Les Fonctions

INF321
Mécanismes de programmation orientée-objet en Java

Amphi 2 - 24 avril 2017

Benjamin Werner

1 - Partager du code

Fonction sans arguments

```
  static void sauterTroisLignes() {
    System.out.println();
    System.out.println();
    System.out.println();
  }


…

sauterTroisLignes();
…
sauterTroisLignes();
…
sauterTroisLignes();
…
```

2 - Partager du code paramètré

Fonction avec arguments

```
static void annoncerTrain(String depart, int quai) {
  System.out.println("le train de "+ depart + "arrive quai " + quai);
}
```

```
annoncerTrain("Palaiseau", 1);
annoncerTrain("Amiens", 24);
annoncerTrain("Sablé", 19);
annoncerTrain("Saint-Cloud", 22);
annoncerTrain("Tanger", 18);
```

3 - Paramétrer un calcul

Fonction rendant un résultat

```
static int fact(int i) {
int r = 1;
while (i > 1) {
   r = r * i;
   i--;
   }
return(r);
 }
```

```
static int fact(int i) {
   if (i <= 1) return(1);
   return(i * fact(i - 1));
}
```

on peut appeler **`return`** partout dans la fonction

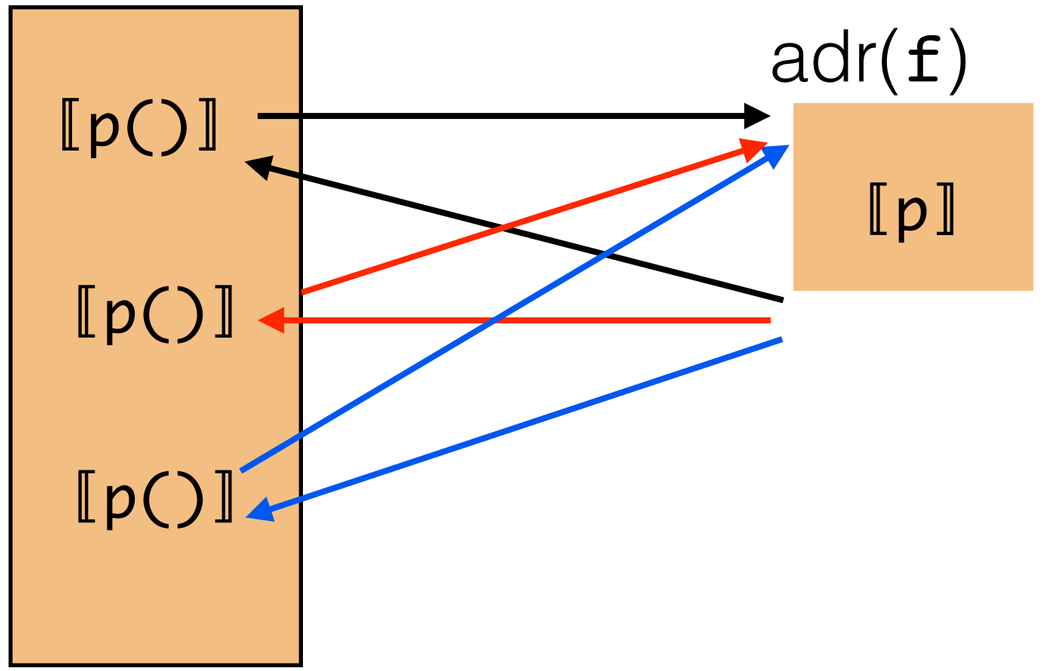
On peut aussi faire **`return;`** pour sortir d'une fonction **`void`** sans résultat

Compilation cas simple

Pas d'arguments, pas de résultat

```
  static void f() {
    p
  }


…
f();
…
f();
…
f();
…
```

