Concepts Informatiques

2019–2020

Matthieu Picantin

universiteouverte.org/2019/02/20/robert-gary-bobo-lanceur-dalerte-malgre-lui

blogs.mediapart.fr/paul-cassia/blog/131019/frais-d-inscription-des-etudiants-une-gratuite-couteuse

```
int res=1,cpt=2,arg=7;
while(cpt<=arg) res*=cpt++;
return res;
```

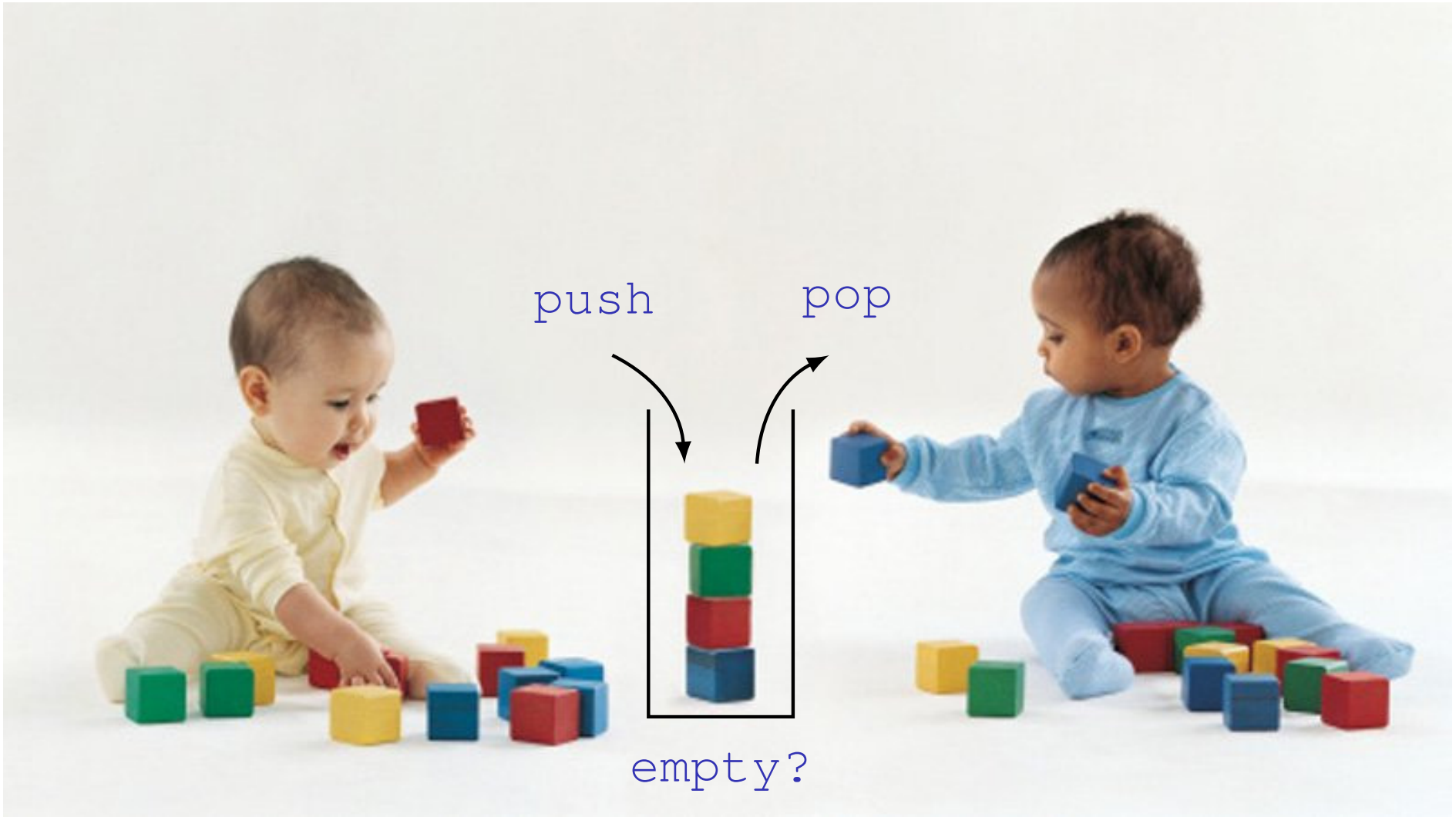
pensée

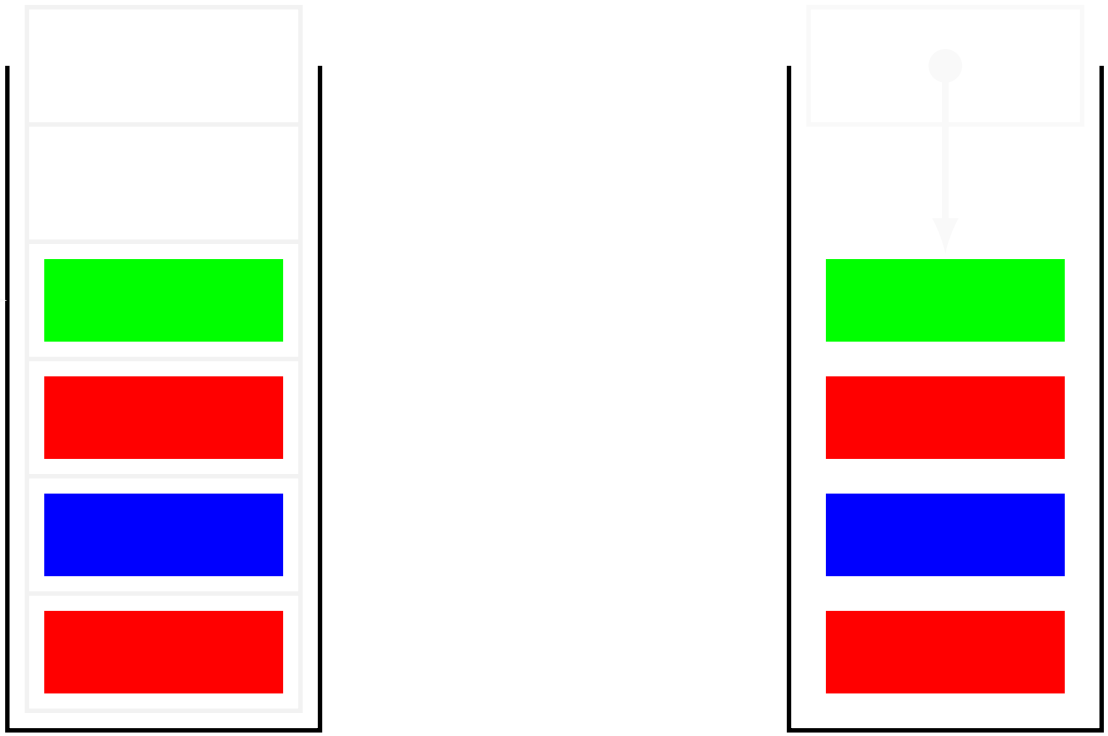
calcul
récursion
fonction
objet
⋮

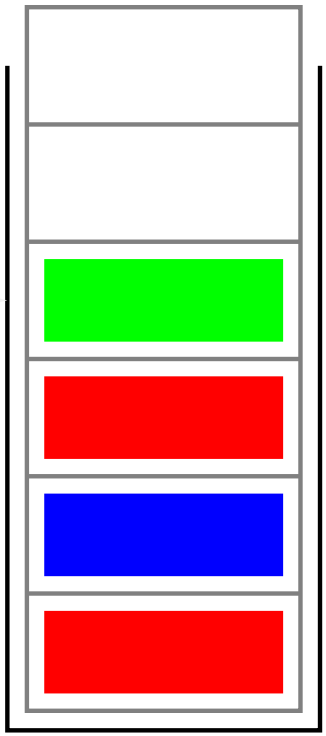
machine

circuit
pile
registre
mémoire
⋮

```
10111000 00000001 00000000
00000000 00000000 10111010
00000010 00000000 00000000
00000000 00111001 11011010
01111111 00000110 00001111
10101111 11000010 01000010
11101011 11110110 11000011
```

