## Stack

Ngăn xếp (stack) là kiểu dữ liệu được tổ chức theo nguyên tắc *vào sau – ra trước (LIFO: Last In First out):* phần tử được nạp vào sau thì sẽ được lấy ra trước. Khi ta tổ chức các hoạt động sau đây sẽ thường tuân thủ theo nguyên tắc của ngăn xếp:

* Tháo các chi tiết của một sản phẩm để sửa chữa, sau đó lại lắp các chi tiết vào sản phẩm
* Xếp nguyên liệu vào các thùng đựng dạng đường ống với một đầu vào, sau đó lấy ra để sử dụng

Có thể mường tượng ngăn xếp như một đường ống st chỉ có một cửa đảm nhận chức năng vào/ra. Khi nạp một đói tượng x vào st, st sẽ được tăng thêm một ngăn (trên cùng) để chứa x, khi cần lấy khỏi st một phần tử thì st sẽ xuất ra phần tử trên cùng và do đó, số lượng sẽ được giảm đi 1.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  | 🡨 🡪 |

#### Các thao tác cơ bản trên stack

Trong Python, ngăn xếp được cài đặt qua kiểu dữ liệu danh sách (list) như sau:

* Khởi tạo stack rỗng: st = []
* Nạp đối tượng x vào stack: st.append(x): đối tượng x sẽ được nạp vào ngọn stack, dung lượng của stack sẽ được tăng thêm 1 đơn vị.
* Xem ngọn stack: st[-1]
* Lấy đối tượng trên ngọn stack ra khỏi stact: x = st.pop(), dung lượng của stack sẽ được giảm bớt 1 đơn vị.
* Kiểm tra stck rỗng? len(st) == 0 hoặc st == []
* Hiển thị nội dung stack: print(st)

### Brackets

*Một biểu thức toán học có thể chứa các cặp ngoặc thuộc ba dạng sau* ( ), [ ] *hoặc* { }*.*

*Hãy kiểm tra tinh hợp lệ của các biểu thức có chứa các cặp ngoặc?*

Ví dụ

|  |  |
| --- | --- |
| BRACKET.INP | OUTPUT |
| a\*(b+c\*d)  a\*(b+c\*d))  a\*([b+c]\*d)  {a\*[b+c]-d}  {a\*{[b+c]\*d))  {a\*{(b+c]\*d}}  {a\*{(b+c)\*d}} | Correct  Syntax Error: Unmatched )  Correct  Correct  Syntax Error: Unmatched )  Syntax Error: Unmatched ]  Correct |

#### Understanding

Ta cần hiểu thế nào là một cặp ngoặc được bố trí hợp lệ?

Đời thường có thể hiểu mỗi cặp ngoặc gồm có nồi (dấu mở ngoặc) và vung (dáu đóng ngoặc tương ứng). Như vậy, các cặp ngoặc cùng loại có thể chứa nhau giống như một dãy nồi-vung có thể đặt lồng nhau.

#### Algorithm

A green and white container

Description automatically generatedTa thể hiện dãy nồi-vung dưới dạng một stack st. Gặp dấu mở ngoặc (nồi) ta nạp vào st để chờ sau này sẽ gặp vung tương ứng. Gặp dấu đóng ngoặc (vung) ta xét ngọn st có chứa nồi cùng loại hay không. Nếu có thì ta lấy nồi đó ra để thu được một cặp nồi-vung cùng loại. Dễ thấy hai tình huống không hợp lệ khi gặp vung (đóng ngoặc) sau đây:

* st rỗng: có vung mà không có nồi cùng loại trong st
* ngăn trên của st không chứa nồi cùng loại.

Bạn nên thêm một phần tử đệm, ví dụ SHAP = '#', vào đáy stack để tiện kiểm tra stack.

Trước khi kêt thúc chương trình, bạn kiểm tra lại stack. Nếu st còn chứa các phần tử khác SHAP tức là còn có những dấu mở ngoặc chưa gặp các dấu đóng ngoặc tương ứng.

#### Program

# Brackets

OPEN = '([{'

CLOSE = ')]}'

SHAP = '#'

# c là dấu mở ngoặc ?

def IsOpen(c): return c in OPEN

# c là dấu đóng ngoặc ?

def IsClose(c): return c in CLOSE

# return dấu mở ngoặc tương ứng với dấu đóng ngoặc c

def GetOpen(c): return OPEN[CLOSE.index(c)]

def Run():

with open('BRACKET.INP') as f:

ss = f.read().split() # Đọc toàn bộ input

# vào dãy các string ss

# print(ss)

for s in ss: Test(s.strip())

def Test(s):

print(s, end = ': ')

st = [SHAP] # phần tử đệm

for c in s:

if IsOpen(c): st.append(c); continue

if IsClose(c):

if st[-1] == GetOpen(c): st.pop()

else: print('Syntax Error: Unmatched', c); return

continue

if st[-1] == SHAP: print('Correct'); return

while st[-1] != SHAP:

print('Syntax Error: Unmatched', st.pop())

