Piles

Informatique pour tou(te)s - semaines 39 & 40

Guillaume CONNAN

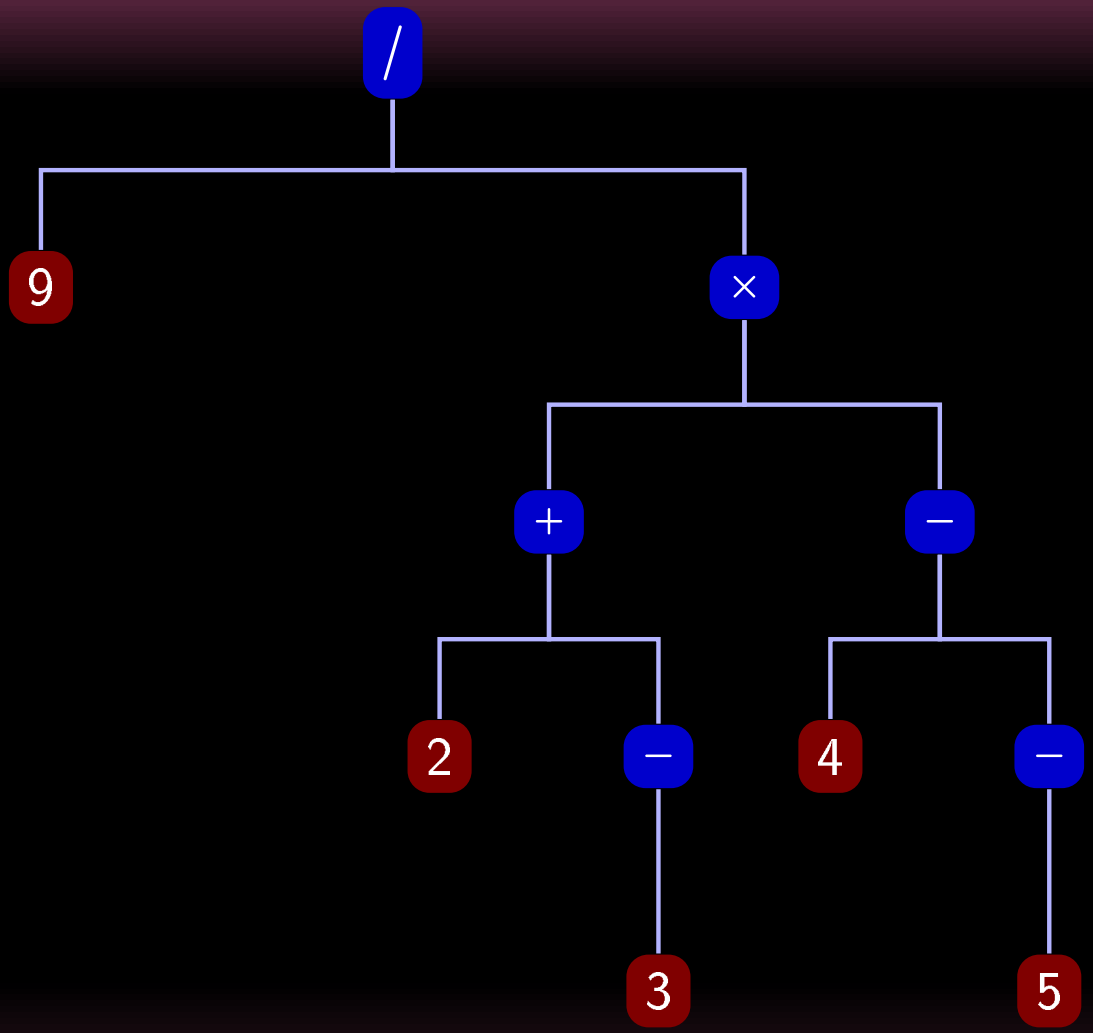
septembre 2015

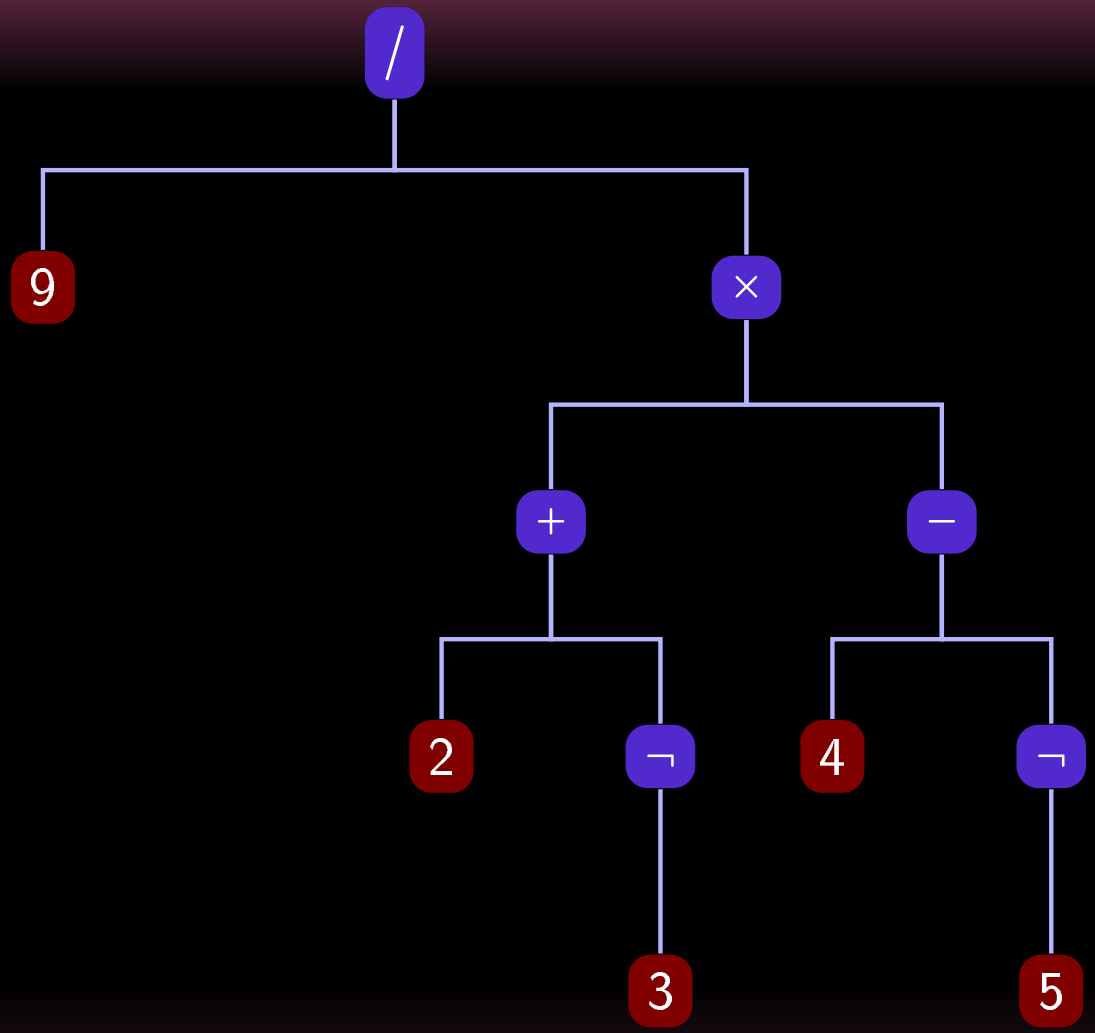
Lycée Clemenceau - MP / MP*

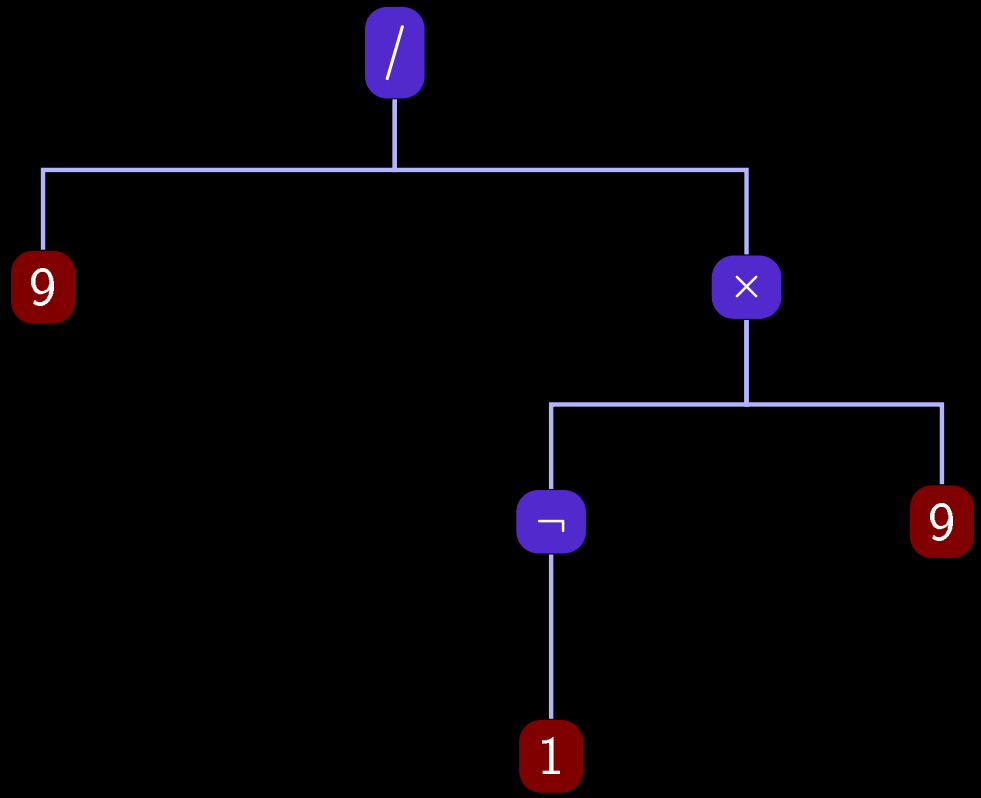
Jan Łukasiewicz (1878–1956)

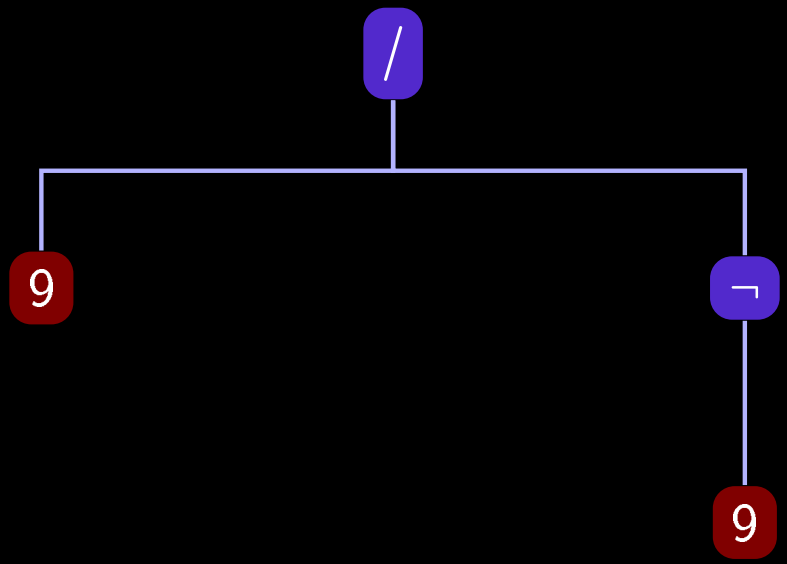
Charles Hamblin (1922 - 1985)

```
In [55]: 9 / 2 + (-3) * 4 - (-5)
Out[55]: -2.5

In [56]: 9 / ((2 + (-3)) * (4 - (-5)))
Out[56]: -1.0
```


/
9
×
+
−
2
−
4
−
3
5


/
9
×
+
−
2
¬
4
¬
3
5


/
9
×
¬
9
1


/
9
¬
9