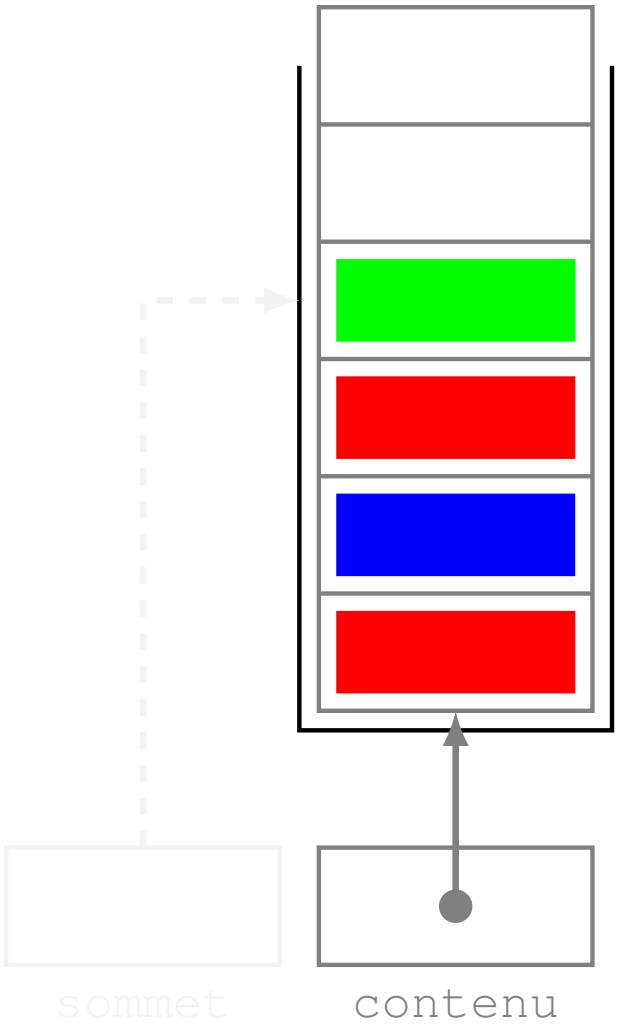

push
pop
empty?