Code compilé

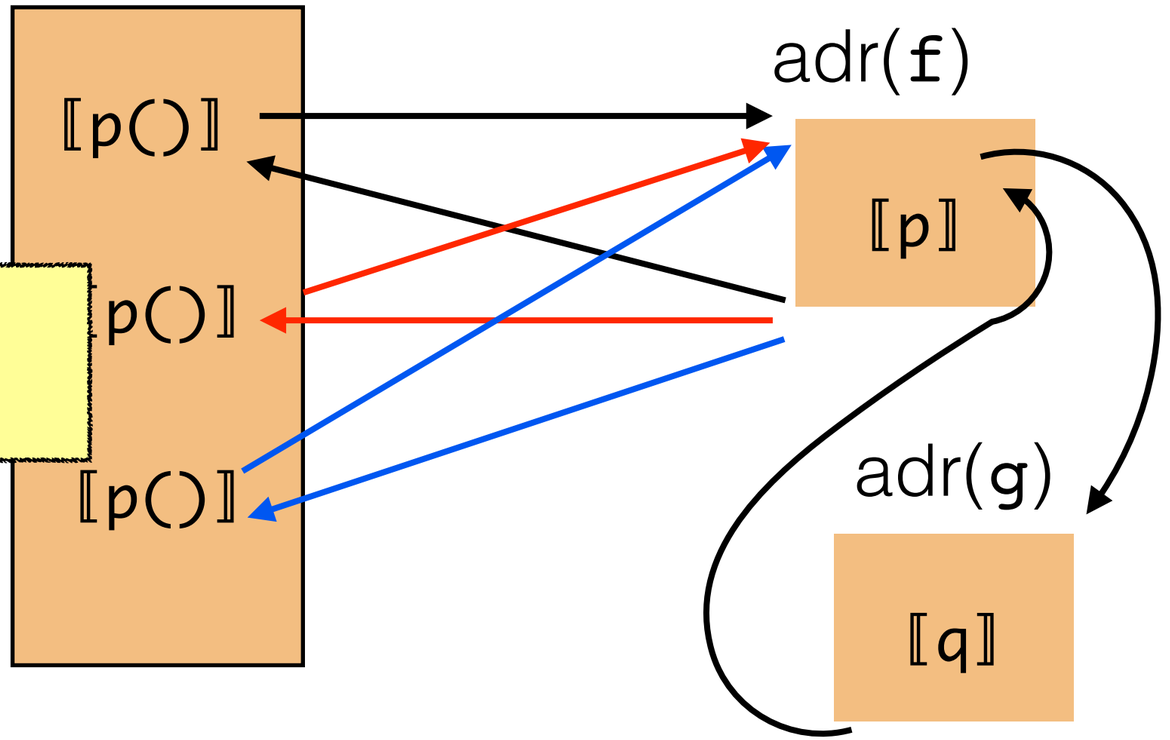


à chaque fois qu'on rentre dans le code de ⟦p⟧ il faut se souvenir de l'adresse de retour.

Compilation cas simple

```
static void f() {
  g();
  p
}

…
f();
…
f();
…
f();
…
```

Il faut se souvenir de plusieurs adresses de retour !



à chaque fois qu'on rentre dans le code de ⟦p⟧ il faut se souvenir de l'adresse de retour.