$\neg 1$

- Pile
- Stack
- LIFO
- DEPS
- empiler / dépiler
- push / pop

- Pile
- Stack
- LIFO
- DEPS
- empiler / dépiler
- push / pop

- Pile
- Stack
- LIFO
- DEPS
- empiler / dépiler
- push / pop

- Pile
- Stack
- LIFO
- DEPS
- empiler / dépiler
- push / pop

- Pile
- Stack
- LIFO
- DEPS
- empiler / dépiler
- push / pop

- Pile
- Stack
- LIFO
- DEPS
- empiler / dépiler
- push / pop

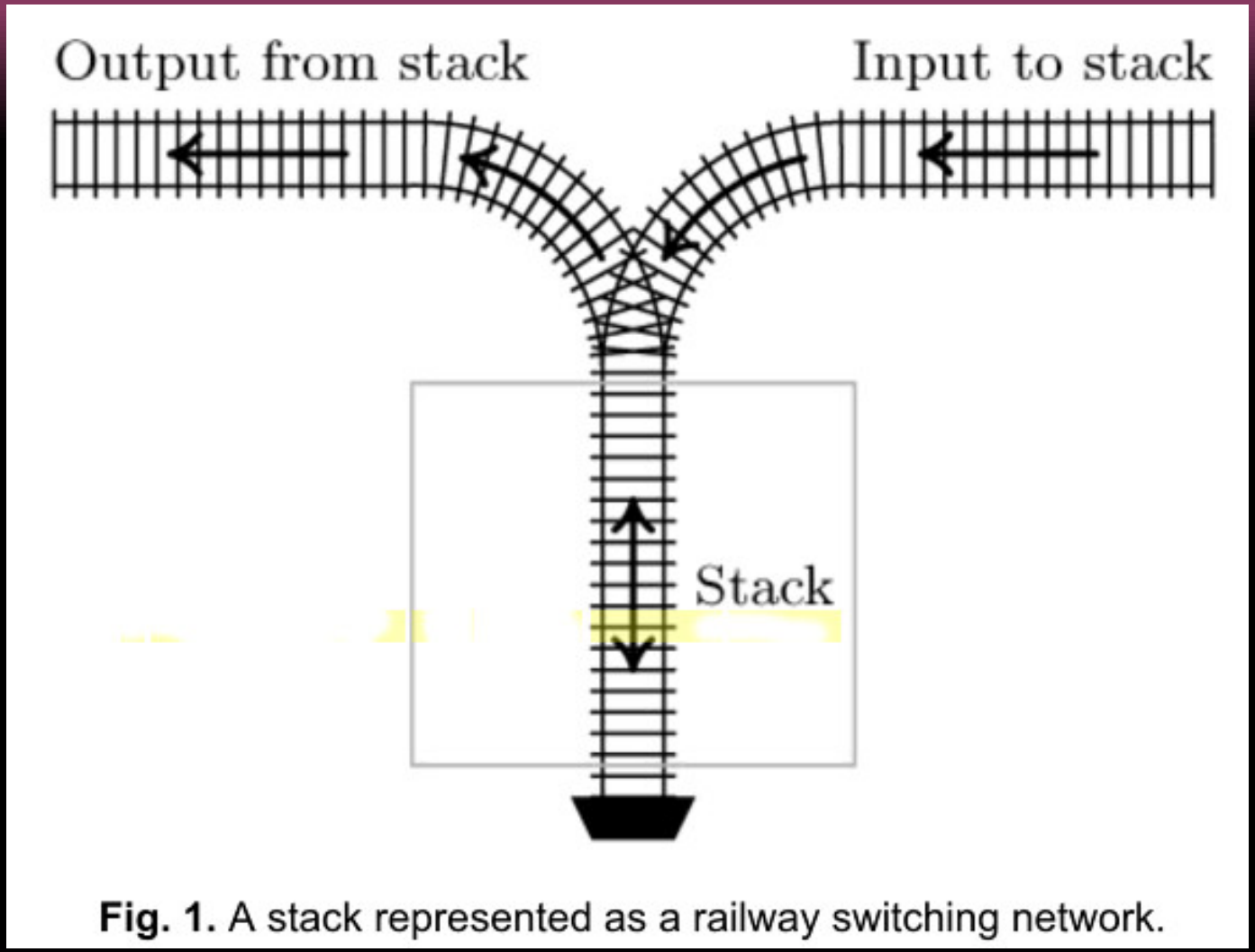
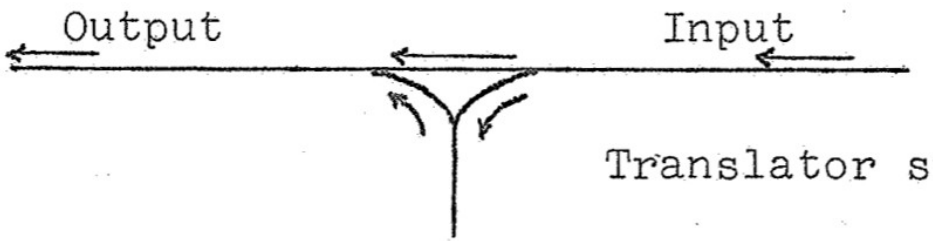


Fig. 1. A stack represented as a railway switching network.