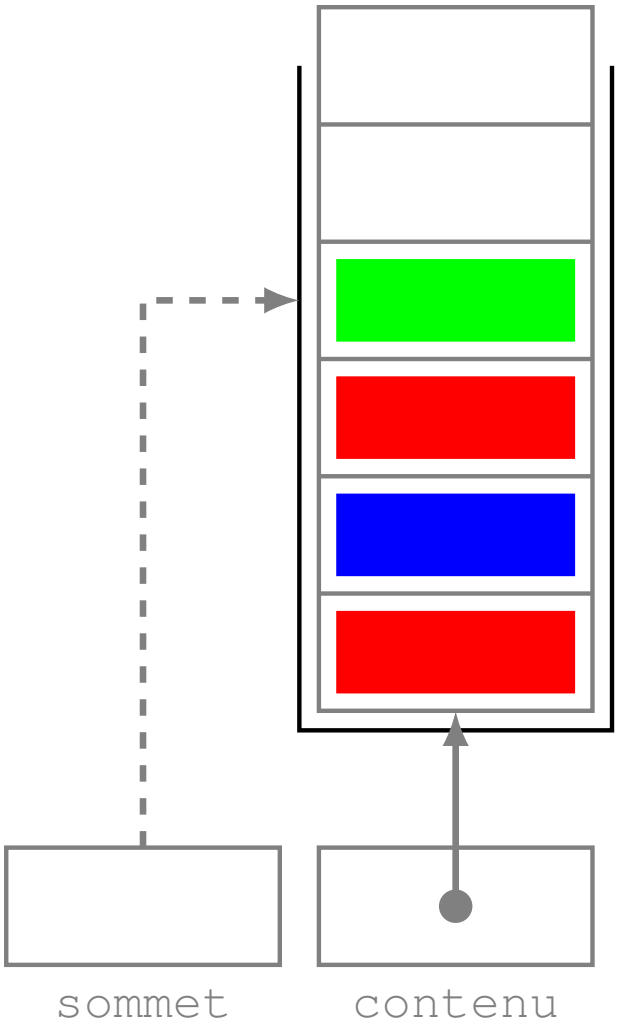


liste

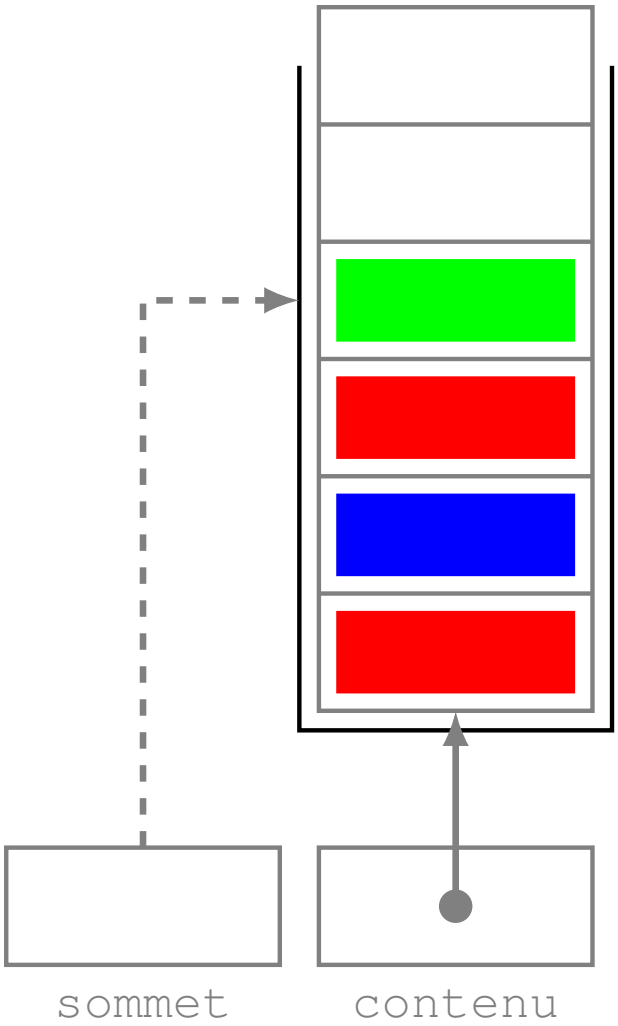
sommet
contenu

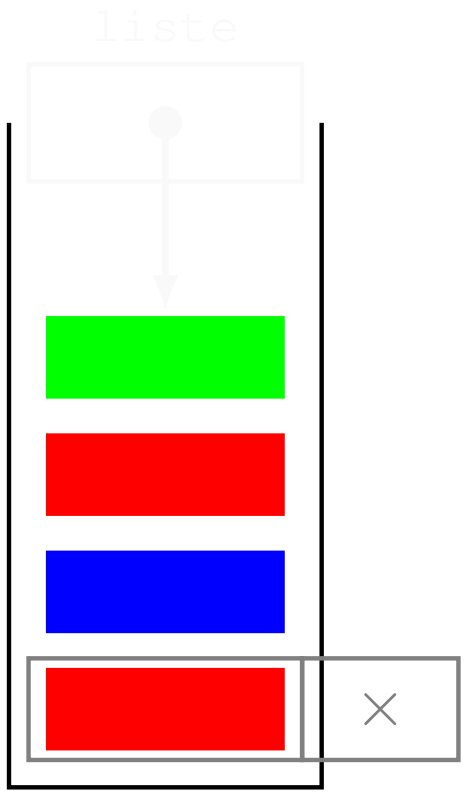

sommet
contenu

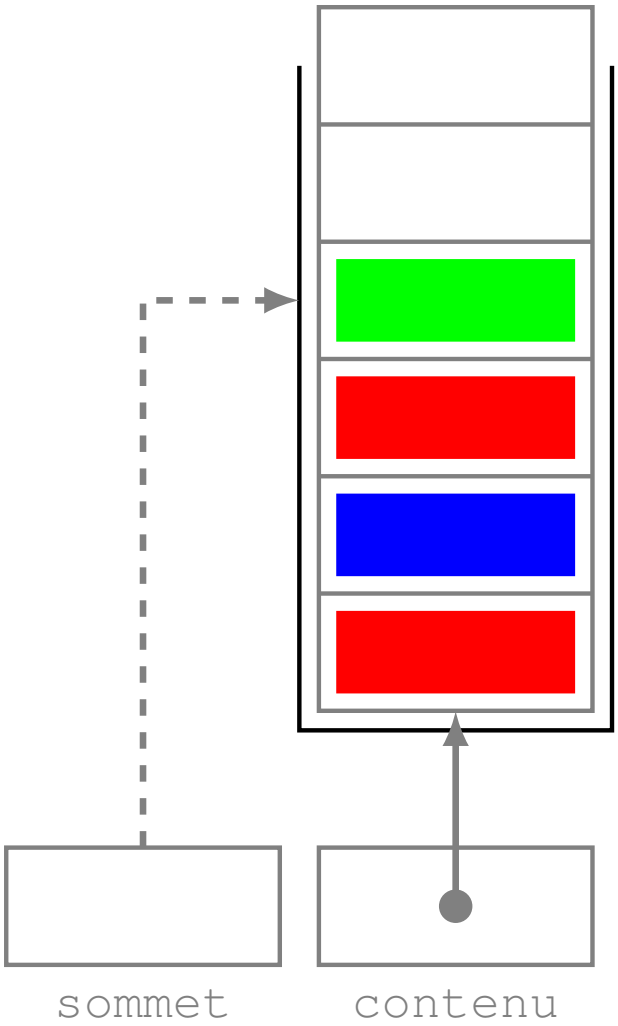
liste