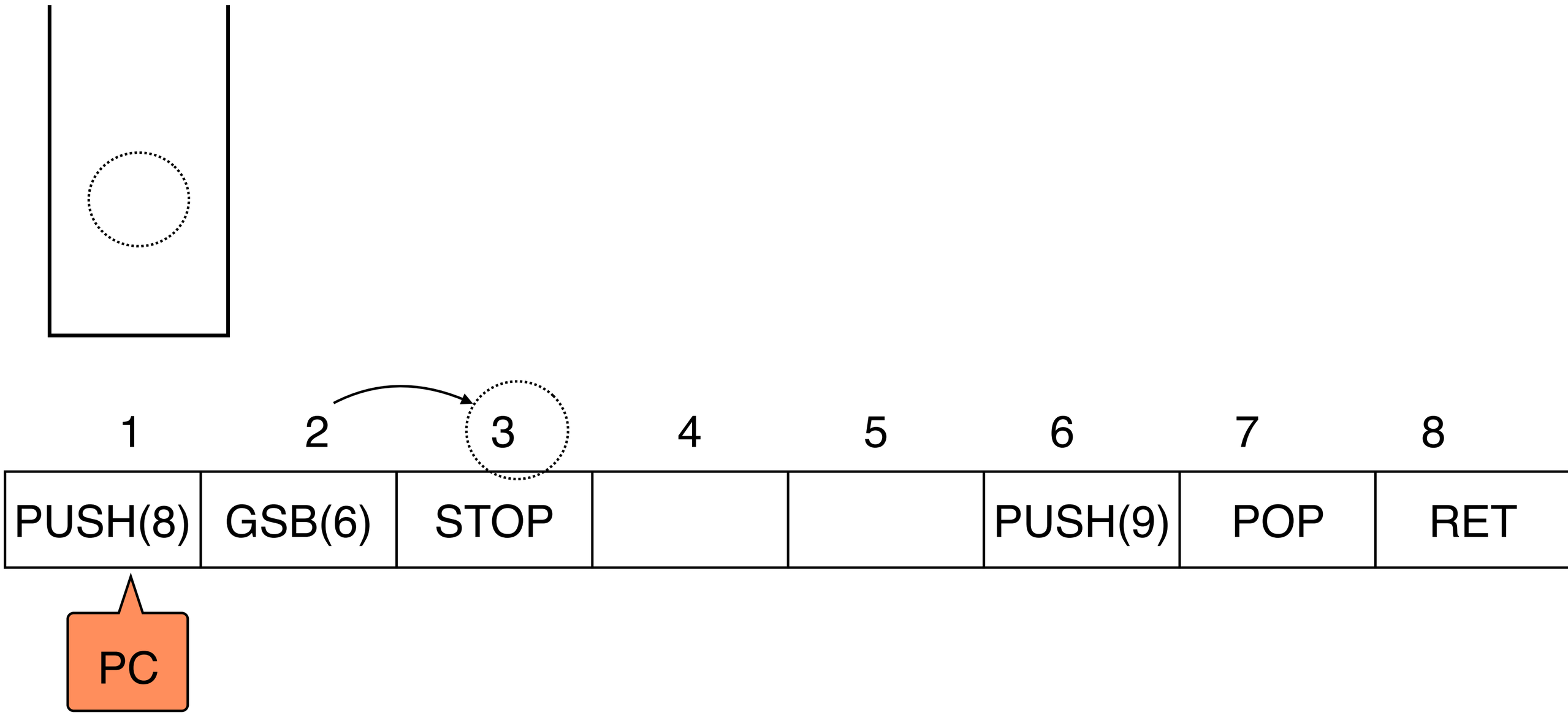
Des adresses dans la pile

On va utiliser la pile pour ranger les adresses de retour

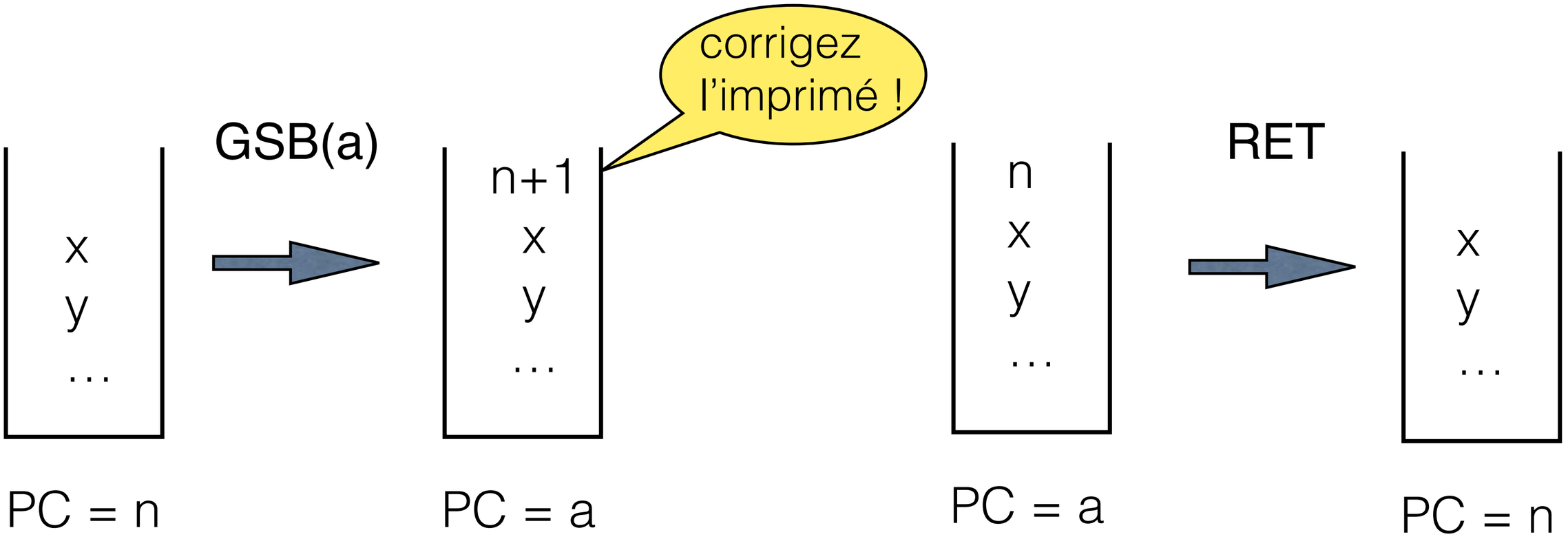
Une paire de nouvelles instructions :