D.E. Knuth

The translation process shows much resemblance to shunting at a three way railroad junction of the following form



E. Dijkstra

Extrait de l'épreuve X-ENS (MP option SI et PC) 2015

Dans la suite nous aurons besoin d'utiliser des piles d'entiers, dont on rappelle la définition ci-dessous :

Définition 2 *Une pile d'entiers est une structure de données permettant de stocker des entiers et d'effectuer les opérations suivantes en temps constant (indépendant de la taille de la pile) :*

- *créer une nouvelle pile vide,*
- *déterminer si la pile est vide,*
- *insérer un entier au sommet de la pile,*
- *déterminer la valeur de l'entier au sommet de la pile,*
- *retirer l'entier au sommet de la pile.*

Nous supposerons fournies les fonctions suivantes, qui réalisent les opérations ci-dessus et s'executent chacune en temps constant :

- newStack(), qui ne prend pas d'argument et renvoie une pile vide,
- isEmpty(s), qui prend une pile s en argument et renvoie True ou False suivant que s est vide ou non,
- push(i, s), qui prend un entier i et une pile s en argument, insère i au sommet de s (c'est-à-dire à la fin de la liste), et ne renvoie rien,
- top(s), qui prend une pile s (supposée non vide) en argument et renvoie la valeur de l'entier au sommet de s (c'est-à-dire à la fin de la liste),
- pop(s), qui prend une pile s (supposée non vide) en argument, supprime l'entier au sommet de s (c'est-à-dire à la fin de la liste) et renvoie sa valeur.

Dans la suite il est demandé aux candidats de manipuler les piles uniquement au travers de ces fonctions, sans aucune hypothèse sur la représentation effective des piles en mémoire.

Premier contact avec la POO

Les piles fabriquées avec des tuples :

```
class pile0 () :

    p = ()

    def depile(self) :
        temp = self.p[-1]
        self.p = self.p[0:-1]
        return temp

    def empile(self, val) :
        self.p = self.p + (val,)

    def est_vide(self) :
        return self.p == ()
```

Premier contact avec la POO

Les piles fabriquées avec des tuples :

```
class pile0 () :

    p = ()

    def depile(self) :
        temp = self.p[-1]
        self.p = self.p[0:-1]
        return temp

    def empile(self, val) :
        self.p = self.p + (val,)

    def est_vide(self) :
        return self.p == ()
```

Second contact avec la POO

Les piles fabriquées avec des listes :

```
class pile1 () :

    p = []

    def dépile(self):
        return self.p.pop()

    def empile(self, val):
        self.p.append(val)

    def est_vide(self):
        return self.p == []
```

Second contact avec la POO

Les piles fabriquées avec des listes :

```
class pile1 () :

    p = []

    def dépile(self):
        return self.p.pop()

    def empile(self, val):
        self.p.append(val)

    def est_vide(self):
        return self.p == []
```

Pop de Python

Pop de Python

```
In [1]: ?list.pop
Docstring:
L.pop([index]) -> item -- remove and return item at index (default last).
Raises IndexError if list is empty or index is out of range.
Type:      method_descriptor
```

Contact plus musclé avec la POO

```
class Pile() :
    from copy import copy

    def __init__(self, sommet = None, suivant = None) :
        self.sommet = sommet
        self.suivant = suivant

    def est_vide(self) :
        return self.sommet == None

    def empile(self, val) :
        p = copy(self)
        self.suivant = p
        self.sommet = val
```

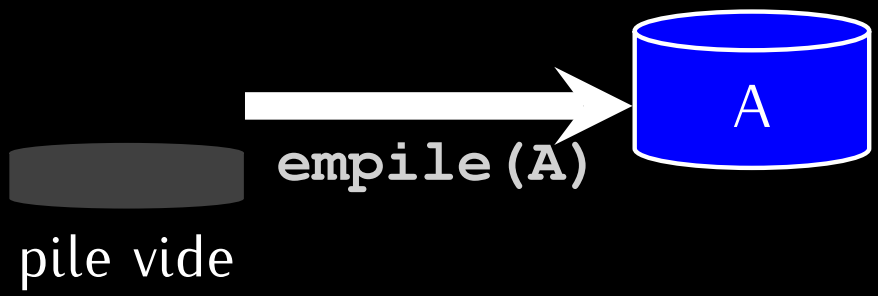
```python
def depile(self) :
    assert not self.est_vide(), "Dépile pile vide"
    p = copy(self)
    top = p.sommet
    self.sommet = p.suivant.sommet
    self.suivant = p.suivant.suivant
    return top
```

```
    def __repr__(self) :
        return str(self.sommet) + "|-"
```

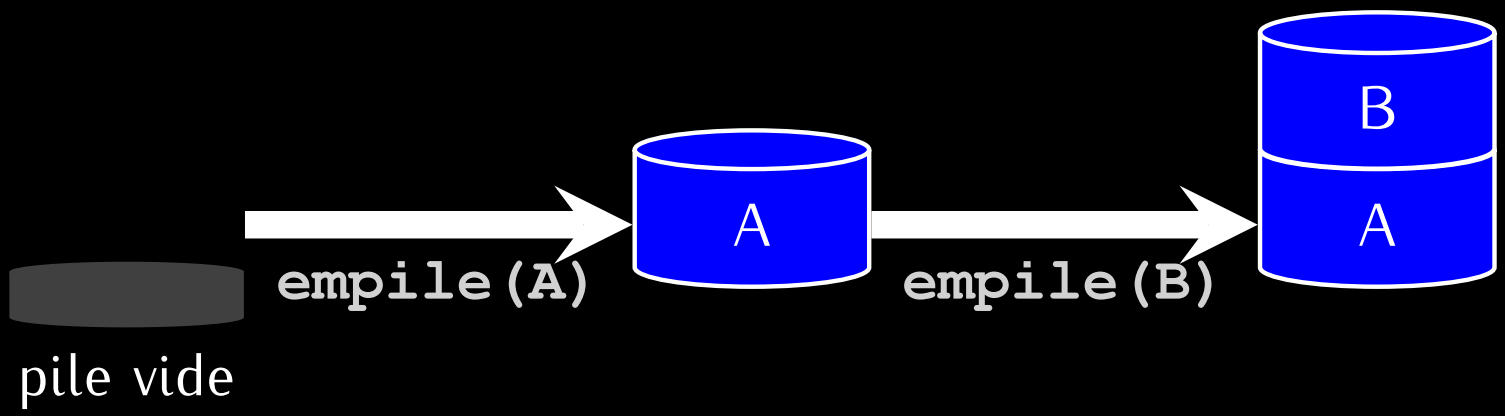
```
In [123]: p = Pile()

In [124]: p.sommet

In [125]: p
Out[125]: None|-

In [126]: p.empile(1)

In [127]: p.empile(2)

In [128]: p
Out[128]: 2|-

In [129]: p.empile(3)

In [130]: p
Out[130]: 3|-
```

```
In [131]: p.depile()
Out[131]: 3

In [132]: p
Out[132]: 2|-

In [133]: p.depile()
Out[133]: 2

In [134]: p
Out[134]: 1|-

In [135]: p.depile()
Out[135]: 1

In [136]: p
Out[136]: None|-

In [137]: p.depile()
---------------------------------------------------------------------------
AssertionError: Dépile pile vide
```

