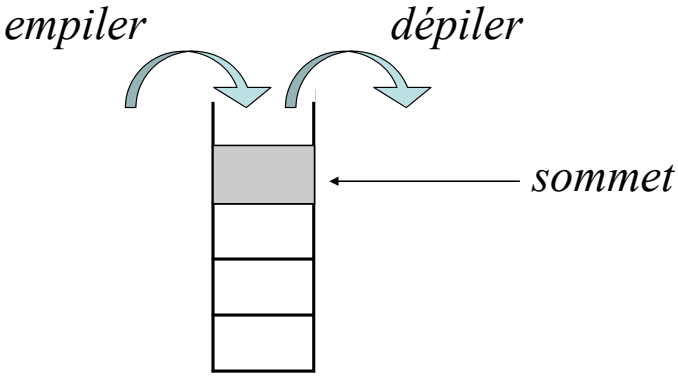
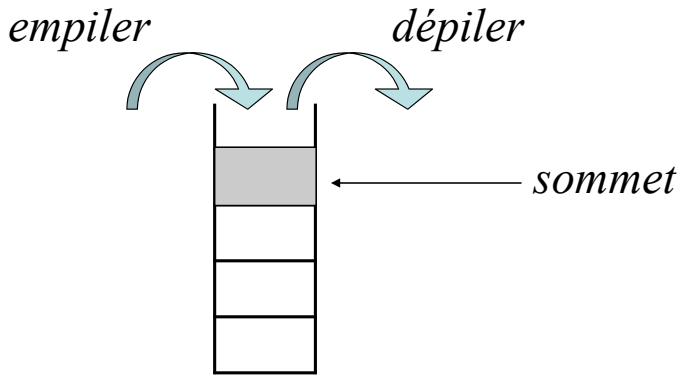
Université Nice Sophia Antipolis

Les piles et le retour sur trace (backtrack)



Le type abstrait PILE



- Les **piles** forment un type de donnée fonctionnant en ordre LIFO [Last In, First Out].

- Nos piles seront ***fonctionnelles***.

La fonction `(empiler x P)` ne modifie pas la pile `P`, elle se contente de calculer une nouvelle pile ! Toutes les opérations sont en $O(1)$.

`(pile-vide)`	;	*constructeur de pile vide*
`(pile-vide? P)`	;	*Pile* → *Boolean*
`(empiler x P)`	;	*Elément* × *Pile* → *Pile**
`(depiler P)`	;	*Pile** → *Pile*
`(sommet P)`	;	*Pile** → *Elément*

où `Pile*` *dénote l'ensemble des piles non vides.*

```
> (define P (empiler 3 (empiler 4 (empiler 5 (pile-vide)))))
> (sommet P)
3
> (sommet (depiler P))
4
> (sommet P)
3
```

• L'**implémentation du type abstrait** passe ici aussi par les listes, qui sont bien pratiques !

```
(define (pile-vide)
  empty)
```

$\varnothing \rightarrow Boolean$

```
(define (pile-vide? pile)
  (empty? pile))
```

$Pile \rightarrow Boolean$

```
(define (empiler x pile)
  (cons x pile))
```

$Elément \times Pile \rightarrow Pile^*$

```
(define (depiler pile)
  (if (empty? pile)
      (error "depiler : Pile vide !")
      (rest pile)))
```

$Pile^* \rightarrow Pile$

```
(define (sommet pile)
  (if (empty? pile)
      (error "sommet : Pile vide !")
      (first pile)))
```