GSB(a) Go to Subroutine
comme GTO, mais avant le saut on empile l'adresse courante

RET Return
retourne à l'adresse qui se trouve en haut de la pile

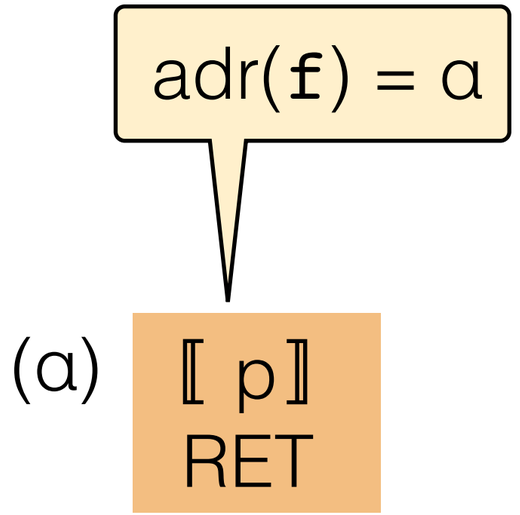

1
2
3
4
5
6
7
8
PUSH(8)
GSB(6)
STOP
PUSH(9)
POP
RET
PC

Formellement



Compilation 1 : sans arguments, sans résultat

$[\![\texttt{f()}]\!] = GSB(adr(f))$



On compte sur le fait que l'exécution de $[\![p]\!]$ rend la pile dans le même état que celui dans lequel elle l'a trouvé.

Compilation 2 : sans arguments, avec un résultat

```
static int i;

static int succ() {
  return(i + 1);
}
```

⟦ `i + 1` ⟧ place le résultat en haut de la pile (avant le **RET**)

```
static int x;
…
  x = succ();
…
```

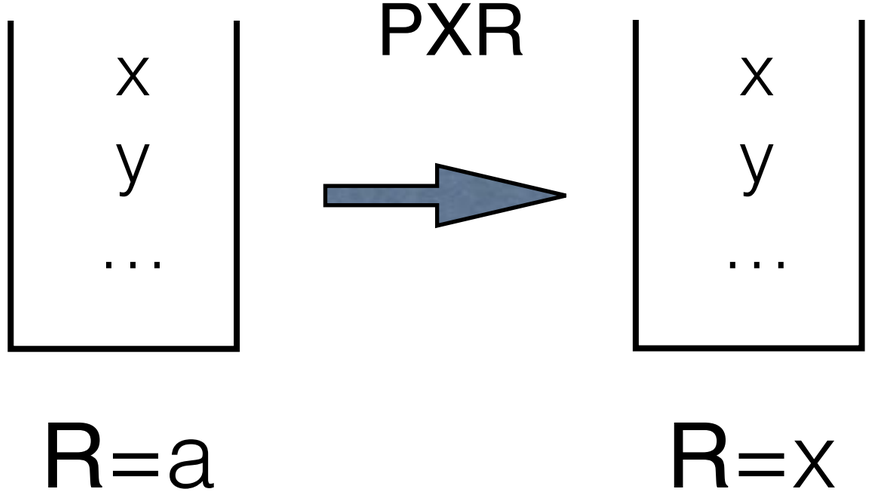
⟦ **`succ()`** ⟧ doit placer ce résultat en haut de la pile (après le **RET**)