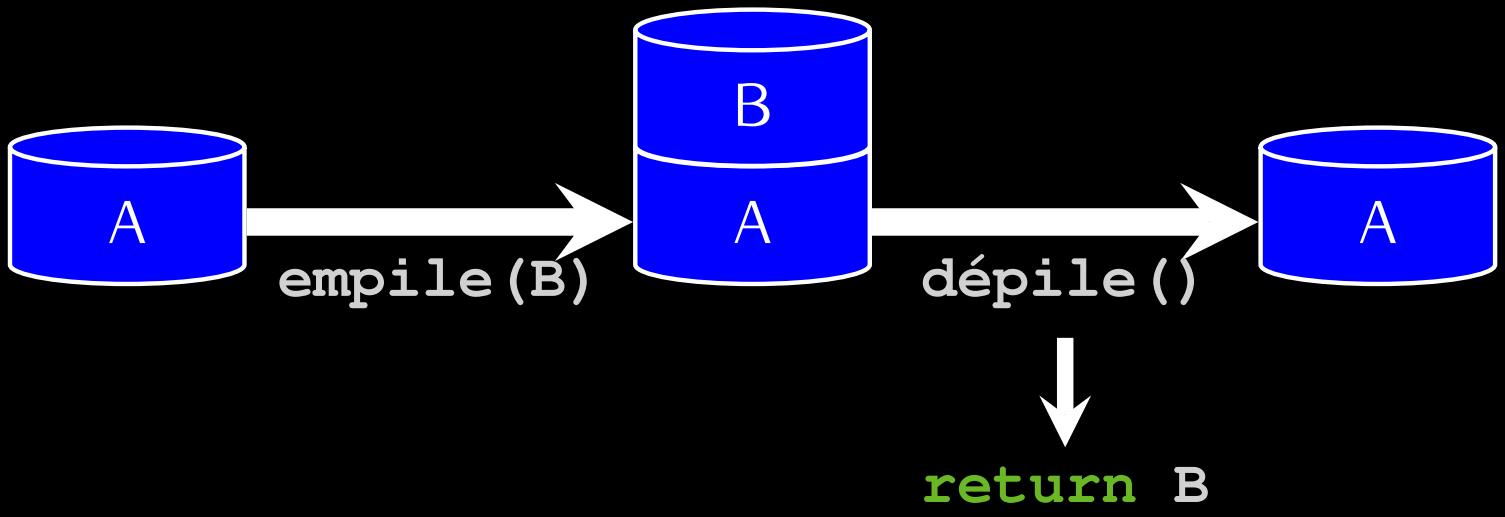
Exemple



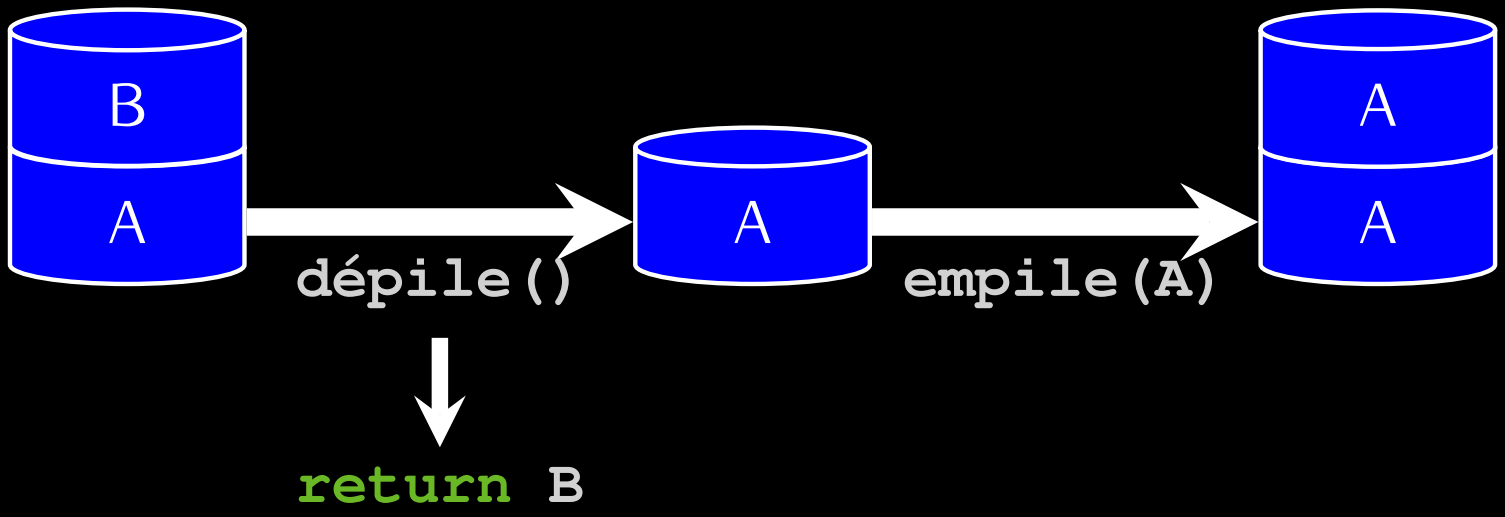
Exemple



Exemple



Exemple



HP15c


HP-15C
15C
A √x x² B eˣ LN C 10ˣ LOG D yˣ % E 1/x Δ% MATRIX CHS ABS FIX 7 DEG SCI 8 RAD ENG 9 GRD SOLVE ÷ x≤y
LBL SST BST HYP GTO HYP⁻¹ DIM SIN SIN⁻¹ () COS COS⁻¹ I TAN TAN⁻¹ RESULT EEX π x↔ 4 SF DSE 5 CF ISG 6 F? ∫xy × x=0
CLEAR
PSE R/S P/R Σ GSB RTN PRGM R↓ R↑ REG x↔y RND PREFIX ← CLx RAN # ENTER LSTx → R 1 →P →H.MS 2 →H →RAD 3 →DEG Re↔Im − TEST
FRAC USER ON f g STO INT RCL MEM x! 0 x̄ ŷ,r . s L.R. Σ+ Σ− Py,x + Cy,x
HEWLETT·PACKARD

$$9/((2 + \neg 3) \times (4 - \neg 5))$$

```
9 2 3 NEG + 4 5 NEG - * /
```

1. On lit l'expression de gauche à droite ;
2. On empile les opérandes ;
3. Dès qu'on lit un opérateur, on l'applique aux opérandes présents sur la pile selon son arité ;
4. On s'arrête quand on n'a plus rien à lire ;
5. on renvoie le dernier élément présent dans la pile.

$$9/((2 + \neg 3) \times (4 - \neg 5))$$

```
9 2 3 NEG + 4 5 NEG - * /
```

1. On lit l'expression de gauche à droite ;
2. On empile les opérandes ;
3. Dès qu'on lit un opérateur, on l'applique aux opérandes présents sur la pile selon son arité ;
4. On s'arrête quand on n'a plus rien à lire ;
5. on renvoie le dernier élément présent dans la pile.

$$9/((2 + \neg 3) \times (4 - \neg 5))$$

```
9 2 3 NEG + 4 5 NEG - * /
```

1. On lit l'expression de gauche à droite ;
2. On empile les opérandes ;
3. Dès qu'on lit un opérateur, on l'applique aux opérandes présents sur la pile selon son arité ;
4. On s'arrête quand on n'a plus rien à lire ;
5. on renvoie le dernier élément présent dans la pile.

Calculator Emacs

On entre 9 , 2 puis 3 :

```
 Emacs Calculator Mode
3:  9
2:  2
1:  3
.
```

Calculator Emacs

NEG

```
Emacs Calculator Mode
3:  9
2:  2
1:  -3
```

Calculator Emacs

```
+
-----------------------------------------------------------------------

 Emacs Calculator Mode
2:  9
1:  -1
.
.

-----------------------------------------------------------------------
```

Calculator Emacs

4 puis 


```
 Emacs Calculator Mode
3:  9
2:  -1
1:  4
.
```

Calculator Emacs

5 puis 


```
 Emacs Calculator Mode
4:  9
3:  -1
2:  4
1:  5
```

Calculator Emacs

NEG

```
 Emacs Calculator Mode
4:  9
3:  -1
2:  4
1:  -5
```

Calculator Emacs

```
 Emacs Calculator Mode
3:  9
2:  -1
1:  9
.
```

Calculator Emacs

*

```
 Emacs Calculator Mode
2:  9
1:  -9
.
.
```

Calculator Emacs

/

```
 Emacs Calculator Mode
1: -1
.
.
.
```

- pas besoin de parenthèses, de règles d'associativité, de priorité ;
- on voit au sommet de la pile les résultats intermédiaires ;
- le calcul est directement implémentable sur machine.