$Pile^* \rightarrow Elément$

N.B. Le type abstrait est inclus dans le teachpack `valrose.rkt`

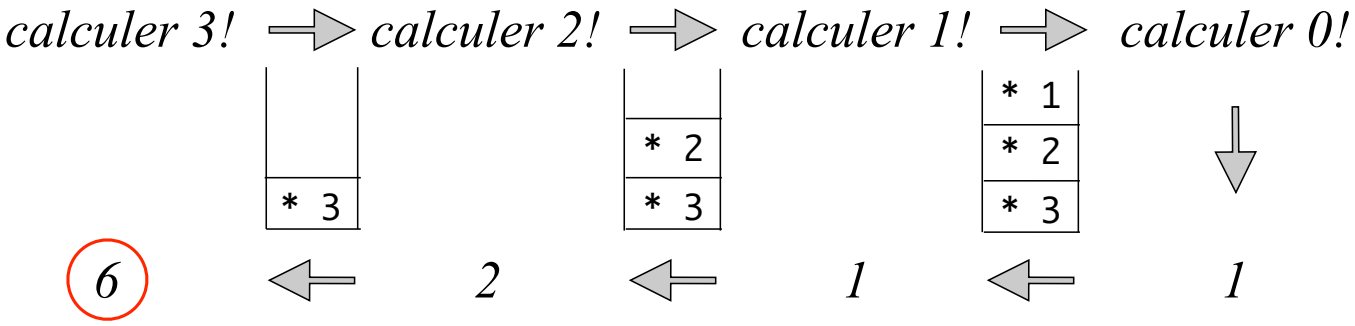
A quoi sert une PILE ?

• En électricité, à fournir de l'énergie. En cuisine, à ranger des assiettes. En programmation, à **stocker des données intermédiaires** qu'il faudra utiliser plus tard pour **revenir en arrière**.

• Par exemple, pour exécuter une ***récurrence enveloppée***, une pile implicite [non visible au programmeur] est utilisée par le système.

```
(fac n) = (* (fac (- n 1)) n)
```

Pour calculer n! , je calcule (n-1)! mais auparavant je mets la multiplication par n en attente dans une pile...



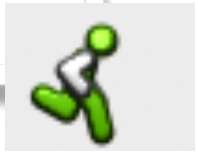
- Les ordinateurs sont des **machines à pile**. Lorsqu'un logiciel fonctionne sur votre machine, une **pile d'appel** est utilisée pour se souvenir du contexte d'appels.

- Python limite la taille de la pile d'appel : en Python, on ne peut pas faire des appels de fonctions imbriqués de profondeur arbitraire!

```
>>> def f(x): return f(x)
>>> f(0)
RecursionError: maximum recursion depth exceeded
```

- Cela pose moins de problème à Racket

```
(define (f x) (f x))
(f 0)
```



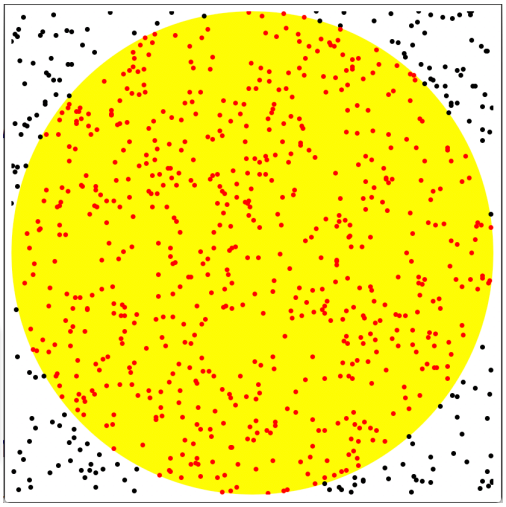
Revenir en arrière avec une pile : animation réversible

Exemple d'animation: calcul de pi par méthode de Monte-Carlo

On lance au hasard N flèches sur le carré de côté 2*R. On obtient B flèches noires (hors du cercle) et N-B flèches rouges.

Pourquoi faire cela? pour calculer pi!

P(flèche dans le disque)= pi/4 ≈ (N-B)/N



Réversibilité

On veut que le bouton -> lance une flèche tandis que le bouton <- permet de reprendre la dernière flèche lancée et d'effacer le dernier l'impact.