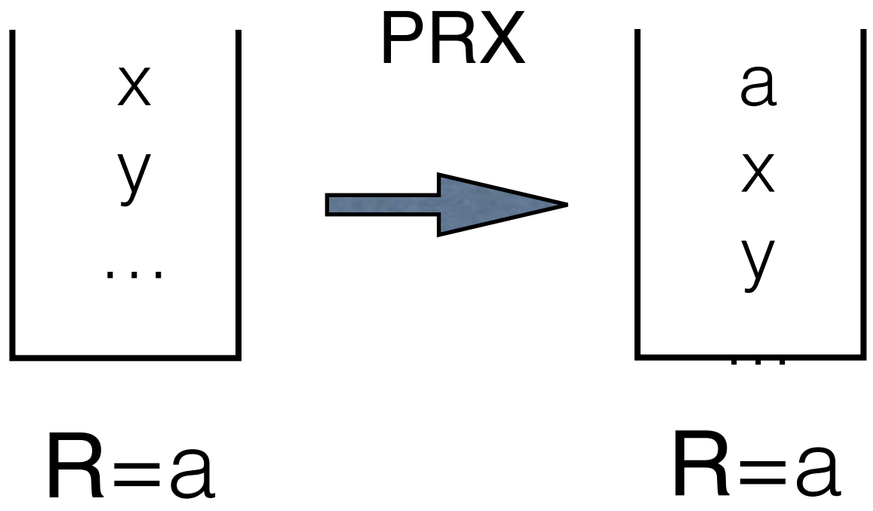
On va simplement « ranger » le résultat quelque part avant le **RET**

Un registre supplémentaire : R

Un emplacement mémoire spécial : **R** (il servira surtout au *résultat*)

Deux instructions : PXR (push X to R) et PRX (push R to X)





Compilation 2 : sans arguments, avec un résultat (bis)

```
static int i;

static int succ() {

  return(i + 1);
}
```

`return(i + 1);` est compilé en :

⟦ i + 1 ⟧ place le résultat en haut de la pile
PXR recopie ce résultat dans R
POP l'efface du haut de la pile
RET on retourne à l'adresse de retour

```
static int x;
…
  x = succ();
…
```

`succ()` est compilé en :

GSB(adr(`succ`))
PRX

on récupère le résultat et le place sur la pile (après le RET)

Compilation 3 : les arguments

```
static int sum(int x, int y) {
  return(i + x + y);
}
```

Dans le corps d'une fonction il y a :

- les arguments
- les variables globales

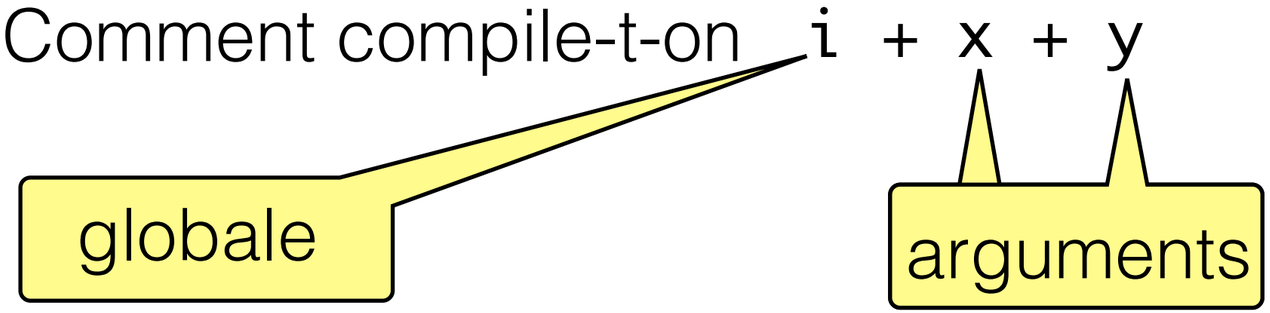
Lors de l'appel de la fonction, on place les valeurs des arguments sur la pile