- pas besoin de parenthèses, de règles d'associativité, de priorité ;
- on voit au sommet de la pile les résultats intermédiaires ;
- le calcul est directement implémentable sur machine.

- pas besoin de parenthèses, de règles d'associativité, de priorité ;
- on voit au sommet de la pile les résultats intermédiaires ;
- le calcul est directement implémentable sur machine.

Notation polonaise

$$9/((2 + \neg 3) \times (4 - \neg 5))$$

« le *rapport* entre 9 et le *produit* de la *somme* de 2 et l'*opposé* de 3 et la *différence* de 4 et de l'*opposé* de 5 »

```
/ 9 * + 2 NEG 3 - 4 NEG 5
```

Sur machine ?

Notation polonaise

$$9/((2 + \neg 3) \times (4 - \neg 5))$$

« le *rapport* entre 9 et le *produit* de la *somme* de 2 et l'*opposé* de 3 et la *différence* de 4 et de l'*opposé* de 5 »

```
/ 9 * + 2 NEG 3 - 4 NEG 5
```

Sur machine ?

Notation polonaise

$$9/((2 + \neg 3) \times (4 - \neg 5))$$

« le *rapport* entre 9 et le *produit* de la *somme* de 2 et l'*opposé* de 3 et la *différence* de 4 et de l'*opposé* de 5 »

```
/ 9 * + 2 NEG 3 - 4 NEG 5
```

Sur machine ?

Notation polonaise

$$9/((2 + \neg 3) \times (4 - \neg 5))$$

« le *rapport* entre 9 et le *produit* de la *somme* de 2 et l'*opposé* de 3 et la *différence* de 4 et de l'*opposé* de 5 »

```
/ 9 * + 2 NEG 3 - 4 NEG 5
```

Sur machine ?

Exercice 1

Décomposez le calcul `/ 9 * + 2 NEG 3 - 4 NEG 5`. *Comment l'organiser sur machine ? Comparez avec la NPI. Expliquez alors le choix fait par Charles* Hamblin *d'utiliser plutôt la NPI sur machine.*

Lors de l'appel d'une fonction, que se passe-t-il ?

Une bloc est créé pour la fonction, contenant :

- Les arguments de la fonction
- Une place pour écrire la valeur de retour
- L'*adresse de retour*, i.e., une information disant ce qu'il faudra faire une fois la fonction terminée.
- De la place pour les variables locales
- Éventuellement d'autres informations

Lors de l'appel d'une fonction, que se passe-t-il ?

Une bloc est créé pour la fonction, contenant :

- Les arguments de la fonction
- Une place pour écrire la valeur de retour
- L'*adresse de retour*, i.e., une information disant ce qu'il faudra faire une fois la fonction terminée.
- De la place pour les variables locales
- Éventuellement d'autres informations

Lors de l'appel d'une fonction, que se passe-t-il ?

Une bloc est créé pour la fonction, contenant :

- Les arguments de la fonction
- Une place pour écrire la valeur de retour
- L'*adresse de retour*, i.e., une information disant ce qu'il faudra faire une fois la fonction terminée.
- De la place pour les variables locales
- Éventuellement d'autres informations

Lors de l'appel d'une fonction, que se passe-t-il ?

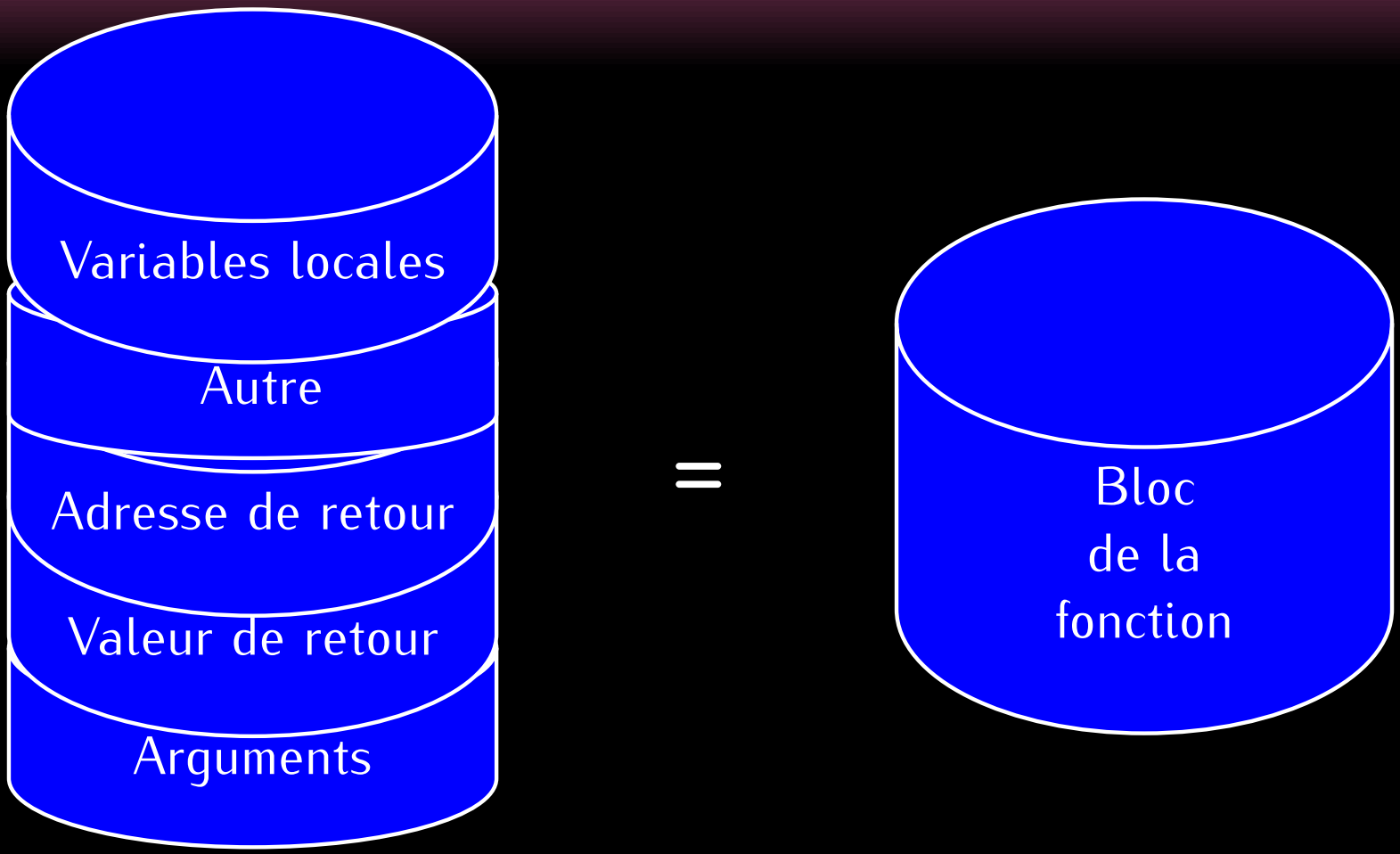
Une bloc est créé pour la fonction, contenant :

- Les arguments de la fonction
- Une place pour écrire la valeur de retour
- L'*adresse de retour*, i.e., une information disant ce qu'il faudra faire une fois la fonction terminée.
- De la place pour les variables locales
- Éventuellement d'autres informations

Lors de l'appel d'une fonction, que se passe-t-il ?