Revenir en arrière avec une pile : animation réversible

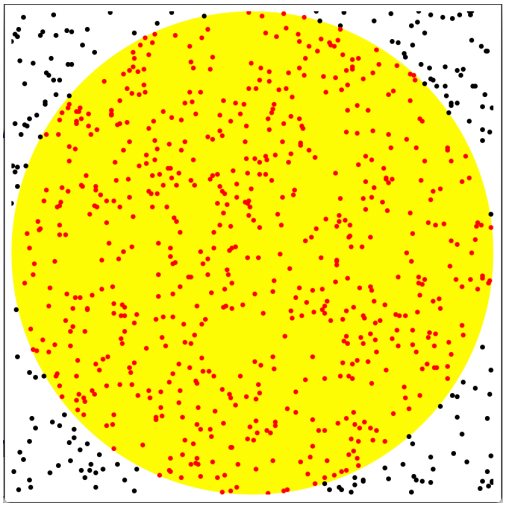
Les premiers éléments de dessin

```
(define R 200) ; le rayon du cercle

(define RECT (rectangle (* 2 R) (* 2 R) 'solid
'white))
(define CERC (circle R 'solid 'yellow))
(define SCENE (underlay RECT CERC))

(define IMPACT-IN (circle 2 'solid 'red))
(define IMPACT-OUT (circle 2 'solid 'black))
(define (distance-au-centre x y)
  (sqrt (+ (sqr (- x R)) (sqr (- y R)))))

(define (impact x y)
  (if (< (distance-au-centre x y) R) IMPACT-IN
IMPACT-OUT))
```

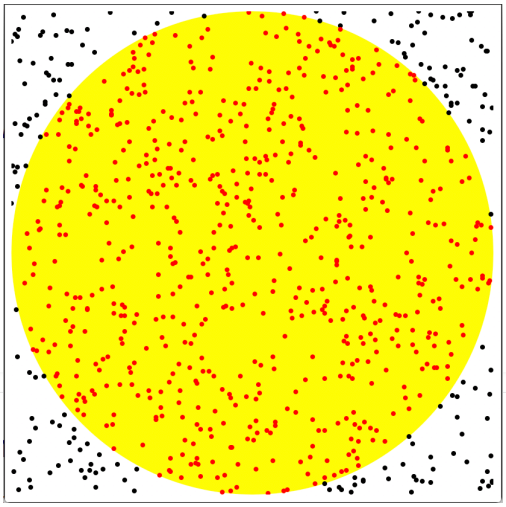


renvoie un point de la bonne couleur en fonction de sa position

Revenir en arrière avec une pile : animation réversible

```
; la structure position
(define-struct pos (x y))

; renvoie une position au hasard
(define (random-pos)
  (make-pos (random (* 2 R)) (random (*
2 R))))
```

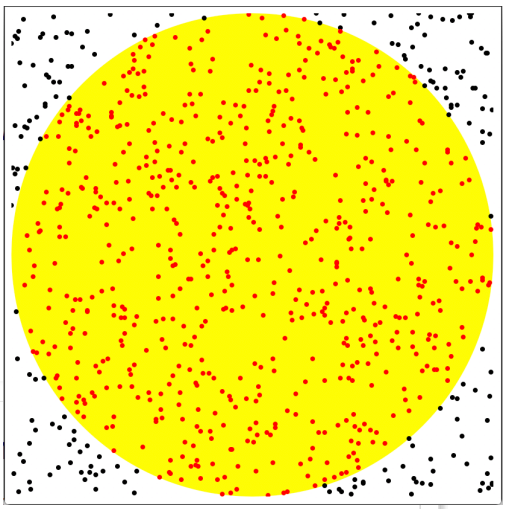


Le monde est une pile P de positions.

```
(define (dessine P)
 (if (pile-vide? P)
      SCENE
      (local
        [(define x (pos-x (sommet P)))
         (define y (pos-y (sommet P)))
         (define P2 (depiler P))]
        (place-image (impact x y) x y
                     (dessine P2)))))
```