`sum(5, 7)` est compilé en :

PUSH(5)	on empile les arguments
PUSH(7)	
GSB(adr(sum))	corps de la fonction
POP	on dépile les arguments
POP	
PRX	on met le résultat sur la pile

Compilation 3.5 : le corps de la fonction

```
static int sum(int x, int y) {
  return(i + x + y);
}
```



PUSH(adr(i))
READ
} `i` en haut de la pile

FETCH(3) `x` en haut de la pile

ADD `i+x` en haut de la pile

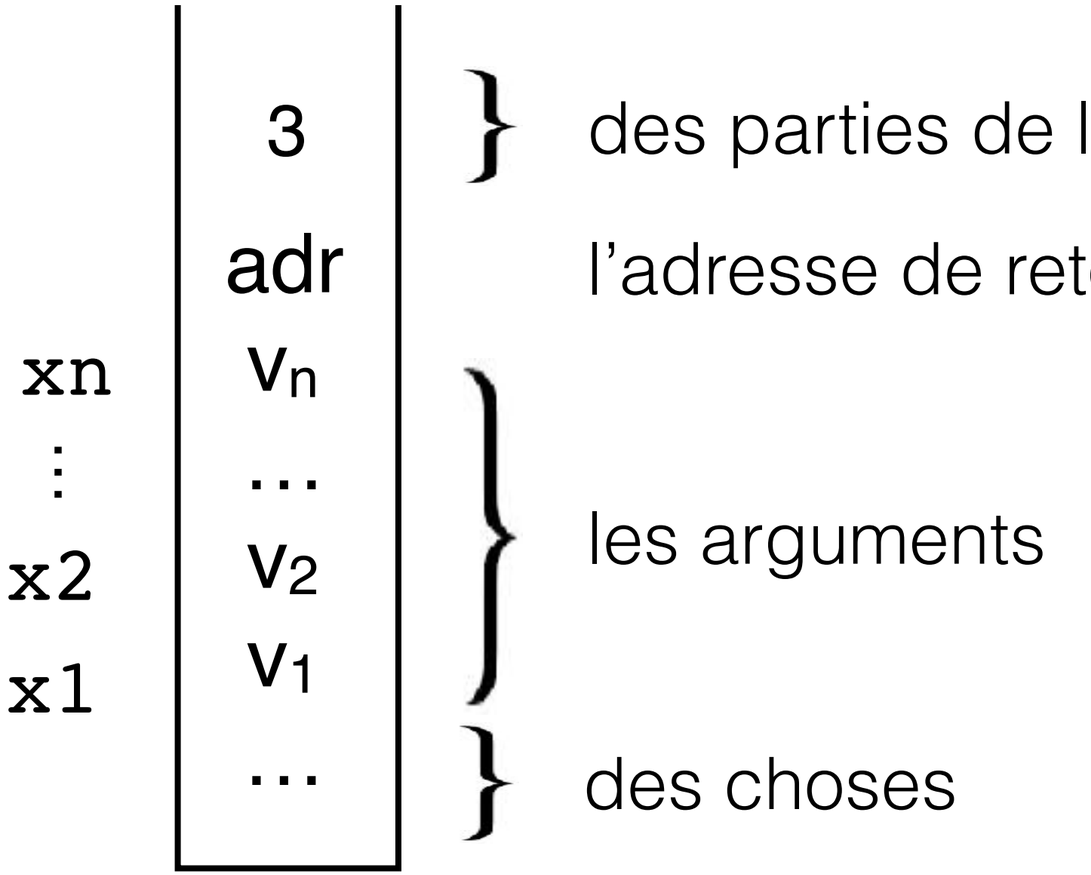
FETCH(2) `y` en haut de la pile

ADD `i+x+y` en haut de la pile

Plus en détail

La pile à l'intérieur du corps d'une fonction

```
f(int x1, int x2, … , int xn) { … 3 + x2 … }
```



expression	⟦x2⟧
3 + x2	FETCH(n-2)
x2	FETCH(n-1)
x2 + 3	FETCH(n-1)

Quels invariants sont corrects ?