Une bloc est créé pour la fonction, contenant :

- Les arguments de la fonction
- Une place pour écrire la valeur de retour
- L'*adresse de retour*, i.e., une information disant ce qu'il faudra faire une fois la fonction terminée.
- De la place pour les variables locales
- Éventuellement d'autres informations


Variables locales
Autre
Adresse de retour
Valeur de retour
Arguments
=
Bloc
de la
fonction

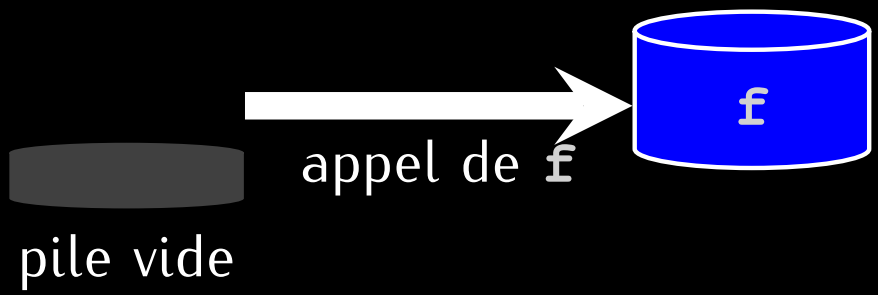
Lorsqu'une fonction appelle une autre fonction, que se passe-t-il ?

Un bloc de la fonction appelé est créé “au dessus” du bloc de la fonction appelante. Ces blocs sont empilé dans la “pile d'exécution”.

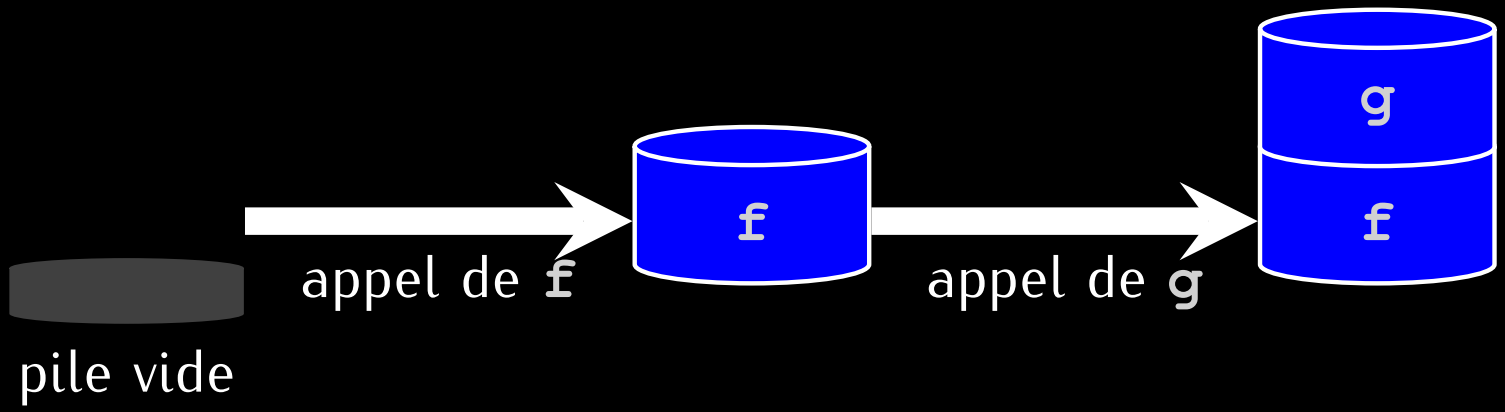
Comment sont gérés les appels de fonctions ?

```
def h(x) :
    return x + 1

def g(x) :
    return h(x) * 2

def f (x) :
    return g(x) + 1
```