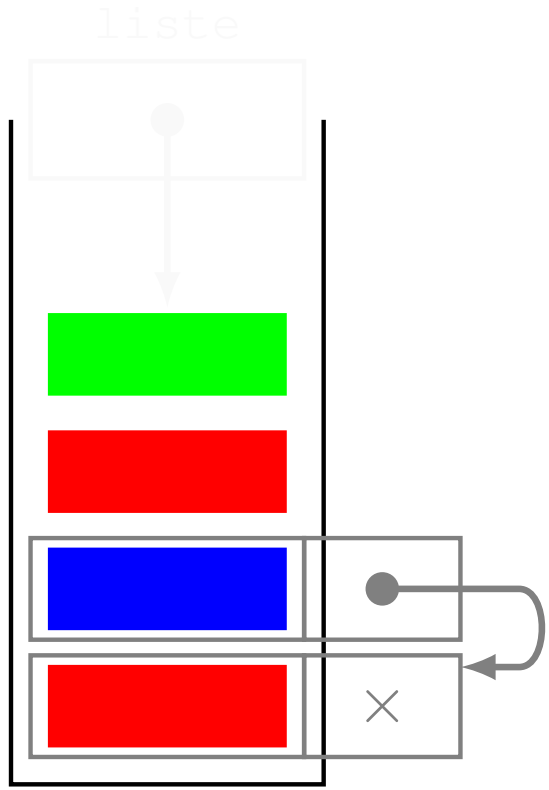

sommet
contenu

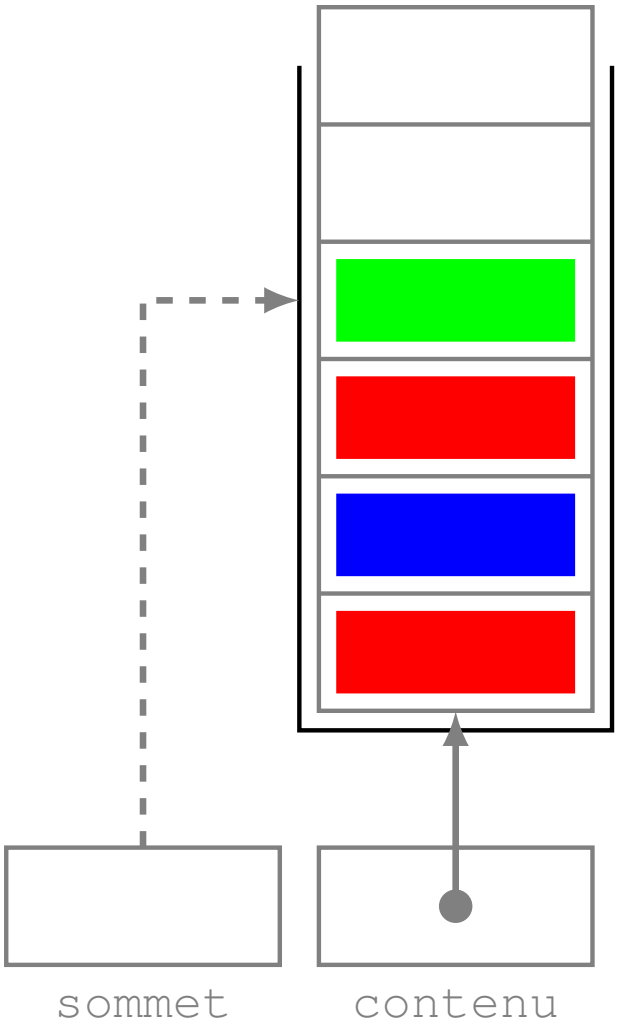
liste

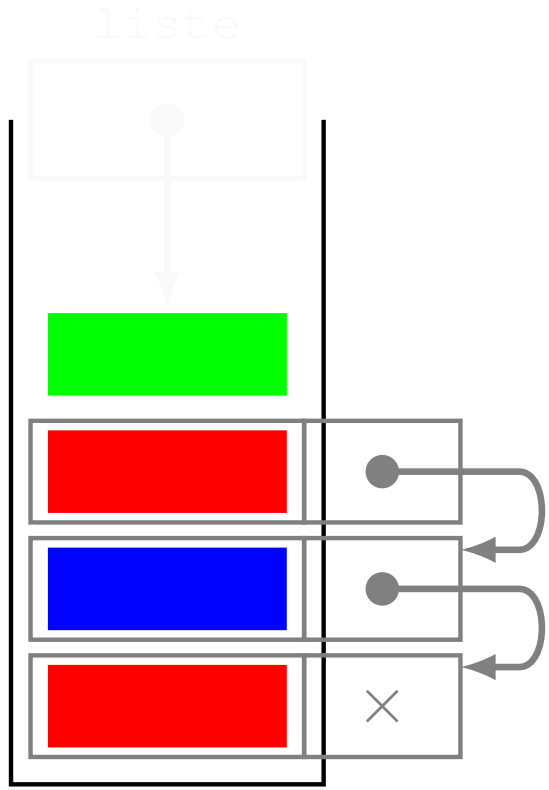

sommet
contenu

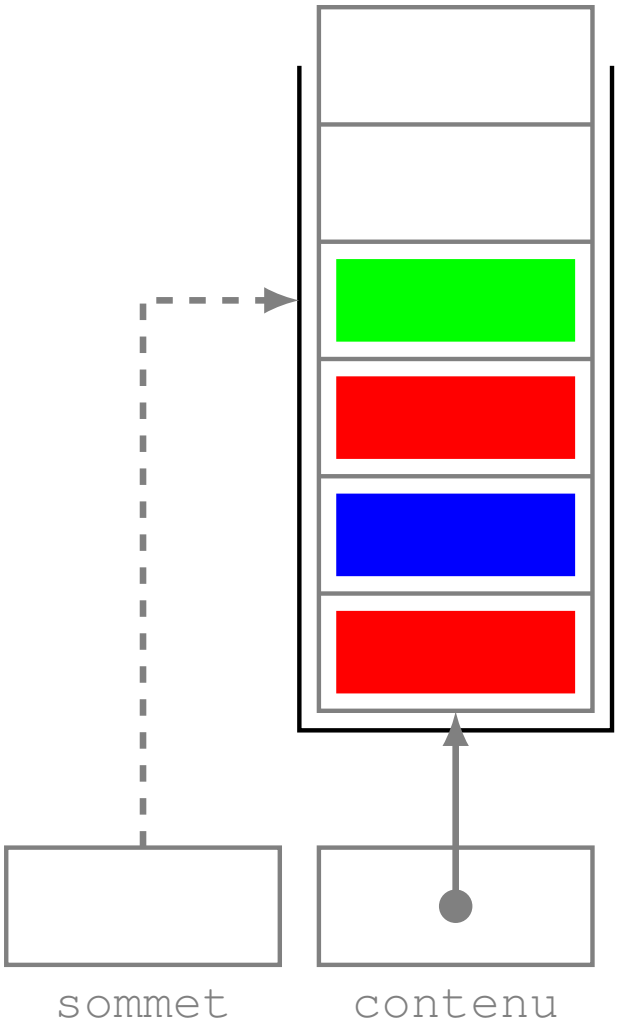

liste

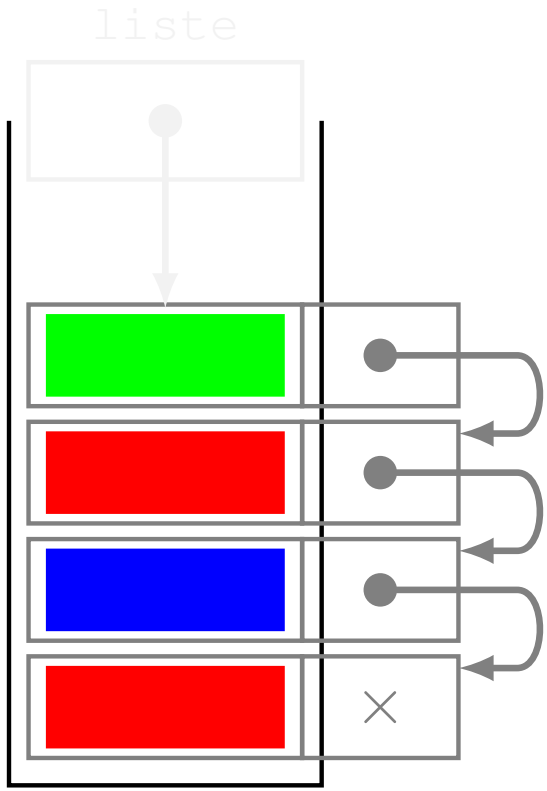