# APPLICATION

Run()

print(" T h e E n d")

#### Output

a\*(b+c\*d): Correct

a\*(b+c\*d)): Syntax Error: Unmatched )

a\*([b+c]\*d): Correct

{a\*[b+c]-d}: Correct

{a\*{[b+c]\*d)): Syntax Error: Unmatched )

{a\*{(b+c]\*d}}: Syntax Error: Unmatched ]

{a\*{(b+c)\*d}}: Correct

T h e E n d

|  |
| --- |
| A person in a suit and tie  Description automatically generated |
| Jan Lukasiewicz  1878-1956 |

### Postfix Syntax

Năm 1924, nhà toán học Poland Lukasiewicz đề xuất phương pháp ghi các biểu thức toán học dưới dạng *dấu phép toán được đặt sau các toán hạng*, ví dụ, thay vì viết *c*×(*d*+*h*) thì ta viết *c d h* + ×

Ký pháp này có đặc điểm là không phải dùng đến các dấu ngoặc để chỉ rõ thứ tự thực hiện các phép tính trung gian. Sau khoảng gần nửa thế kỷ, khi máy tính điện tử ra đời (năm 1949), các nhà tin học phát hiện ra rằng nếu chuyển các biểu thức toán học sang dạng hậu tố thì rất tiện lợi cho việc thực hiện tính toán kết quả trên một ngăn xếp. Có lẽ, vì thế mà mọi người thường gọi ký pháp hậu tố là *ký pháp Ba Lan* nhằm tôn vinh quê hương ông.

### Tính trị của biểu thưc hậu tố

Gia sử ta có một biểu thức hậu tố dưới dạng string

pol = "c d h + \* "

Đây là dạng Poland của biểu thức chính quy c \* (d + h). Ta sẽ triến khai hàm Exe(pol) tính toán trị của biểu thức pol với các quy ước sau đây:

Mỗi biến trong pol có tên là một chữ cái viết thường trong bảng chữ cái tiếng Anh, a, b, c, ... và được nhận giá trị mặc định theo thứ tự a = 0, b = 1, ...

Hàm Exe khi đó sẽ được thực hiện trên một ngăn xếp st kiểu nguyên như sau:

Exe(pol):

Khởi tạo ngăn xếp rỗng st

for each symbol c in pol do

if c là biến then nạp trị của c vào st

else if c là phép toán @ then

thực hiện phép toán @ trên các toán hạng nằm trên ngọn của st,

đồng thời lấy các toán hạng ra khỏi ngọn st,

ghi kết quả trung gian vào ngọn của st

end if

return kết quả là giá trị trên ngọn của st

Ta minh hoạ quá trình tính trị của dạng hậu tố Poland pol = 'c d h + \*' như sau:

|  |  |  |
| --- | --- | --- |
| Exe('c d h + \*') = ? | | |
|  | Khởi trị | st = [] |
| 'c' | nạp Val('c') = 2 vào st: | st = [2] |
| 'd' | nạp Val('d') = 3 vào st: | st = [2, 3] |
| 'h' | nạp Val('h') = 7 vào st: | st = [2, 3, 7] |
| '+' | thực hiện st[-2] += st[-1]: | st = [2, 10] |
| '\*' | thực hiện st[-2] \*= st[-1]: | st = [20] |
|  | return | 20 |

#### Program

# postfix expression

def IsVar(c): return 'a' <= c <= 'z'

def Val(c): return ord(c) - ord('a')

def Exe(pol):

print('Input Postfix expression:', pol)

st = []

for c in pol:

if IsVar(c): st.append(Val(c)); continue

if c == '+':

st[-2] += st[-1]

st.pop()

continue

if c == '-':

st[-2] -= st[-1]

st.pop()

continue

if c == '\*':

st[-2] \*= st[-1]

st.pop()

continue

if c == '/':

if st[-1] == 0:

print('Syntax eror: divide by zero')

exit(0)

st[-2] //= st[-1]

st.pop()

continue

if c == '%':

if st[-1] == 0:

print('Syntax eror: divide by zero')

exit(0)

st[-2] %= st[-1]

st.pop()

continue

return st[-1]

# APPLICATION

r = Exe("c d h + \*")

print(' Result:', r)

print(" T h e E n d")

#### Output

Input Postfix expression: c d h + \*

Result: 20

T h e E n d

#### Testing

*Bạn hãy mở rộng chương trình trên để có thể thực hiện các phép toán sau đây:*

* *Các phép toán 2 ngôi +, -, \*, /, %*
* *Các phép toán 1 ngôi:*
* *~x đảo dấu của x: cho ra giá trị -x*
* *x> tăng x thêm 1: cho ra giá trị x+1*
* *x< giảm x bớt 1: cho ra giá trị x-1*
* *x! tính giai thừa x! = 1\*2\*…\*n*