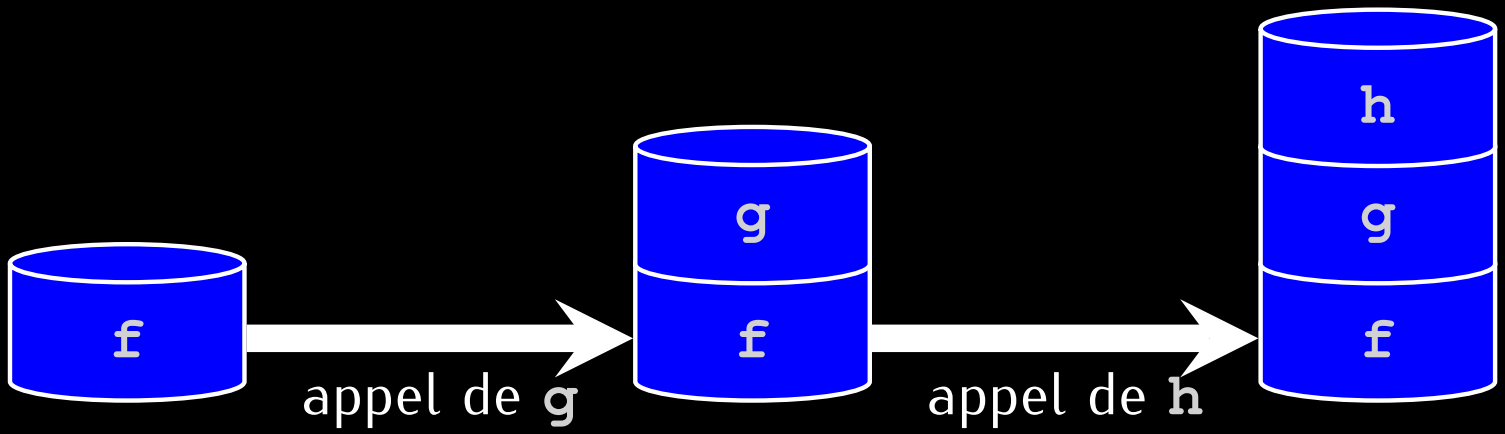
Exemple



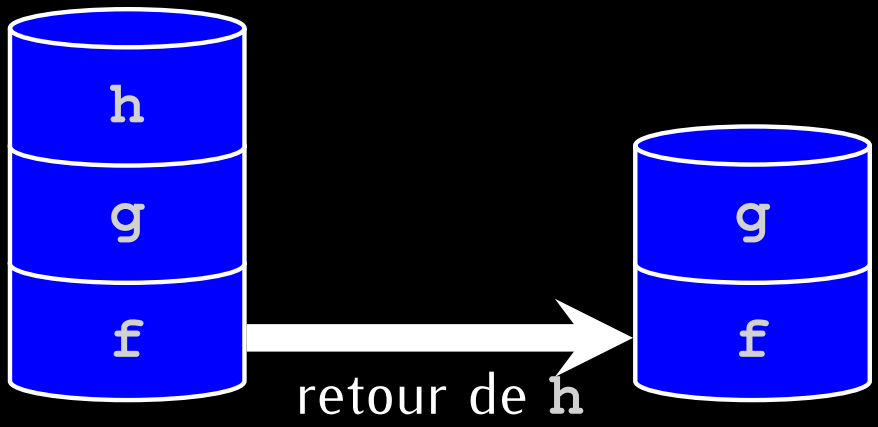
Exemple



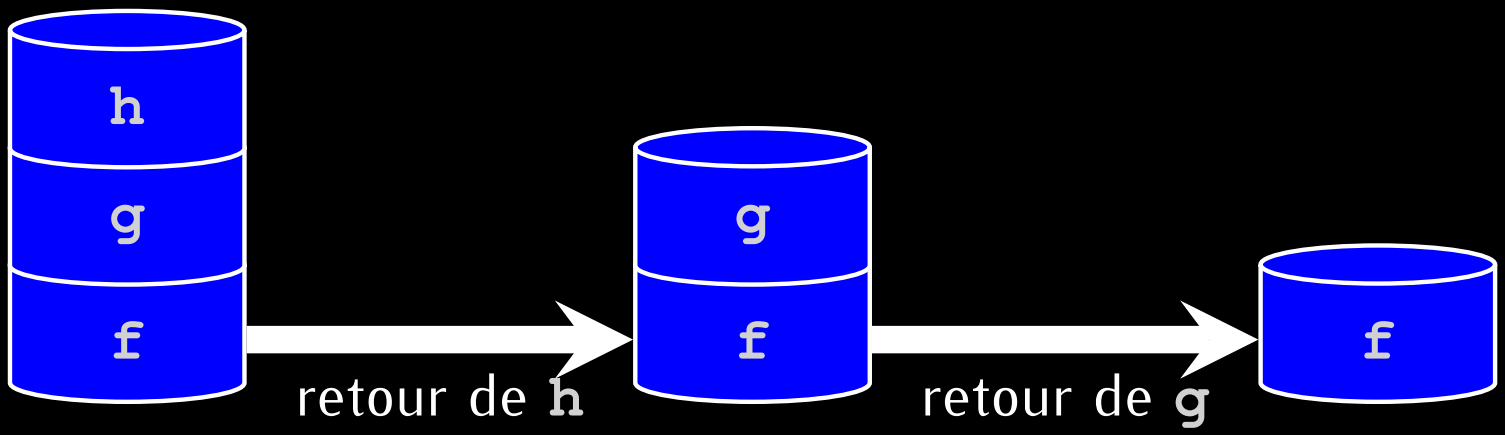
Exemple



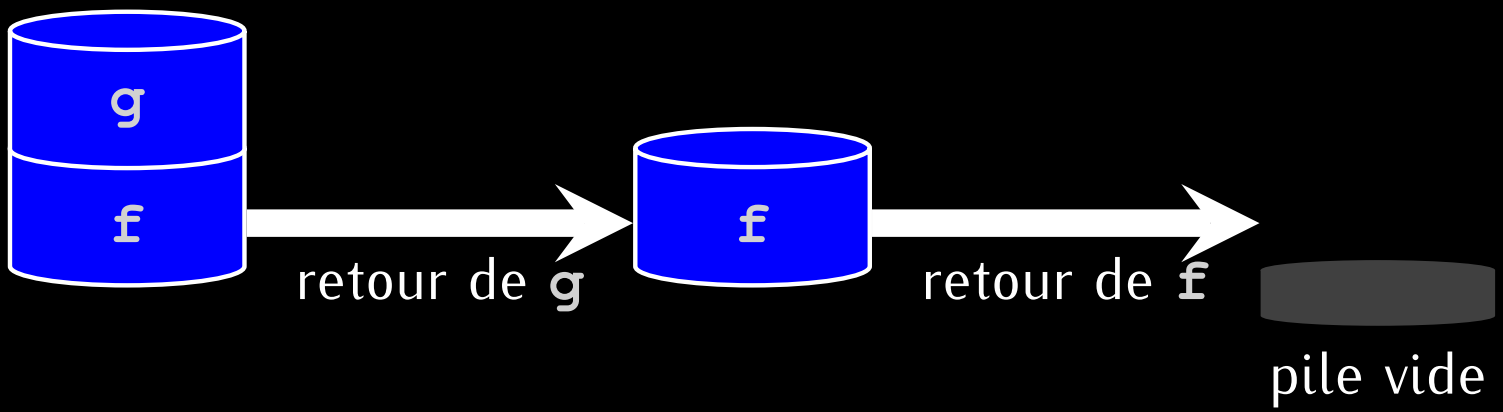
Exemple



Exemple

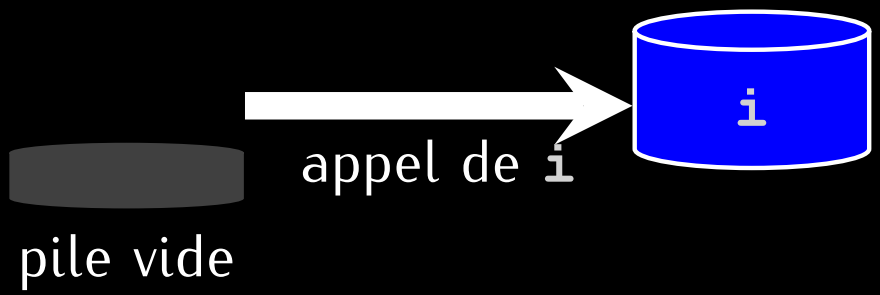


Exemple

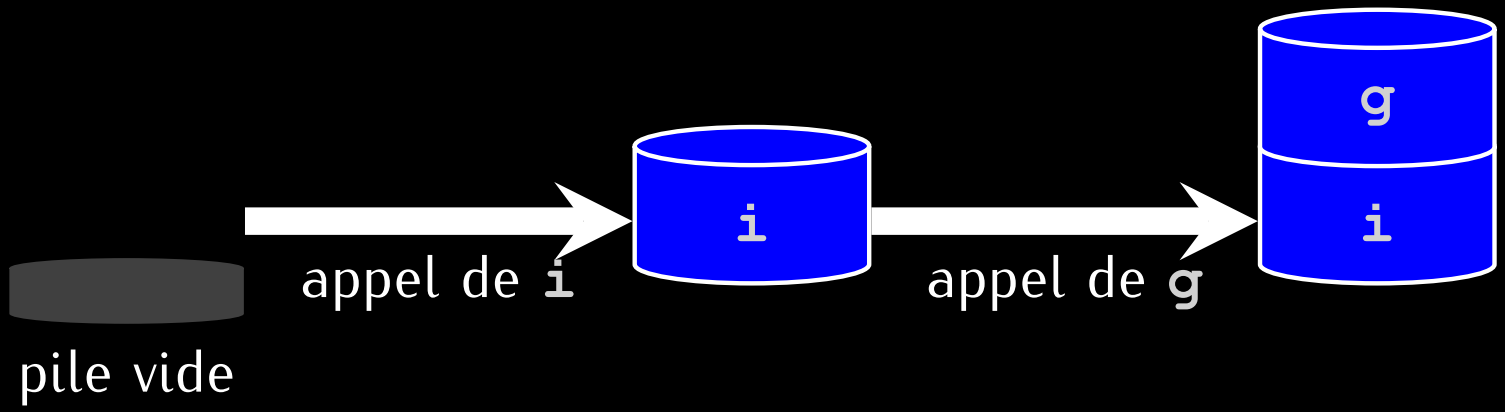


```
def h(x) :
    return x + 1

def g(x) :
    return h(x) * 2

def i(x) :
    a = g(x)
    b = h(x)
    return a + b
```