sommet
contenu

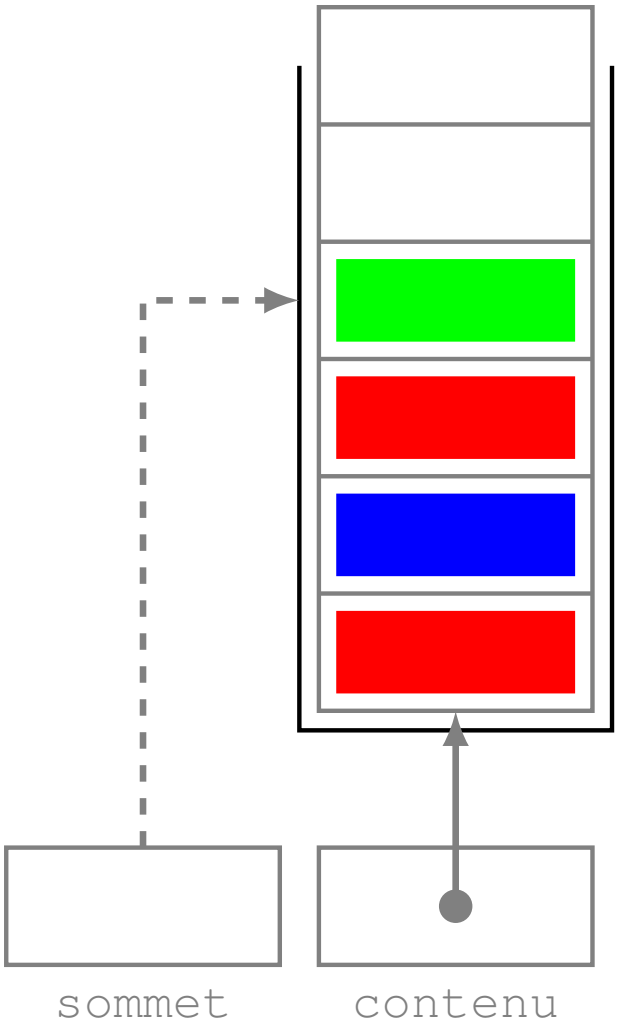

liste

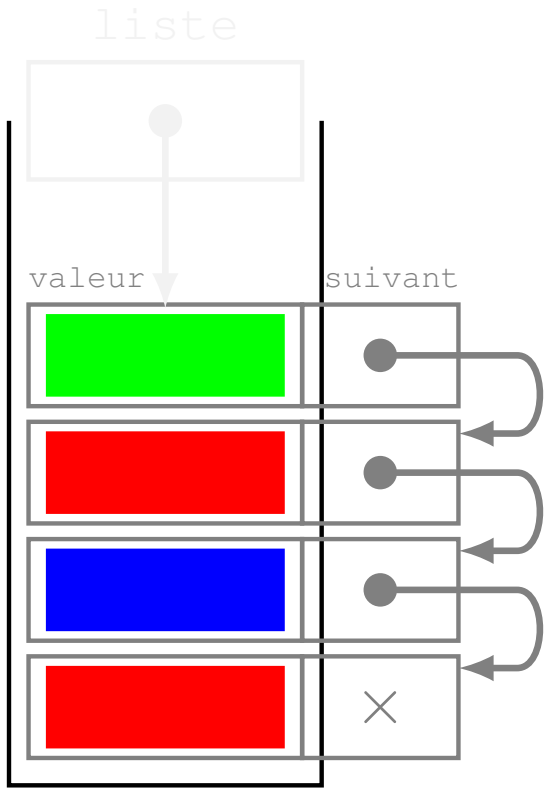

sommet
contenu

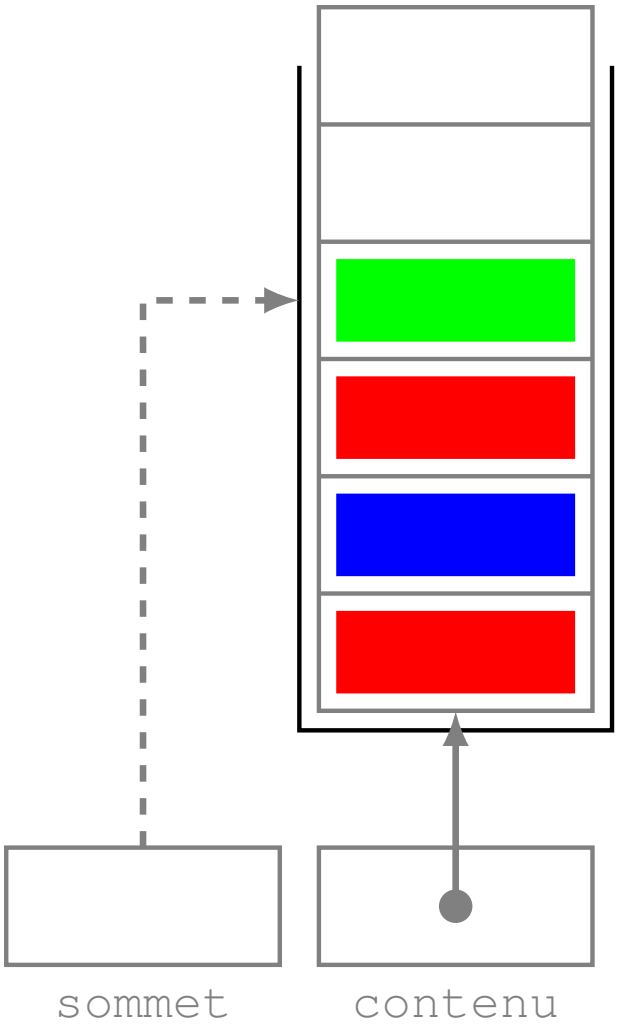

liste

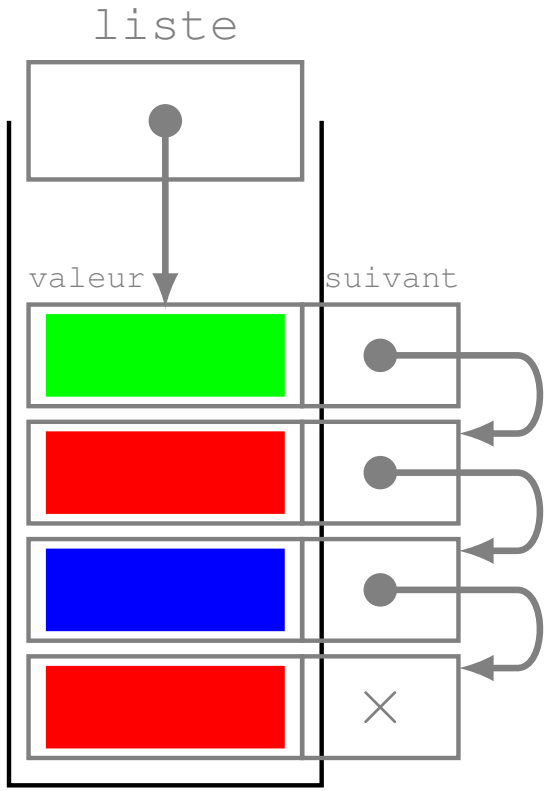

sommet