Une animation réversible

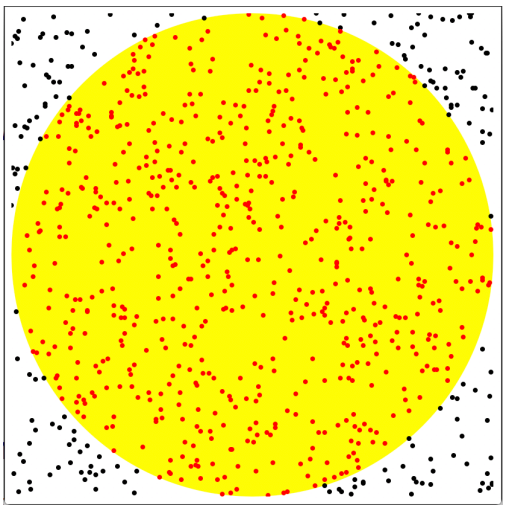
Reste la gestion du clavier et le big-bang…



```
(define (clavier P k)
  (cond
    [(key=? k "left") (depiler P)]
    [(key=? k "right") (empiler (random-pos) P)]
    [else P]))

(big-bang (pile-vide)
 (on-draw dessine)
 (on-key clavier))
```

Une animation vraiment réversible



Ce n'est pas une animation vraiment réversible, car:

si on presse <- puis ->, on va tirer une nouvelle flèche au hasard, et non remettre la flèche qu'on vient d'annuler.

On veut pouvoir annuler l'annulation, mais ce n'est pas ce qui se passe.

Comment faire? Réponse : le monde doit contenir deux piles:

- la pile des positions visibles
- la pile des positions annulées par <-

La suite en TD…

Autre application des piles : parcours itératif d'un arbre

• Les arbres binaires formant un type de donnée récursif, il est naturel de les programmer récursivement !

• Mais il peut être intéressant de produire un algorithme itératif.

• La présence de deux fils introduit des mises en attente. Munissons-nous donc d'**une pile en paramètre** et empilons-y les sous-arbres qu'il restera à visiter !

• Exemple : soit à calculer le **nombre de feuilles** d'un arbre A.

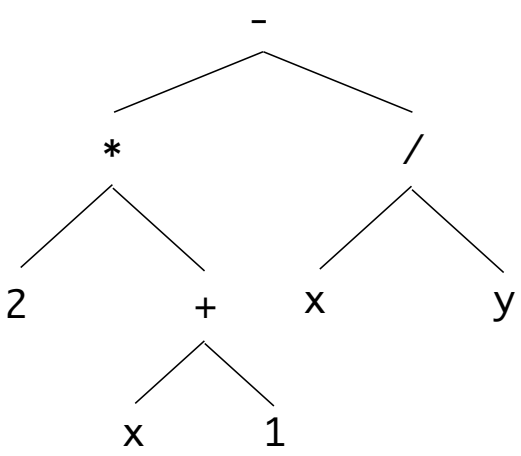
STRATEGIE DU PARCOURS EN PROFONDEUR ITERATIF :

- si je suis sur un noeud, j'empile [un pointeur sur] le fils droit et je vais visiter le fils gauche.

- si je suis sur une feuille, je regarde s'il reste un arbre à visiter au sommet de la pile.

```
(define (nb-feuilles A)           ; le nombre de feuilles
  (local [(define (iter A P acc)  ; arbre, pile courante, résultat courants
            (begin (printf "A=~a P=~a acc=~a\n" A P acc)     ; pour le debug...
                   (if (feuille? A)
                     (if (pile-vide? P)
                         (+ acc 1)
                         (iter (sommet P) (depiler P) (+ acc 1)))
                     (iter (fg A) (empiler (fd A) P) acc))))]
    (iter A (pile-vide) 0)))
```

```
> (nb-feuilles '(- (* 2 (+ x 1)) (/ x y)))
A=(- (* 2 (+ x 1)) (/ x y)) P=() acc=0
A=(* 2 (+ x 1)) P=((/ x y)) acc=0
A=2 P=((+ x 1) (/ x y)) acc=0
A=(+ x 1) P=((/ x y)) acc=1
A=x P=(1 (/ x y)) acc=1
A=1 P=((/ x y)) acc=2
A=(/ x y) P=() acc=3
A=x P=(y) acc=3
A=y P=() acc=4
==> 5
```



