

Potato 语言简述

Potato 是一种脚本语言，他的命令十分简单，专门为本次比赛和以后比赛的类似题型而开发。

下面我们介绍 Potato 语言。

类型系统

Potato 只支持一种类型，那就是 **32 位无符号整形**，也就是 C++ 语言中的 `unsigned int` 类型，它无法支持其他任何类型，包括浮点数，64 位整数，结构体，联合体，枚举类型等等。

算术系统

在 Potato 中，我们只支持基本的算术运算：`+`，`-`，`*`，`/`，`%`，`&`，`|`，`^`，`<<`，`>>`，`<`，`>`，`<=`，`>=`，`==`，`!=`，`&&`，`||`

下面我们介绍这些运算的规则和注意事项。

- `+`，`-`，`*`，`/`，`%`，`&`，`|`，`^` 和 C++ 中 **32 位无符号整形** 的运算规则相同。其中 `/` 运算(除法运算)必须保证除数不是 0，不然解释器会报错。
- `<<`，`>>` 的运算规则与 C++ 中无符号 **32 位整数** 的左移和右移运算相同。但是必须保证移动的位数必须在 `[0, 31]` 内，否则解释器报错。
- `<`，`>`，`<=`，`>=`，`==`，`!=`，`&&`，`||` 运算规则与 C++ 语言中对应的逻辑运算相同。如果为真，那么结果是 1，否则结果是 0。

常量

我们定义常数为 0 到 $2^{32} - 1$ 之间的数字，由无前导零的十进制直接表出。比如 `0`，`10`，`4294967295` 都是常数，而 `00000`，`1 + 2`，`1000000000000000`，`-1` 都不是常数。

变量与存储系统

Potato 不支持用户任何的声明变量或者数组的行为，但是 Potato 自带了：

- 9 个 **32 位无符号整形** 变量 `i`，`j`，`k`，`p`，`q`，`t`，`x`，`n`，`m`。
- 变量 `n`，`m` 在 potato 开始被解释之前就存储了一些关于读入的信息。
- 一个大小为 10^5 的 **32 位无符号整形** 数组 `a[0...105-1]`
- 一个大小为 10 的 **32 位无符号整形** 数组 `result[0...9]`

有下面这些注意事项：

- 如果我们要取出数组中的一个元素，我们只能用下面两种形式(假设我们要对 `a` 数组进行读写)：
 - `a[const]`，其中 `const` 是一个 $0 \sim$ 数组长度-1 的常量。不能进行越界的访问，比如 `a[-1]`，`a[1000000000]`。
 - `a[i]`，`a[j]`，`a[k]` 来表示数组中的一个元素。注意，我们不能用 `x`，`n`，`m`，`p`，`q`，`t` 这些变量来访问数组。
 - 不可以对数组访问的下标进行任何的计算。比如 `a[i + j]` 是不合法的。
 - 当 `i`，`j`，`k` 的值不在合法的范围内，并且通过 `a[i]`，`a[j]`，`a[k]` 这样的语句访问数组中的元素时，会报错并且终止程序。
 - 不要多加空格，比如 `a[10000]`，`a [1000]`，`a [233]`，`a[j]` 都是不合法的。
- `x`，`n`，`m` 是三个特殊的变量，`result` 是一个特殊的数组，他们的特殊性我们将在之后说明。

语句集合

一份正确的 Potato 代码由若干行组成，每行恰好包含一个语句，每个语句只能是下面列举的这几种之一：

- `assign a b`: 将 b 的值赋值给 a , 这里 a 和 b 可以是变量, 也可以是数组中的一个元素。
- `calc a b op c`: 这里 a, b 和 c 可以是变量, 也可以是一个数组中的一个元素。这条语句计算 $b \text{ op } c$ 的结果, 并将结果赋值给 a , 其中 op 是算术系统中提及的一个运算符。

这里我们说明一下特殊的变量 x, n, m 和特殊数组 $result$ 的特殊之处:

- x, n, m 不能通过 `assign` 和 `calc` 进行赋值, 否则解释器报错。
- $result$ 数组中的每个位置的值只能被修改一次, 但可以读无数次。
- `read`: 这条指令从输入流中读入一个数字, 并将结果赋值给变量 x 。
- `label L`: 其中 L 是一个字符串。这条指令会设置一个名字叫做 L 的语句标号, 语句标号的作用我们后面会讲。
 - L 必须是一个长度不超过 20 的字符串。
 - 不同的语句标号的名字必须两两不同。
- `jump x`: 其中 x 是一个字符串, 代表一个语句标号的名字。执行 `jump` 语句会使下一条待执行的语句变成语句标号 x 所在的那一行的 `label` 语句 (具体见执行方式)。
- `cjump x`: 其中 x 是一个字符串, 代表一个语句标号的名字。这条指令会判断变量 t 的值是否是 0, 如果不是 0 则执行 `jump x`, 否则接着往下执行 (具体见执行方式)。
- `end`: 这个语句放在整段代码的末尾, 表示代码结束, `end` 后面的代码不会被执行。

执行方式

Potato 语言有一个额外的全局变量 $curline$ 来维护下一条执行的语句在哪一行, 初始 $curline = 1$ 。

每次执行完一个语句后, 如果当前执行的语句不是跳转语句(`jump`)与条件跳转语句(`cjump`), 那么 $curline \leftarrow curline + 1$ 。如果当前执行的是跳转语句(`jump`), 那么 $curline$ 会变成语句标号 x 所在的那行的行号。

如果当前执行的是条件跳转语句(`cjump`), 那么 $curline$ 会进行判断, 然后选择是 +1 还是跳转。

一个Potato代码例子

这份代码读入 n 个整数(整数的个数存在变量 n 中), 他统计这 n 个整数中不超过 lim 的数字的最大值(lim 存在入变量 m 中), 最后将结果存入 $result[0]$ 中。

```

1  assign p n
2  assign k 0
3  label loop
4  read
5  calc t x > m
6  cjump skip
7  calc t x <= k
8  cjump skip
9  assign k x
10 label skip
11 calc p p - 1
12 assign t p
13 cjump loop
14 assign result[0] k
15 end

```

解释器

我们提供了一份 Potato 的解释器 `potato.cpp` 作为下发文件，供选手们使用

首先，你要将该 `potato.cpp` 编译

```
1 | g++ potato.cpp -o potato -O2 -std=gnu++11 -static
```

之后，假设你本地的一份用 Potato 输写的代码是 `a.po`

如果你是 windows 系统：

```
1 | potato a.po
```

如果你是 Linux/Ubuntu 系统：

```
1 | ./potato a.po
```

之后解释器会从标准输入中读入 $n + 2$ 个数，前两个数为 n, m ，分别为 Potato 的变量 n, m 的初始值，之后 n 个数表示将用 read 读入的数据。

例如之前的 Potato 代码，我们将他存为 `a.po`

在Linux环境下运行：

```
1 | ./potato a.po
```

然后输入

```
1 | 5 10
2 | 1 2 5 11 114514
```

得到输出：

```
1 | Cell Used: 0
2 | Instruction Used: 47
3 | Result : 5 0 0 0 0 0 0 0 0 0
4 | ok program end
```

解释器已知的一些问题

由于解释器是一个字符串一个字符串读入的，所以如果你在同一行写一些多余的词，他们会被算到其他行去

例如，下面的代码

```
1 | read
2 | read x
```

解释器会输出：

```
1 | wrong answer syntax error in 3-th line
2 |
```

因为解释器读入 read 后，认为已经读完了第二行