contenu

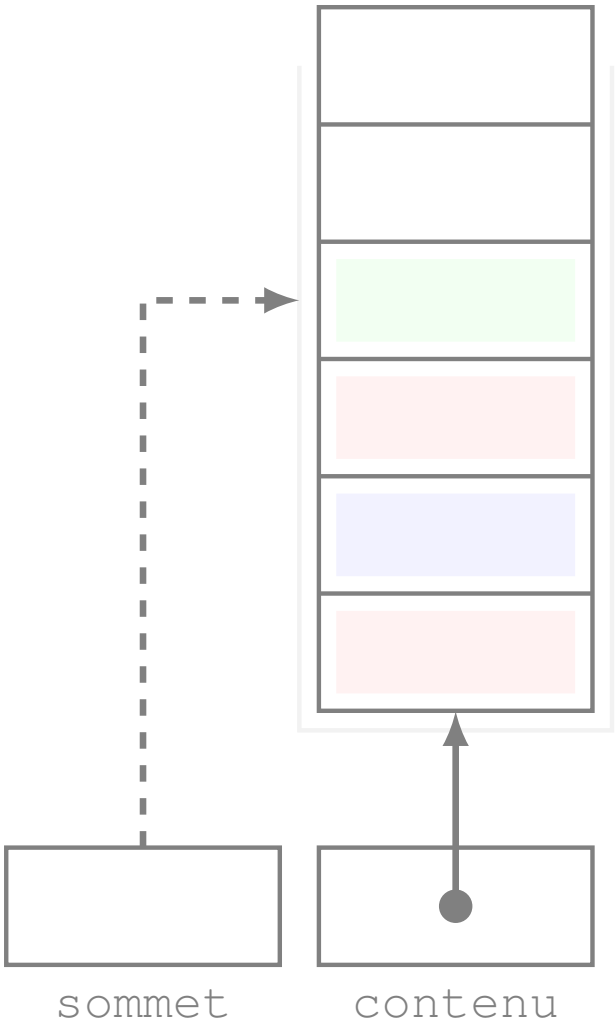

liste

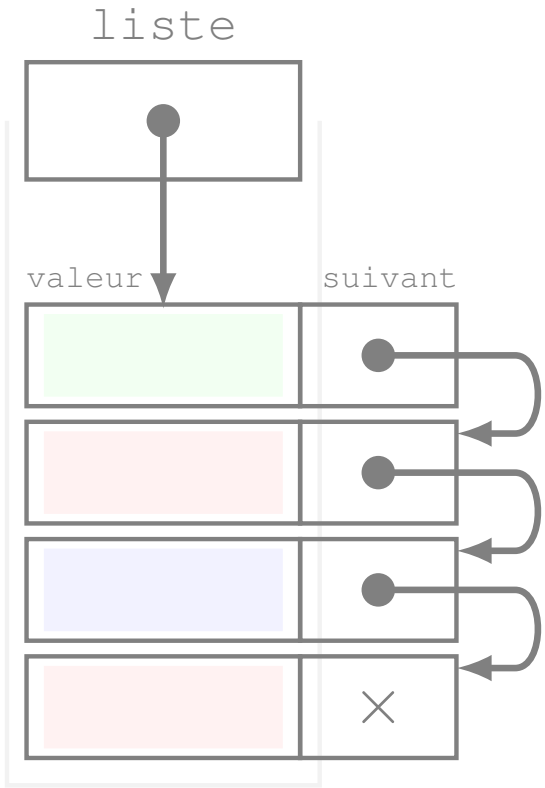

sommet
contenu

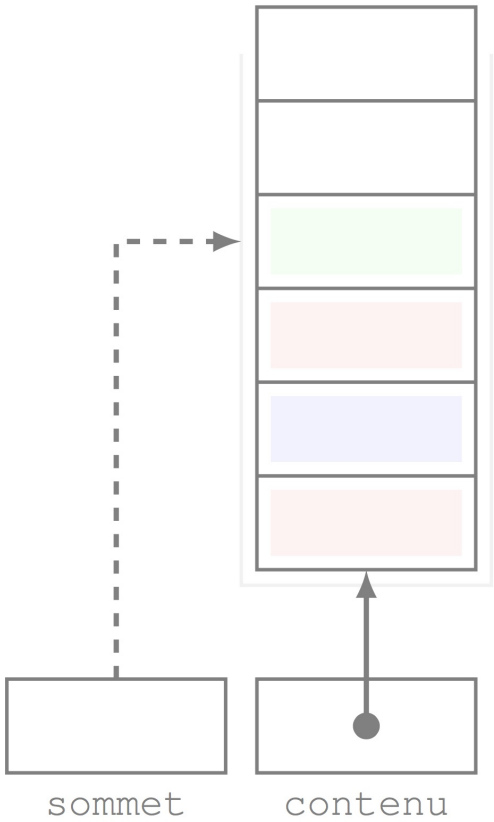

liste
valeur
suivant

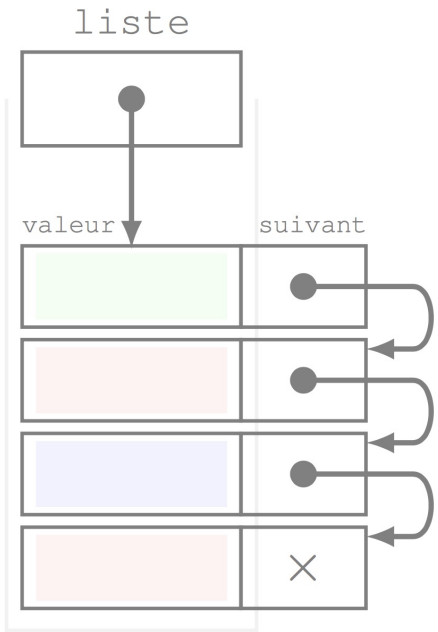

sommet