Parcours postfixe d'un arbre (notation « polonaise inversée »)

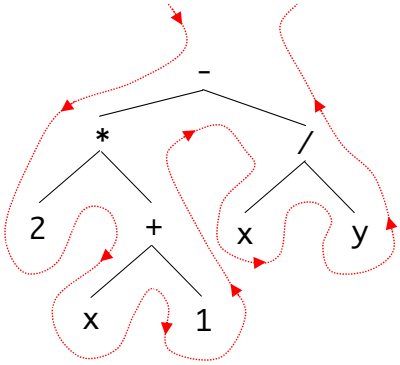
On a déjà vu le parcours **préfixe** et le parcours **infixe** : il nous reste à voir le parcours **postfixe**!

Dans le parcours préfixe, on visite l'opérateur **avant** les fils

(- * 2 + x 1 / x y)

Dans le parcours infixe, on visite l'opérateur **entre** les deux fils

(2 * x + 1 - x / y)



(2 x 1 + * x y / -)

Dans le parcours postfixe, on visite l'opérateur **après** les fils

Attention, le parcours postfixe n'est pas le parcours infixe à l'envers: comparer (- * 2 + x 1 / x y) et (2 x 1 + * x y / -)

Evaluer une expression en parcours postfixe

Plus facile qu'on pourrait le penser: l'ordre du parcours postfixe correspond à l'ordre d'évaluation dans l'arbre :
1) fils gauche, 2) fils droit, 3) racine

***RAPPEL : STRATEGIE D'EVALUATION POUR UN* ARBRE**

- si je suis sur une feuille, j'ai fini

- si je suis sur un noeud, :

1) j'évalue mon fils gauche

2) j'évalue mon fils droit,

3) je combine les deux résultats selon l'opération à ma racine.

```
(define (valeur A)   ; RAPPEL! (NB: l'arbre A est arithmétique)
  (if (feuille? A)
      A
      (local [(define vg (valeur (fg A)))
              (define vd (valeur (fd A)))]
        ((op->fun (racine A)) vg vd))))
```

Evaluer une expression en parcours postfixe

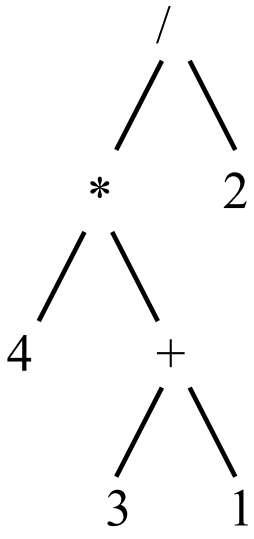
On va faire la même chose directement sur le parcours postfixe, en utilisant une pile pour se souvenir des valeurs
déjà calculées

```
(iter '() '(4 3 1 + * 2 /))
(iter '(4) '(3 1 + * 2 /))
(iter '(4 3) '(1 + * 2 /))
(iter '(4 3 1) '(+ * 2 /))
(iter '(4 4) '(* 2 /))
(iter '(16) '(2 /))
(iter '(16 2) '(/))
(iter '(8) '())
résultat = 8
```

Au début la pile est vide

Quand un nombre arrive, on l'empile

Quand un opérateur arrive, on dépile les deux premiers nombres, on les compose, et on empile le résultat



