Lisp

时间限制： 1.0 秒

空间限制： 128 MB

相关文件： 题目目录

题目背景

Lisp 语言由 John McCarthy 教授（1927-2011，1971 年图灵奖得主）发明于 1958 年，是现今还在使用的第二古老的高级语言（最古老的是 Fortran，比 Lisp 发明还早一年）。Lisp 的语法与常见的编程语言很不一样，Lisp 程序由很多列表组成，Lisp 也得名于列表处理器（**List Processor**）。

本题的任务是编写一个列表处理器，实现 Lisp 语言的基本功能（当然你可以把它看作是 Lisp 语言的一个简化版本）。输入是一段 Lisp 程序，输出是这个程序的运行结果。下面将详细的介绍需要实现的功能。

题目描述

Lisp 是一种函数式编程语言，每一个表达式都可以计算出一个结果。Lisp 表达式有两种形式——原子或列表。所谓原子，简单的讲就是一个字符串，可以使用的字符有大小写英文字母、数字和 `+-*/!?=<>_` ；而列表则是由若干个表达式和一对括号在两侧括起构成。

原子的例子：

```
12
+
John
Burger
```

上面给出了一些原子的例子。 `12` 就是一个整数，就像其他编程语言中的 `int` 一样，但后面三个却略有不同。在 Lisp 中，这些含有非数字字符的原子被称作标识符。实际上每一个标识符都指代了一个值，不过有些标识符对应的值 Lisp 已经自动绑定好了，比如 `+` 指代加法函数，而更多的标识符需要在程序中人为地指定一个值。下面则是几个列表的例子。在书写格式上，列表中的每两个表达式由一个空格分隔，左右再加一对圆括号括起。

列表的例子：

```
(f a b c)
(define x (+ 2 3))
(+ 1 1)
```

在 Lisp 中，函数的使用是通过列表来完成的。通常使用圆括号时，会把列表中的第一个表达式的值默认为是一个函数，后面表达式的值则被视为传入该函数的参数，而函数的返回值就是整个表达式的值。所以 `(+ 1 2)` 就代表了 1 和 2 做加法运算，该表达式的结果显然为 3 。

```
(+ 1 2)        //表示1和2做加法运算，结果为3。
(+ (* 5 2) 3)    //先计算5乘2，结果为10；再与3相加，最终结果为13。
```

在本题中，基本的函数有四个：加减乘除（整除），分别用标识符表示 `+` 、 `-` 、 `*` 、 `/` 指代。而我们要处理的数字（包括运算的中间结果），也都是小于等于 10^9 的自然数，且保证整除运算第二个参数不为 0 。我们的列表处理器同样支持布尔类型，分别用标识符 `True` 和 `False` 指代逻辑值真和假。此外还有一些 Lisp 预先定义好的函数，我们将在下面逐一介绍。

eq?

判断两个整数或两个逻辑值是否相等。如果相等返回 `True` ，否则返回 `False` 。 两个参数的值 或者同为整数，或者同为逻辑值。

```
(eq? 1 1)        //True
(eq? 4 (+ 1 2))        //False
(eq? (eq? 1 2) False)     //True
```

define

这是唯一一个没有返回值的函数，所以它不能被嵌套在列表之中。它的作用是为标识符绑定一个值。要求每个标识符只能被绑定一次，Lisp 自带的标识符相当于已经被绑定过一次，所以不能再成为 `define` 函数的第一个参数。

```
(define a 5)        //a的值绑定为5
(define add +)        //add的值绑定为加法函数
```

lambda

返回一个新定义的函数。 第一个参数是一个由若干个（至少一个）标识符构成的列表，表示新函数的参数列表，这些标识符做为新函数的参数仅在第二个参数中有效。需要注意的是，这里虽然也使用了列表的形式，但无需进行函数运算。

第二个参数表示新函数的返回值。在第二个参数中使用的标识符有两种情况，一种是通过 `define` 定义的标识符，一种是做为新函数参数的标识符。因为在定义函数时并不需要进行具体计算，我们要做的只是把它先存储下来，所以在第二个参数中可以使用暂时还没有绑定过值的标识符，只要保证在使用该函数进行计算时每个标识符都已经绑定了值即可。另一方面， `lambda` 表达式可以嵌套使用，这就导致了在第二个参数中使用的标识符也可能是外层函数的参数。为避免歧义，Lisp对标识符的取值采取就近原则，即优先解释为较近层的函数参数；如果在任何一层的参数列表中都找不到这个标识符，就采用 `define` 表达式绑定的值。

此外，一个单独的 `lambda` 表达式没有任何意义。在Lisp中，一个值为函数的表达式一定被嵌套在列表之中：或者在 `define` 函数中使用，或者做为一个匿名函数直接参与运算。

```
(define add3 (lambda (x y z) (+ (+ x y) z)))    //定义了一个将三个参数相加的函数add3;
((lambda (+ -) (* + -)) 2 3)            //这里用匿名函数的形式定义了乘法函数，然后计算2乘3;
(add3 2 3 4)                  //使用刚刚定义的add3函数，结果为9。
```

cond

选择函数，参数个数不定但至少为 2 ，其中每个参数都是一个由两个表达式构成的列表，且第一个表达式的值一定是逻辑类型（真或假）。类似地，这里的列表也不起函数计算作用。