contenu


liste
valeur
suivant

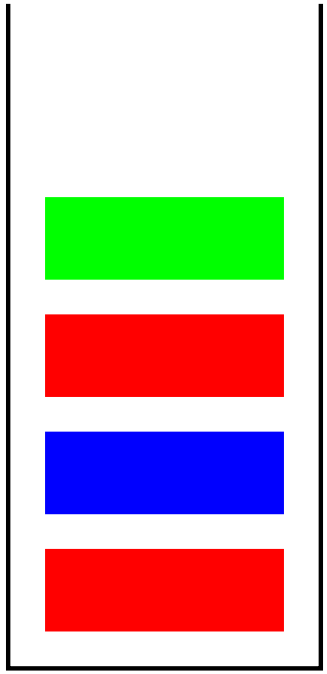

sommet
contenu

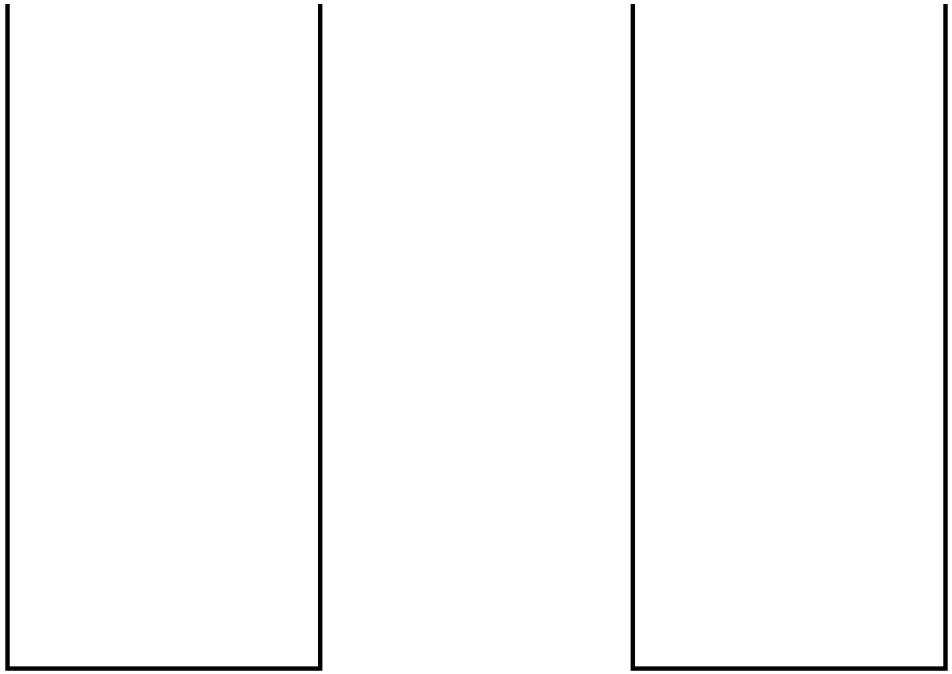

liste
valeur
suivant

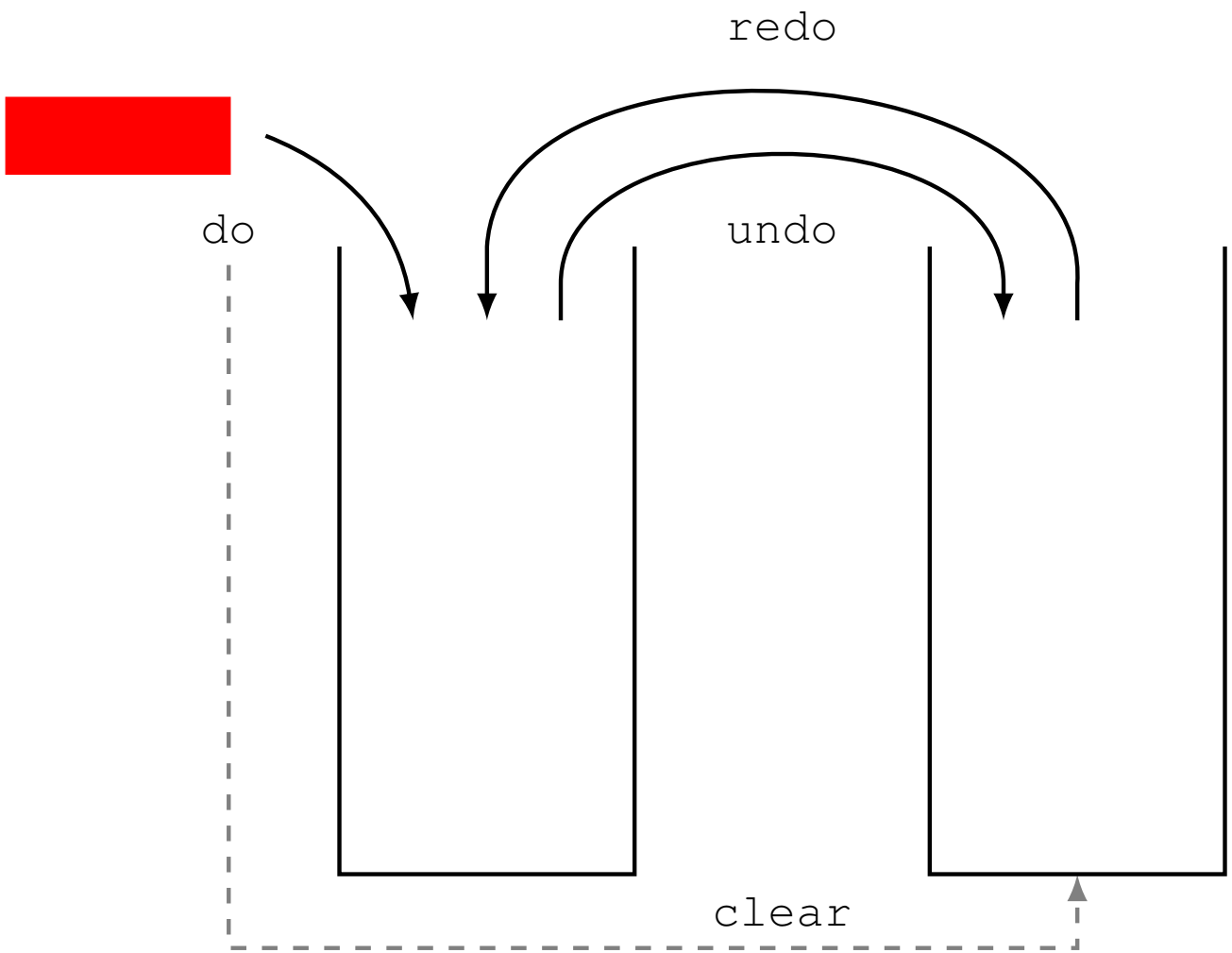

sommet
contenu

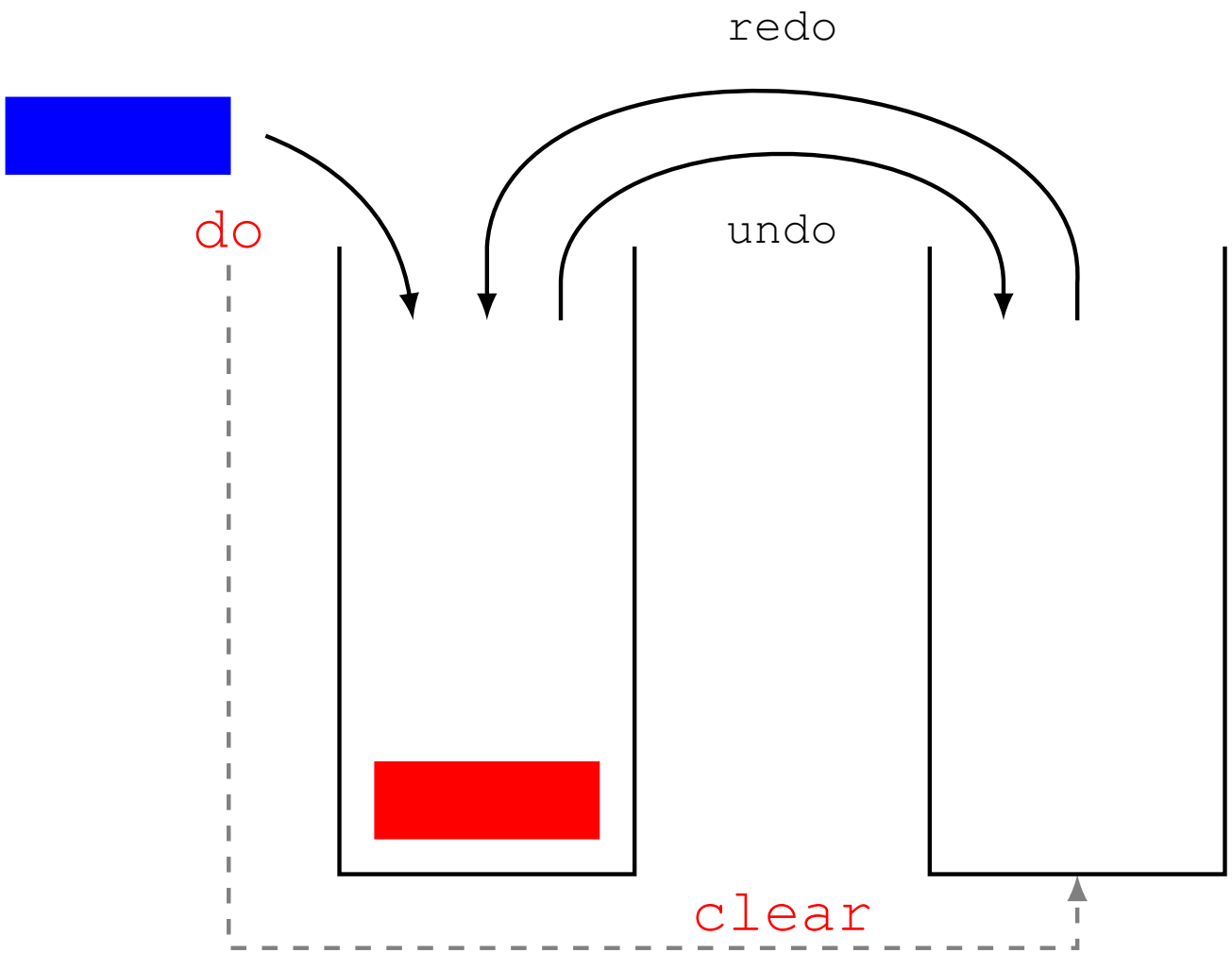

