‘+ , A1 , A2)  # 乘法的化简类似如此:  PRODUCT-REDUCE(M1 , M2):  if (NUMBER?(M1) & NUMBER?(M2)):  ret M1 \* M2  else:  ret (‘\* , M1 , M2)  # 讲这个例子是为了理解, 如何不用具体的语法,而用抽象的语法抽象的进行编程, 使用抽象屏障控制构造这些表达式  # 但是真正的奥义没那么简单, 而是在操作这些表达式的时候都是同一种表达式  # 有了这个能力之后,可以把它抽象化,隔离这层,然后再用它来写更厉害的另一层  # “可以写一种完全不同的语言, 而其实际上是由另一个语言解释并运行的” |

引用: 组织表达式被求值, 就是表达式它自己, 而不是什么别的被求过的表达式