`cond` 函数会依次检查每个参数的第一个表达式，如果值为真，则将第二个表达式的值返回，并且不再继续检查后面的参数。

```
(cond ((eq? a 2) 0) ((eq? a 3) 1) (True 2))    //a等于2时返回0，等于3时返回1，其它情况返回2。
```

保证每个 `cond` 函数至少存在至少一个参数，其第一个表达式为真。

输入格式

从标准输入读入数据。

一段 Lisp 程序，其中每行是一个表达式。

保证程序格式正确无误，运行时不会出现任何异常（每个函数都一定会有返回值、进行计算时不会遇到无法解释的标识符等等）。

输出格式

输出到标准输出。

对于 Lisp 程序中每一行的表达式，相应输出一行。

如果该表达式使用了 `define` 函数则输出 `define` ，否则输出该表达式的值。

样例1输入

```
(define y 10)
(define f (lambda (x y) (+ x ((lambda (x) (* x y)) y))))
(f 1 2)
y
```

样例1输出

```
define
define
5
10
```

样例1解释

定义了一个二元函数：$f(x,y)=x+y\times y$ 。

样例2输入

```
(define y 10)
(define sqr+y (lambda (x) (+ y (* x x))))
(define f (lambda (x y) (sqr+y x)))
(sqr+y 5)
(f 5 1)
```

样例2输出

```
define
define
define
35
35
```

样例2解释

`f` 函数的参数 `y` 并不能在 `sqr+y` 函数中起作用，即使 `f` 调用了 `sqr+y` 。

样例3输入

```
(define fact (lambda (n) (cond ((eq? n 1) 1) (True (* n (fact (- n 1)))))))
(fact 1)
```

```
(fact 5)
(fact 10)
(define sum (lambda (n) (cond ((eq? n 1) 1) (True (+ n (sum (- n 1)))))))
(sum 50)
```

样例3输出

```
define
1
120
3628800
define
1275
```

样例4输入

```
(define fun1 (lambda (x) (cond ((eq? x 0) 1) (True (fun2 (- x 1))))))
(define fun2 (lambda (x) (cond ((eq? x 0) 2) (True (fun1 (/ x 2))))))
(fun1 2)
(fun2 2)
(fun1 5)
(fun2 5)
```

样例4输出

```
define
define
1
2
1
1
```

样例 5 输入

```
(define 33g (lambda (x1 x2 x3 x4) ((lambda (x1 x2) ((lambda (x1 x2 x3) (+ x4 (+ x1 ((lambda (x4
 x1) (+ (* x4 x3) (* x1 x1))) x2 x3)))) x2 x3 0alpha)) x1 x3)))
(define 0alpha 666)
(33g 1 2 3 4)
(33g 2 4 6 8)
(33g 0 0 0 0)
(define 666kk (lambda (x1) (lambda (x2 x3) (+ x1 (+ x2 x3)))))
((666kk 10) 20 30)
```

样例 5 输出

```
define
define
445561
447566
```

```
443556
define
60
```

提示

针对题意有些疑问的地方，我们统一在这里做出进一步的说明。

在本题中，一个表达式的值有且仅有三种可能：整数，逻辑值（`True` 或 `False`）和函数，传给函数的参数和函数返回值的类型可能是其中的任意一种。对于Lisp内置的函数，加减乘除函数的参数只能是整数，`eq?` 和 `cond` 等的参数要求如题面所述。

用 `lambda` 表达式定义函数 `f1` 时，如果在参数一中定义了标识符 `x`，那么这个 `x` 仅在后面表示返回值的参数二中起作用。如果参数二调用了在别的地方定义的函数 `f2`，我们认为参数二只是使用了那个函数的函数名 `f2`，函数 `f2` 的返回值部分不会被算做在 `f1` 的参数二里，自然 `x` 就不会在 `f2` 中生效了。换言之，作用范围只考虑定义时书写上的关系，而不考虑运行时的调用关系。样例 2 恰好说明了这个问题。另一个类似的例子：在C语言中，即使 `a` 函数调用了 `b` 函数，`a` 中定义的变量同样也不能在 `b` 中起作用。

子任务

前 30% 的数据，没有 `lambda` 函数；

前 60% 的数据，`lambda` 函数不会嵌套出现；

对于 100% 的数据，输入程序的行数小于等于 200，每一行不多于 200 个字符（换行符不计），且函数调用时深度不会超过 50。调用 Lisp 自带函数和匿名定义的函数不计入函数调用深度，样例3中计算 `sum(50)` 时函数调用深度恰好为 50。

递交历史

#	状态	时间
No data available in table		