Evaluer une expression en parcours postfixe

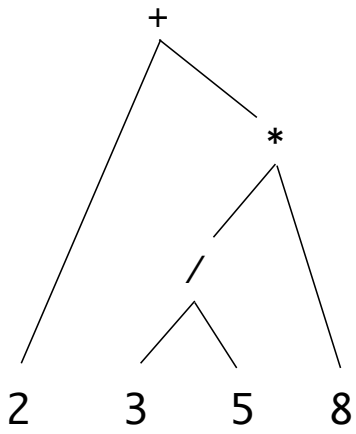
```
(define (postfixe->valeur L)
; L est le parcours postfixe d'un arbre arithmétique
  (local
     [(define (iter P L)
         (cond
           [(empty? L) (if (pile-vide? (depiler P))
                           (sommet P)
                           (error "pas un parcours postfixe!"))]
           [(number? (first L))
                (iter (empiler (first L) P) (rest L))]
           [else (local
               [(define n (sommet P)) ;échoue si pas parcours suff
                (define m (sommet (depiler P)))
                (define res ((op->fun (first L)) n m))]
               (iter (empiler res (depiler (depiler P)))
                     (rest L)))]))]
    (iter pile-vide L)))
```

• Exemple d'**analyse syntaxique itérative** : reconstruction d'un arbre à partir de son parcours postfixe

On analyse itérativement le parcours postfixe :

(2 3 5 / 8 * +)

cf TD !

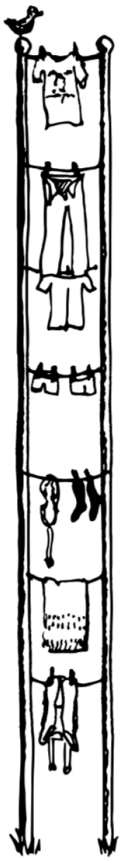


et on reconstruit l'arbre d'origine :

(+ 2 (* (/ 3 5) 8))

• Vous visualisez la **pile** ?...

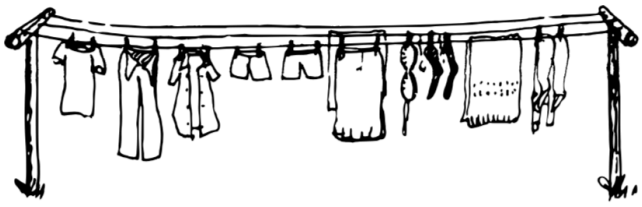
Les langages à piles



Certains langages de programmation rudimentaires fonctionnent sur le même principe:

- un programme est une suite de symbole
- certains sont des valeurs, d'autres des instructions de pile , par exemple
  - DUP : duplique le sommet de pile,
  - SWAP: échange les deux premiers éléments de pile
  - etc.

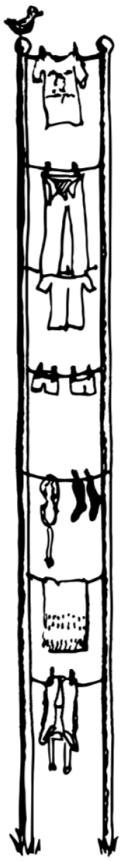
Un exemple dans le langage FORTH (http://thinking-forth.sourceforge.net):



```
2 3 DUP * SWAP - .
```

affiche 7 (a calculé 3*3-2)

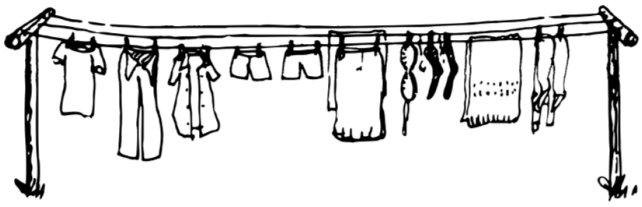
Les langages à piles



« Encore un langage de programmation que personne n'utilise et qui sert uniquement à écrire des thèses… »

Pourtant non, plusieurs langages à piles jouent un rôle important dans la vie de tous les jours:

- POSTSCRIPT : le langage des imprimantes
-Bitcoin/Ethereum « smart contracts »



ethereum