*Hãy chuyển đổi các biểu thức sau đây sang dạng Poland và test chương trình để kiểm tra kết quả:*

c+d\*k

d\*d - c\*d\*h + h\*h

c-d>!+k\*d

(k-c\*d)-(c+(d+b\*e))

-(e-b)

-(-e-b)

#### Đáp án

c+d\*k -> cdk\*+ : 32

d\*d-c\*d\*h+h\*h -> dd\*cd\*h\*-hh\*+ : 16

c-d>!+k\*d -> cd>!-kd\*+ : 8

(k-c\*d)-(c+(d+b\*e)) -> kcd\*-cdbe\*++- : -5

-(e-b) -> eb-~ : -3

-(-e-b) -> e~b-~ : 5

### Chuyển đổi biểu thức chính quy sang dạng Poland Postfix

Để chuyển biểu thức chính quy e sang dạng Poland chúng ta cần những quy tắc sau đây:

#### Quy tắc xây dựng biểu thức chính quy

Ta quy định các biểu thức được xây dựng từ các biến a, b, c,…. nhận các giá trị nguyên a=0, b=1,. c=2, … và các phep toán như đã trình bày ở phần trên.

#### Quy định thứ tự ưu tiên của các phép tóan

Khi tính toán giá trị một biểu thức số học ngươi ta thường quy định một trật tự ưu tiên như sau:

1. Tính trị của các biểu thức trong ngoặc
2. Tính trị các lời gọi hàm
3. Tính trị các phép toán một ngôi
4. Tính trị các phép toán hai ngôi

Như vậy, phép toán có độ ưu tiên nhỏ thì được ưu tiên hơn.

Nếu hai phép toán có cùng độ ưu tiên thì thực hiện từ trái qua phải. Trong Python, nếu gặp các phép toán lũy thừa thì thực hiện từ phải qua trái, ví dụ

2\*\*2\*\*3 = 2\*\*(2\*\*3) = 2\*\*8 = 256

Ví dụ, biểu thức

e = c - d > ! + k\*d

sẽ được thực hiện theo trật tự sau đây:

t1 = d>

t2 = t1!

t3 = c-t2

t4 = k\*d

t5 = t3+t4

Như vậy dạng Poland chính là một biểu thức thể hiện *trật tự tính toán* nói trên:

pol = c d > ! - k d \* +

Chúng ta tạm quy định thứ tự ưu tiên cho các phép toán như sau:

|  |  |
| --- | --- |
| Phép toán | Độ ưu tiên |
| >  <  !  \*  /  %  +  - | 100  100  100  200  200  200  300  300 |

#### Algorithm

Gọi Compile(e) là hàm chuyển đổi biểu thức chính quy e sang dạng Poland pol.

Ta duyệt từng kí tự c trong e và xét các tình huống:

* Nếu c là biến: đưa c ra pol
* Nếu c là phép toán @: rỡ các phép toán đã nạp trước trên stack có độ ưu tiên bằng hoặc nhỏ hơn độ ưu tiên của @ đưa ra pol. Cuối cùng nạp phép toan @ vào stack để chờ xử lý sau.
* Nếu c là dấu mở ngoặc '(': đưa c lên stack để chờ sau này gặp ')' sẽ xử lý sau
* Nếu c là dấu đóng ngoặc ')': rỡ các phép toán trên ngọn stack đưa ra pol cho đến khi gặp dấu mở ngoặc '(' thì bỏ đi.

Cuối cùng ta vét nốt các phép toán trên stack đưa ra pol.

#### Minh họa

|  |  |  |  |
| --- | --- | --- | --- |
| Compile('c\*(d + e - b\*h)') = ? | | |  |
|  |  | st | pol |
|  | Khởi trị | [] | '' |
| 'c' | đưa 'c' ra pol |  | 'c' |
| '\*' | nạp '\*' vào st | ['\*'] |  |
| '(' | nạp '(' vào st | ['\*', '('] |  |
| 'd' | đưa 'd' ra pol |  | 'cd' |
| '+' | nạp '+' vào st | ['\*','(','+'] |  |
| 'e' | đưa 'e' ra pol |  | 'cde' |
| '-' | rỡ '+' từ st ra pol  nạp '-' vào st | ['\*','(','-'] | 'cde+' |
| 'b' | đưa 'b' ra pol |  | 'cde+b' |
| '\*' | nạp '\*' vào st | ['\*','(','-','\*'] |  |
| 'h' | đưa 'h' ra pol |  | 'cde+bh' |
| ')' | rỡ st ra pol đến khi gặp '('  thì bỏ '(' | ['\*'] | 'cde+bh\*-' |
|  | Hết input: vét not st ra pol |  | 'cde+bh\*-\*' |
|  | return pol |  | 'cde+bh\*-\*' |