A. l’exécution de $[\![p]\!]$ rend la pile inchangée
B. l’exécution de $[\![p]\!]$ laisse la taille de la pile inchangée
C. l’exécution de $[\![p]\!]$ peut augmenter la taille de la pile
D. l’exécution de $[\![p]\!]$ peut diminuer la taille de la pile

Les arguments sont-ils statiques ?

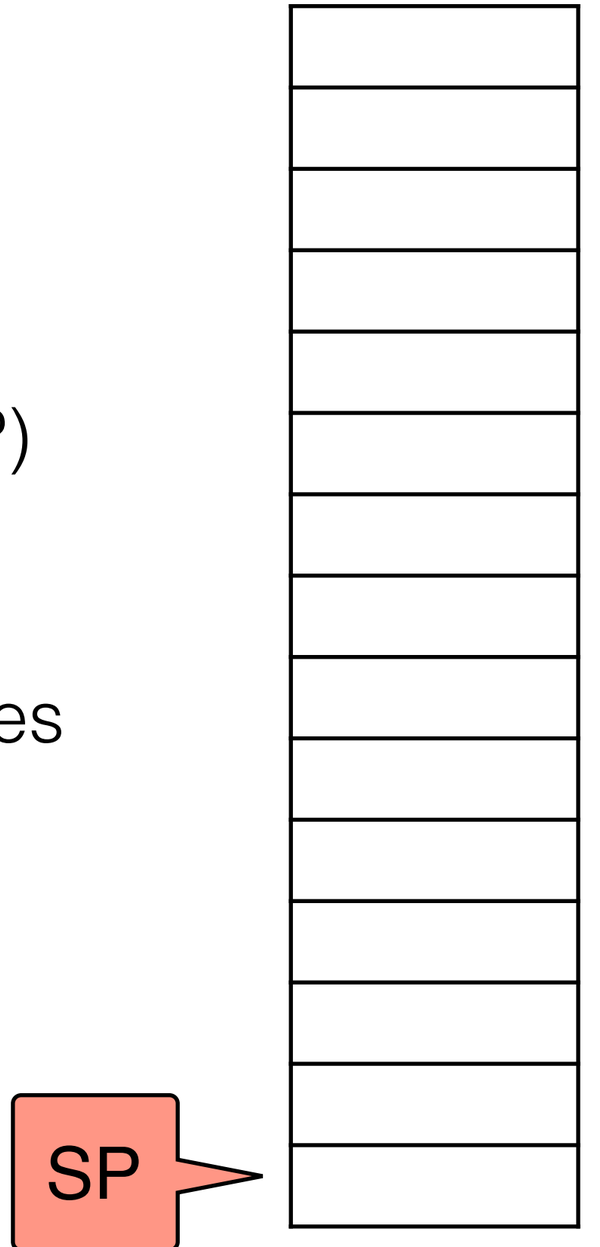
```
static int i;

static int f(int x) {

  return(i + x);

}
```

Dans la pile :
FETCH(2)

Presque statique: relatif
au compteur de pile (SP)

FETCH(2) = lire 2 adresses
avant SP

complètement statique :
l'emplacement mémoire est
choisi
une fois pour toute lors de
la compilation



Récursivité : deux mots

```
static int fact(int i) {
  if (i <= 1) return(1);
  return(i * fact(i - 1));
}
```

Ca marche

```
static int foo(int i) {
  return(foo(i));
}
```

Que ce passe-t-il ?

pour résumer

```
f(int x1, int x2, … , int xn) { … p … }
```

Dans le corps d'une fonction, il y a :

- des variables statiques (valeur stocké dans un emplacement statique de la mémoire)
- des arguments (valeurs stockés dans la pile)
- des variables locales :

```
int x = 3;
…
}
```

elles seront également rangées dans la pile.