Language +

这个语言是用于基础入门的练习语言，旨在尽可能少的引入细节的前提下，让编程初学者了解机器执行的过程和程序编译的过程。出于一些特殊的考虑，这个语言的后缀名应该为 `*.pl`，即 `plus language`。

该语言仅包括赋值、加法运算和输出语句。

文件

该仓库包括两个部分，一个是测试，另外的 `plus.c` 是完整实现（包括高级主题），可以通过 `python test.py --stage=1`，来指定运行第一部分的测试，等等的。

形式化表示

形式化定义如下，我们**约定**一下符号：

- `'A'` 表示字符 `A`
- `'A'-'Z'` 表示字符 `A` 到字符 `Z` 中的任意一个字符，例如：`M`、`K`，注意 `n`、`m` 并不在这个表示内。
- `{ A }` 表示 `A` 出现 `0 - n` 次，方便起见，`{ A }+` 表示 `A` 出现 `1 - n` 次
- `[ A ]` 表示一个可选的 `A`，即 `A` 出现0次或者1次
- `A | B` 表示 `A` 或者 `B`
- `A B` 表示先跟 `A`，然后跟 `B`
- `->` 表示推导出，即右侧内容可以替换（reduce，归约）为左侧符号，或者左侧符号可以推导为右侧符号

在这种约定下，我们的起始符号为 `Start`，那么：

```
### 语法 ###

# 起始: 0条或者多条语句
Start -> { Statement }
# 语句:
# echo XXX或者XXX = YYY
Statement -> `echo` Express
           | Identifier `=` Express
# 表达式
Express -> Express `+` Value
         | Value

### 词法 ###

# 值，可以是任意一个整数或者标识符
Value -> { Integer }+
       | Identifier

# 标识符，以`A-Z|a-z|_`开始，后可以跟数字或者字母或者下划线
Identifier -> Alphabet { Alphabat | Integer }

# 整数，0 - 9的任意字符
Integer -> `0` - `9`
```

```
# 字母表，任何一个字母，或者下划线
AlphaBet -> `A` - `Z`
          | `a` - `z`
          | `_`
```

语言所有的值类型均为**整数类型**。除此之外，一个**变量名不应该超过255个字符**。

目标

任务在于编写一个**读取文件**，并输出该文件**执行结果**的语言解释器，同时，你的解释器应该**拒绝掉所有不合法输入**，你可以考虑下面的步骤：

1. 编写文件读写程序
2. 编写一个符号表，它能根据**符号名**，**查找**返回符号是否存在，以及它的值，并能够**插入**新的符号
3. 编写一个词法分析器，它能将文件输入**拆分**为一个个的 `Token`，并**拒绝掉所有非正确**的 `Token`，并给出**报错提示**。
4. 编写一个语法分析器，它能**检查语法**，并转换为一个**合适的表示**，并拒绝掉**不合理的组合**，并给出**报错提示**。
5. 编写一个解释执行程序，它能**执行代码**，注意我们排除掉了所有非正确的程序，所以这里**不需要再检查**。

在基础任务之外，我们准备了**高级主题**，高级主题为**可选**内容：

- 执行：请考虑你的解释执行程序，尝试优化掉其中的**符号查找**，将其转变为**数组下标访问**。请尝试验证新的版本，并尝试测量其执行性能。
- 报错：请优化你的报错提示，让其包括**行号**、**错误位置**，并用优雅的方式把它打印输出出来。
- 排版：请尝试把你的语法分析结果重新输出，将其变成一个**优美漂亮**排版的代码。

可行的代码结构

这里给出一个基本的思想，用于参考编写代码：

总体结构

```c
int main()
{
    int err;
    size_t token_list_length;
    size_t exec_code_length;
    char *buff = NULL;
    // token
    Token *token_list = NULL;
    // 语法分析器输出，用于解释执行，比如可以保存 struct { 指令; 原操作数; 目标操作数; } 等等
    IR *exec_code = NULL;
    // step1. 读取文件
    err = 读取文件(&buff);
    if (err != 0)
    {
        goto fail;
    }
    // step2. 词法分析
    // 注意的是，C语言的数组返回值是不包括长度的（它是一个指针）
    // 因此这里我们还记录token_list_length，即返回的token_list的长度
    // 下面也是一样的
    err = 词法分析(&token_list, &token_list_length);
```

```
    if (err != 0)
    {
        goto fail;
    }
    // step3. 语法分析和制导翻译
    err = 语法分析_制导转换(token_list, token_list_length, &exec_code,
&exec_code_length);
    if (err != 0)
    {
        goto fail;
    }
    // step4. 解释执行
    err = 执行(exec_code, exec_code_length);
    if (err != 0)
    {
        goto fail;
    }
fail:
    return 0;
}
```

示例

方便理解，我们给出几个例子，以及对应的执行结果：

变量赋值

```
a = 1
b = a
c = b
d = c
echo d
```

执行结果:

```
1
```

不使用的变量

```
a = 1
b = 2
c = 3
echo a
```

执行结果：

```
1
```

表达式

```
a = 1
b = 1
c = a + b
d = b + c
echo a
echo b
echo c
echo d
```

执行结果：

```
1
1
2
3
```

混乱的排版

注意，语言**不强制以空格、换行符区分语句和标识符**，等等。

```
a = 1
  b = 2   c = 3  d = a
     + b + c
        echo d
```

上面的程序好好排版后是这样的：

```
a = 1
b = 2
c = 3
d = a + b + c
echo d
```

执行结果：

```
6
```

不合法的输入：未知的token

```
a@ = 1
echo a
```

注意到`@`不是任何一个合法的字母，因此**词法分析器**应该拒绝掉该输入，并给出**报错。**

可行的错误提示可以看起来是这样的：

```
Error: Unrecognized token `@`
```

在高级主题中，你可以尝试打印这样的错误提示：

```
1  a@ = 1
     ^
Error: Unrecognized token `@`
```

语法错误：缺失标识符

```
= 1
```

注意到赋值符号左侧缺失了一个标识符，**语法分析器**应该拒绝本输入，并给出**报错。**

可行的错误提示可以看起来是这样的：

```
Error: Missing an identifier after assign operator `=`
```

在高级主题中，你可以尝试打印这样的错误提示：

```
1  = 1
  ^
Error: Missing an identifier after assign operator `=`
```

参考

下面的内容**难度依次递增**，但是哪怕是第一项也可能存在无法理解的情况，因此**已经很努力了，但是也很难理解的情况下，一定要来询问。**

字符串表

字符串表可以是下面的内容：

- 哈希表
- 字典树
- 平衡二叉树（AVL树、AA树、红黑树、etc.）
- 平衡树（B树）

读文件

- fopen、fread
- CreateFile、ReadFile

词法分析

- DFA
- NFA
- AC自动机

语法分析

- 最左推导、最右推导、自顶向下分析、自底向上分析
- LL(k)文法
- 递归下降分析器
- LR(k)文法