Một phép toán một ngôi có thể được sinh ra trong qua trình chuyển đổi biểu thức sang dạng Poland. Đó là hép đổi dấu. Ví dụ e = -b cần được hiểu là đổi dấu của số, b do đó -b cần được dịch sang dạng Ba Lan là b~, với ~ là kí hiệu phép toán đổi dấu. Quan sát kỹ bạn sẽ phát hiện ra rằng phép tóan - được chuyển thành phép đổi dấu khi và chỉ khi nó là phép toán 1 ngôi, cụ thể là khi - đứng đầu biểu thức hoặc ngay sau dấu '(". Ví dụ

-(b+c) sẽ được dịch thành bc+~

(-b+c) sẽ được dịch thành b~c+

Để tiện xử lí. ta bỏ các dấu cách trong biểu thức e và thêm phần tử đệm '#' vào đầu biểu thức e và ngăn đầu tiên (ngăn số 0) của stack.

Chương trình dưới đây thực hiện hai pha: Pha đầu tiên dịch biểu thức toán học e sang dạng Poland pol. Pha thứ hai thực hiện tính toán trên pol. Cả hai pha đều sử dụng stack.

#### Program

# Expression

# bậc ưu tiên

OP={'>':100,'<':100,'~':100,'!':100,'\*':200,'/':200,'%':200,'+':300,'-':300}

# các lỗi cú pháp

SE={1:'unmatched', 2:'unexpected or unmatched symbol'} # Syntax Eror

# các lỗi thực thi

RE={1:'divide by zero', 2:'negative parameter'} # Run Eror

SHAP = '#'

BL = chr(32) # dấu cách

def Go(msg = ' ? '):

if input(msg) == '.': exit(0)

def IsVar(c): return 'a' <= c <= 'z'

def IsOp(c): return c in OP

def Val(c): return ord(c) - ord('a')

def Deg(c): # bậc ưu tiên

return OP[c] if c in OP else 1000

def SynErr(e, c): # Lỗi cú pháp

print('Syntax Error',e,end =': ')

print(SE[e], c)

exit(e)

def RunErr(e, c): # Lỗi thực thi

print('Run Error',e,end =': ')

print(RE[e], c)

exit(e)

def Compile(e):

s = SHAP + e.strip().replace(chr(32),'') # xóa các dấu cách

pol = ''

st = [SHAP]

for i in range(1,len(s)):

c = s[i]

if IsVar(c): pol += c; continue

if c == '(': st.append(c); continue

if c == ')':

while IsOp(st[-1]): pol += st.pop()

if st[-1] == SHAP: SynErr(1, c)

if st[-1] == '(': st.pop(); continue

if IsOp(c):

if c in '+-' and s[i-1] in '#(':

if c == '-': c = '~'

else: continue

# đưa các phếp toán có Deg <= Dec(c) trong st ra pol

while Deg(st[-1]) <= Deg(c): pol += st.pop()

st.append(c) # Nạp c vào st

continue

if c != BL: SynErr(2, c)

# Rỡ hết các phép toan trên st ra pol

while IsOp(st[-1]): pol += st.pop()

if len(st) > 1: SynErr(2, st[-1])

return pol

def Fac(n):

v = 1

for i in range(2,n+1): v \*= i

return v

def Exe(pol):

st = []

for c in pol:

if IsVar(c): st.append(Val(c)); continue

if c == '~': st[-1] = -st[-1]; continue

if c == '>': st[-1] += 1; continue

if c == '<': st[-1] -= 1; continue

if c == '!':

if st[-1] < 0: RunErr(2,st[-1])

st[-1] = Fac(st[-1])

continue

if c == '+':

st[-2] += st[-1]

st.pop()

continue

if c == '-':

st[-2] -= st[-1]

st.pop()

continue

if c == '\*':

st[-2] \*= st[-1]

st.pop()

continue

if c == '/':

if st[-1] == 0: RunErr(1,st[-1])

st[-2] //= st[-1]

st.pop()

continue

if c == '%':

if st[-1] == 0: RunErr(1,st[-1])

st[-2] %= st[-1]

st.pop()

continue

return st[-1]

# APPLICATION

ee = ['c+d\*k', 'd\*d-c\*d\*h+h\*h','c-d>!+k\*d','(k-c\*d)-(c+(d+b\*e))', '-(e-b)', '-(-e-b)']

for e in ee:

print(e, end = ' -> ')

pol = Compile(e)

print(pol,':', Exe(pol))

print(" T h e E n d")