

波蘭表示法

波蘭表示法是一種算數的表示法，其特點就是將運算子放在運算元的前面，如：

$3 + 4$ 以波蘭表示法表示則會變成 $+ 3 4$

因此波蘭表示法又稱為前綴表示法，其優勢為不必與我們平常習慣的中綴表示法一樣依靠括號來表示運算順序，如以下例子：

$(3 + 4) * 2 \rightarrow * + 3 4 2$

$(3 + 4) * 5 - 6 \rightarrow -*+ 3 4 5 6$

這種作法廣泛的運用在 Lisp 語系的 S-exp 中，如以下 Common Lisp 例子

```
( defvar foo ( - ( * ( + 3 4 ) 5 ) 6 ) )  
  
( print foo )
```

等價於：

```
int foo = ( 3 + 4 ) * 5 - 6 ;  
  
printf( "%d", foo ) ;
```

現在你需要實作一個基於前綴方式的計算器，其主要邏輯如下：

```
Print ' Welcome use our calculator!'

repeat
  Print '> '
  token_list = ReadFormula();

  if no error
    then result <- EvalFormula( token-list );
    if error
      PrintErrorMessage() ;
    else
      print result ;
  else
    PrintErrorMessage() ;

until END-OF-FILE encountered
Print 'ByeBye~'
```

換言之，你的程式理論上會有三個主要的 function：

1. ReadFormula()

此 function 的工作就是將 user input 依照 terminal 規則切開，並檢查是否有非法的 token 存在，若沒有則將該行所有的 token 傳回

我們定義在本次的題目中，所有的 formula 皆為 line-enter 或者 End-of-file 結束，也就是說 formula 最多就是一行，不會有以下的情形發生：

+ 3

4

terminal 規範如下：

OPERATOR	['+' '\-' '*' '/']
INT	[+-]?[0-9]+
LINEENTER	\n
WHITESPACE	[\s]*

Note:

Token 與 token 之間不一定會有 white-space 分隔，也就是說 input 有可能為：

+ * 3 4 / 6 3
或者
+*3 4/6 3

兩者運算結果皆為 14，你的程式應該要能夠正確的依照 terminal 規範切割，而非完全的仰賴 white-space

另外，關於不合法的字串，題目刻意會要求你們用一個反常的方式處理 **(所以請一定要認真閱讀題目!)**

我們規定：

如果有一 token 原本合法被判定為 INT，但因為後來出現的某字元而變為非法，那錯誤的輸出必須是整個 token，而非只有非法的部分

如果有一 token 從一開始就是非法，那此 token 的分割字元就只有 white-space

Ex:

```
> +0a
Error: Unknown token +0a
//注意不是 Error: Unknown token a, +0 == 0 == -0

> -0a
Error: Unknown token -0a

> *0a
Error: Unknown token 0a

> /0a
Error: Unknown token 0a

> /a0
Error: Unknown token a0

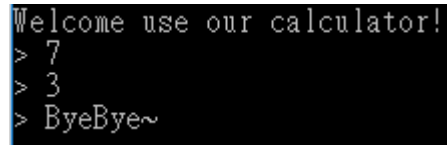
> +-a
Error: Unknown token a

> (=32? Hello world!!
Error: Unknown token (=32?
```

測資不保證不會是全空，也不保證會有 line enter 結尾，因此：

```
+ 3 4
- 9 6\n
```

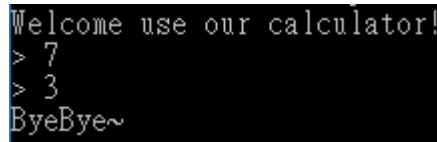
的結果為：



```
Welcome use our calculator!
> 7
> 3
> ByeBye~
```

```
+ 3 4
- 9 6<EOF>
```

則會輸出：



```
Welcome use our calculator!
> 7
> 3
ByeBye~
```

以上兩筆測資皆有可能出現，你的程式必須要能夠正確處理

2. EvalFormula()

此 function 為本程式之核心，他會嘗試計算 ReadFormula() 所傳回之 token list 並將結果回傳

Formula 的文法如下：

```
PROG      -> TOPEXPS
           | λ
TOPEXPS   -> TOPEXP TOPEXPS
           | λ
TOPEXP    -> EXP LINEENTER
           | LINEENTER
EXP       -> OPERATOR WHITESPACE EXP WHITESPACE EXP
           | INT
```

3. PrintErrorMessage()

假設 ReadFormula() 或 EvalFormula() 無法順利完成工作，則會拋出錯誤訊息，本次程式總共會有三種錯誤訊息：

```
a. Error: Unknown token xxx
   // Error: (space) Unknown (space) token (space) xxx (line-
   enter)
```

此錯誤由 `ReadFormula()` 拋出，假設從 `user input` 讀取到了沒有被定義的輸入，`ReadFormula()` 必須立即將錯誤顯示並將有問題的 `token` 一併顯示，最後將該行剩餘的 `input` 清空

Ex :

```
> + 3 a
Error: Unknown token a
> + 3 1.2
Error: Unknown token 1.2
```

b. Error: Illegal formula!

```
// Error: (space)Illegal (space) formula! (line-enter)
```

此錯誤由 `EvalFormula()` 輸出，只要 `EvalFormula()` 發現 `ReadFormula()` 傳回的 `token list` 不是一個正確的前綴表達式，就要拋出此錯誤

Ex:

```
> + 3 6 5
Error: Illegal formula!
> 3 + 5
Error: Illegal formula!
> + 2 + 2
Error: Illegal formula!
```

c. Error: Divide by ZERO!

```
// Error: (space)Divide (space)by (space)ZERO! (line-enter)
```

此錯誤由 `EvalFormula()` 輸出，當解析到除法且除數為零時，不得回傳任何結果，必須立即拋出此錯誤

Ex:

```
> / 5 0
Error: Divide by ZERO!
```

最後，如果有一 formula 同時違反了多個規則，請依照以下之準則選出一個輸出即可：

Unknown token 的優先層級**最高**

其次是 Divide by ZERO!

最後如果沒有以上兩種錯誤 才會是 Illegal formula!

```
> / 3 0
Error: Divide by ZERO!
> / 3 0 hello
Error: Unknown token hello
> / 3 0 5
Error: Divide by ZERO!
> + 3 / 3 0 5
Error: Divide by ZERO!
> / 3 0 5
Error: Divide by ZERO!
> / 3 + 3 -3
Error: Divide by ZERO!
> /3++3-3
Error: Divide by ZERO!
> /3--3-3
Error: Divide by ZERO!
> /3*3+0
Error: Divide by ZERO!
> /3*3-0
Error: Divide by ZERO!
> / 100 + 3 / 2 0 3
Error: Divide by ZERO!
> / 100 + 3 2 / 2 0
Error: Illegal formula!
> / 100 + 3 2 5 / 2 0
Error: Illegal formula!
> / 100 + 3 2 5 - 2 hello
Error: Unknown token hello
> / 100 + / ** 2 * 2 2 * 2 2 1
Error: Illegal formula!
> / 100 + / ** 2 * 2 2 * 2 2 1 32
1
```