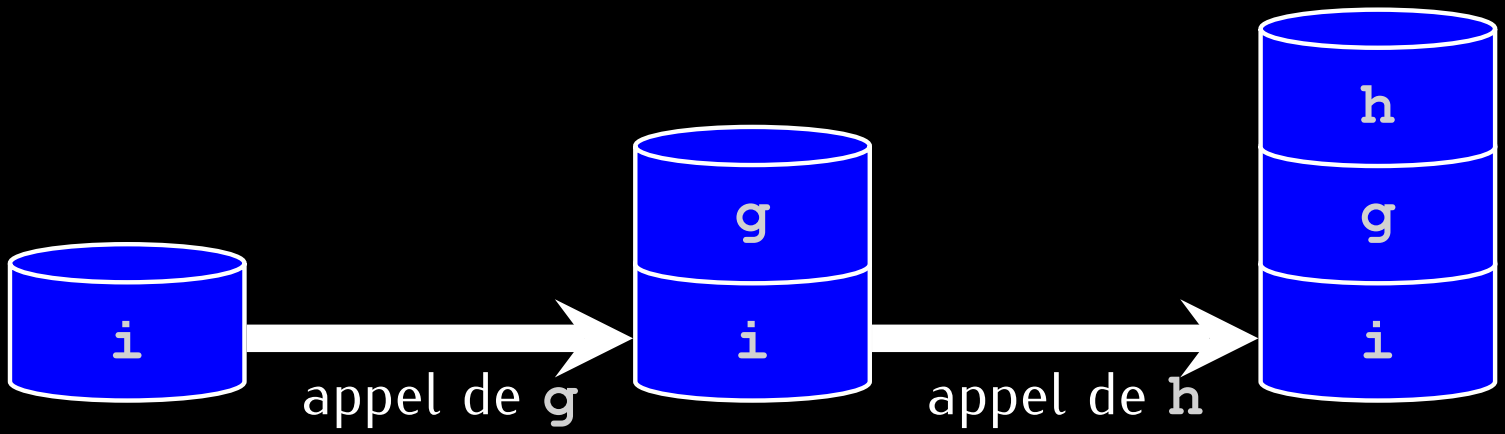
Exemple



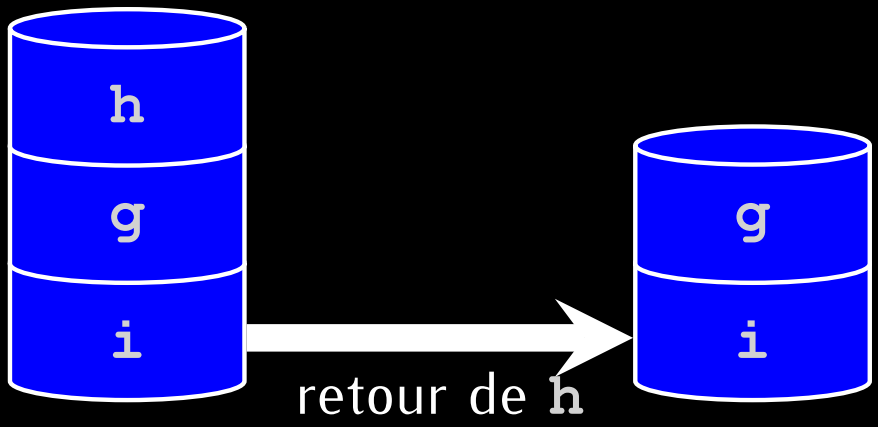
Exemple



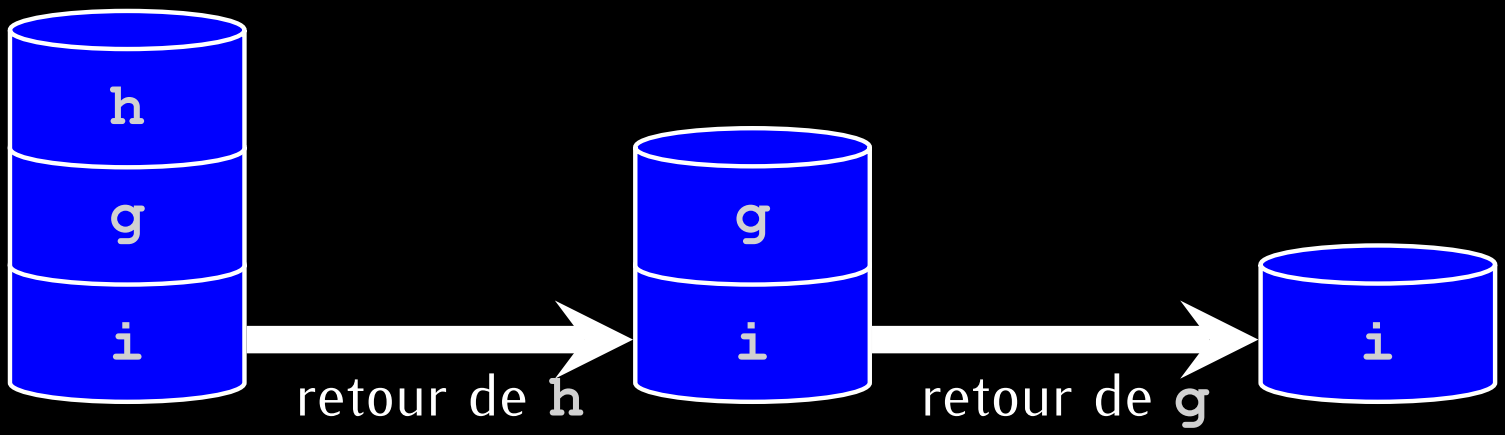
Exemple



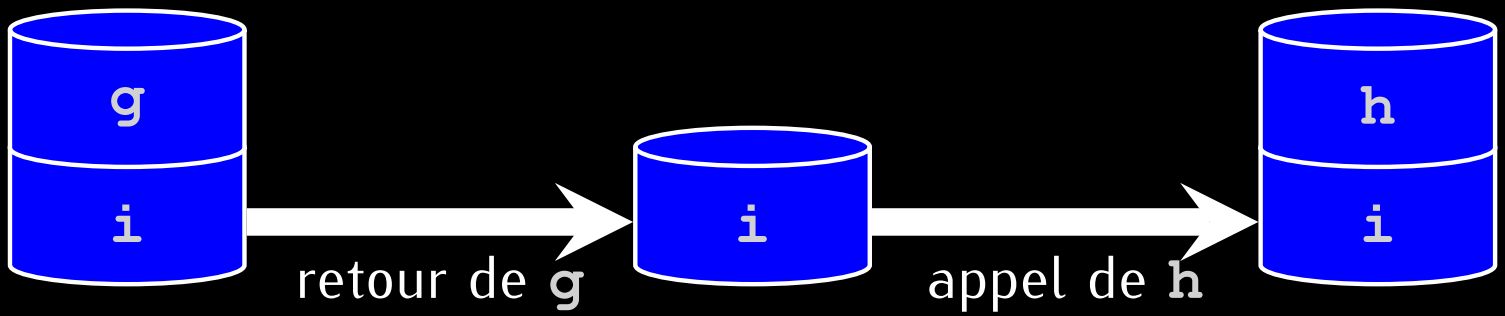
Exemple



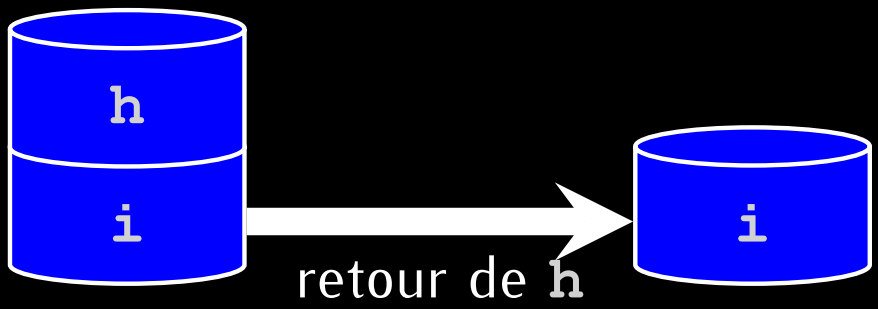
Exemple



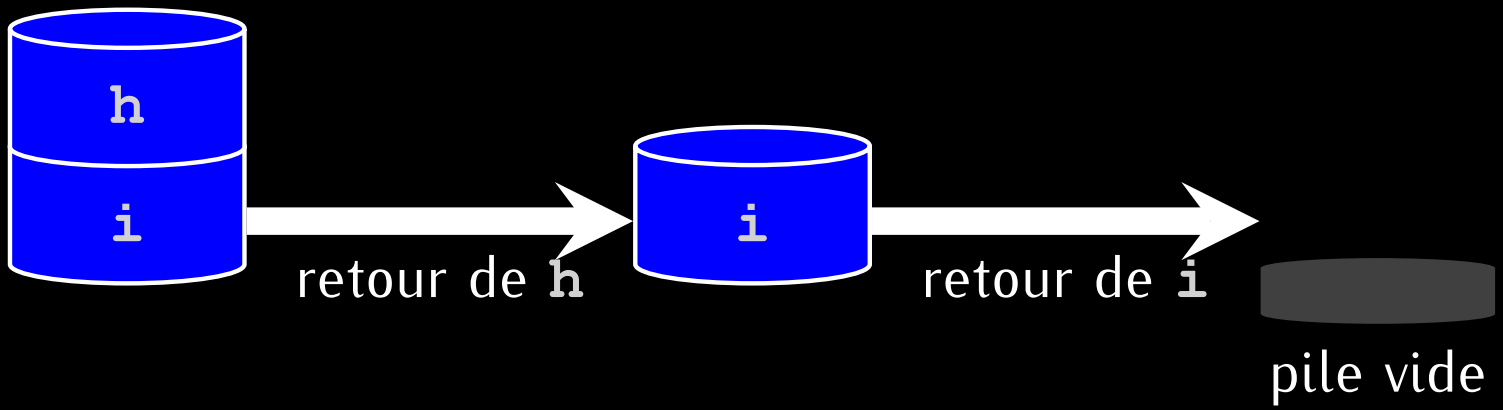
Exemple



Exemple



Exemple



Merci à Jean-Loup Carré pour ces beaux dessins de piles...