liste
valeur
suivant

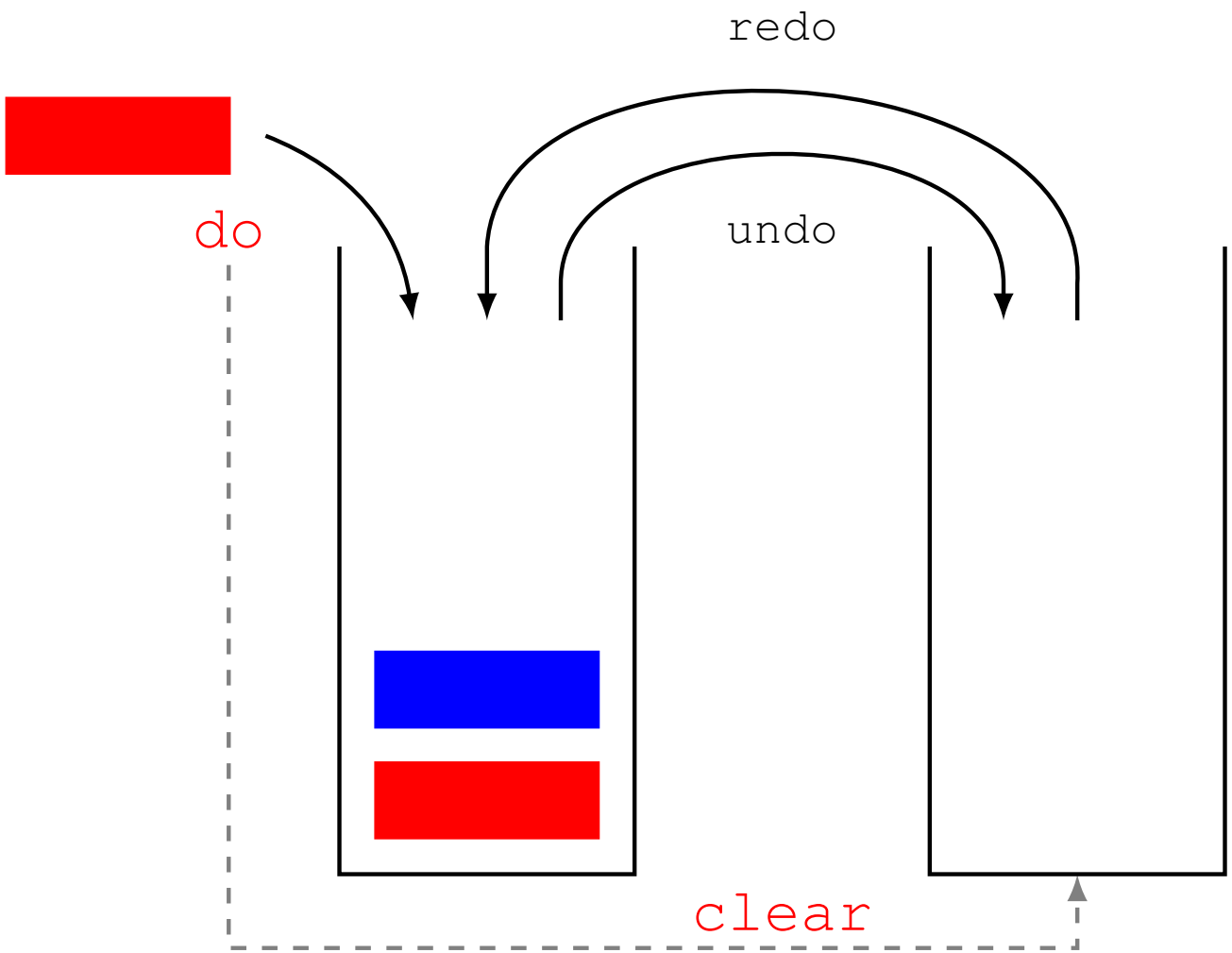

these implementations are not the ones you are looking for

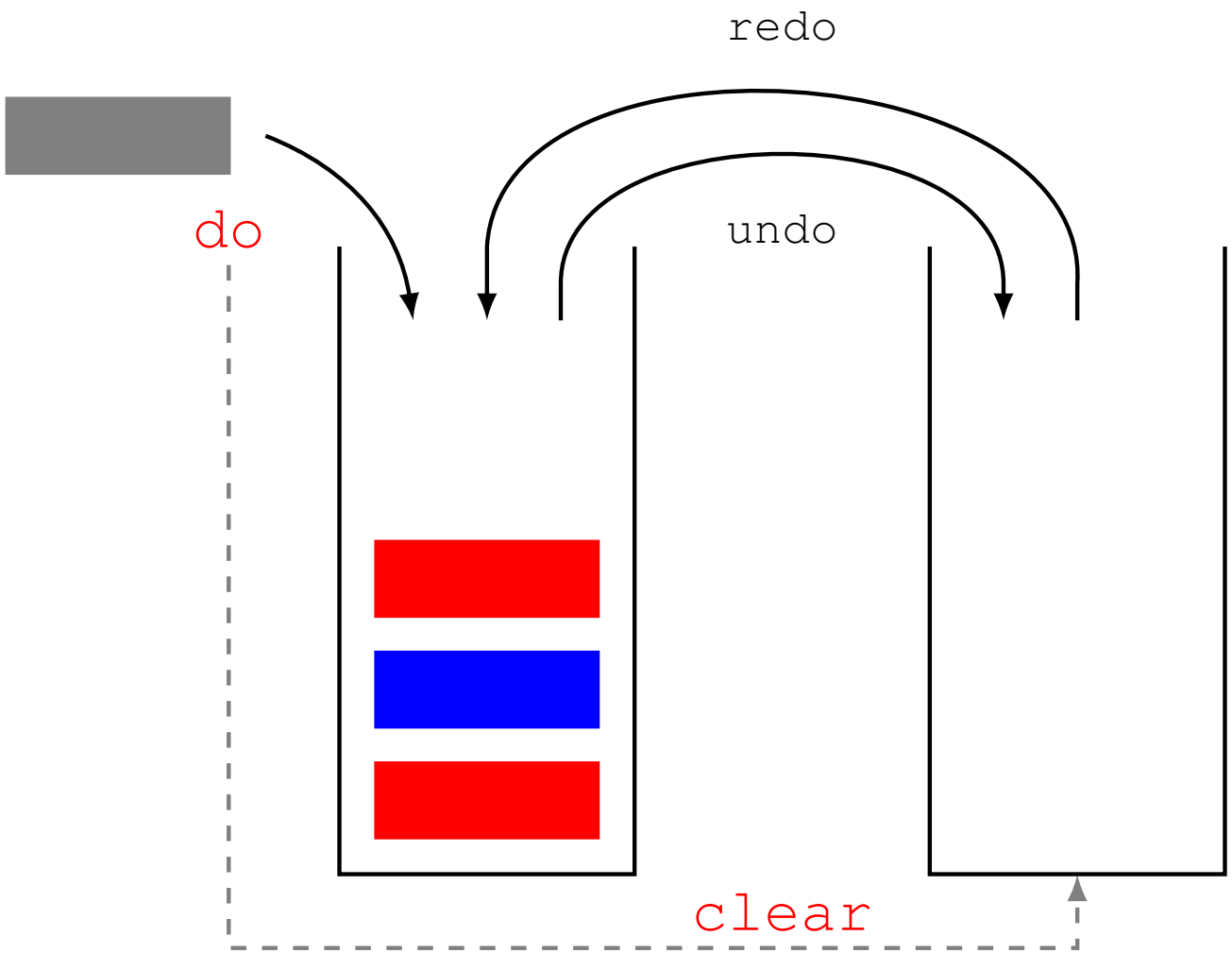


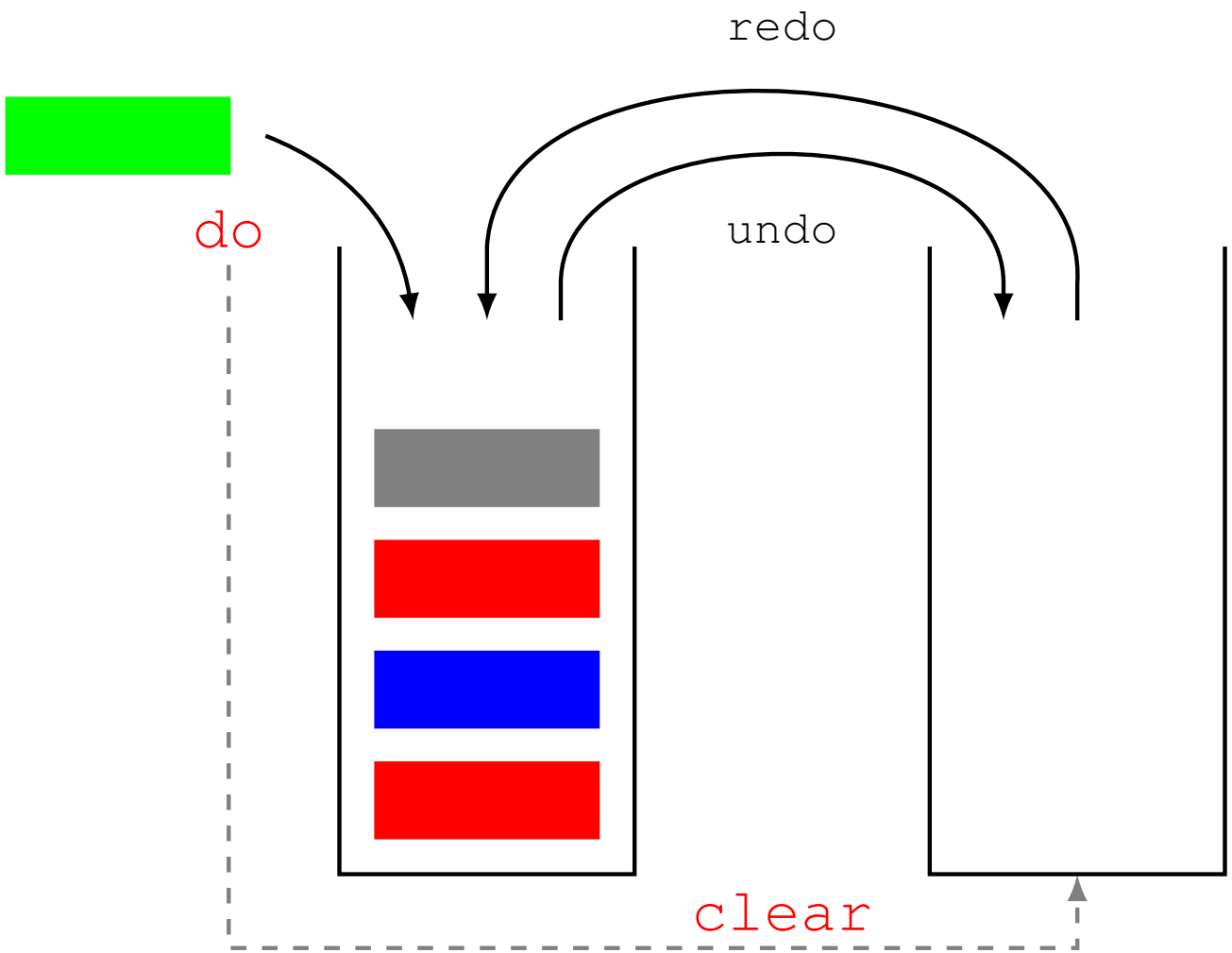


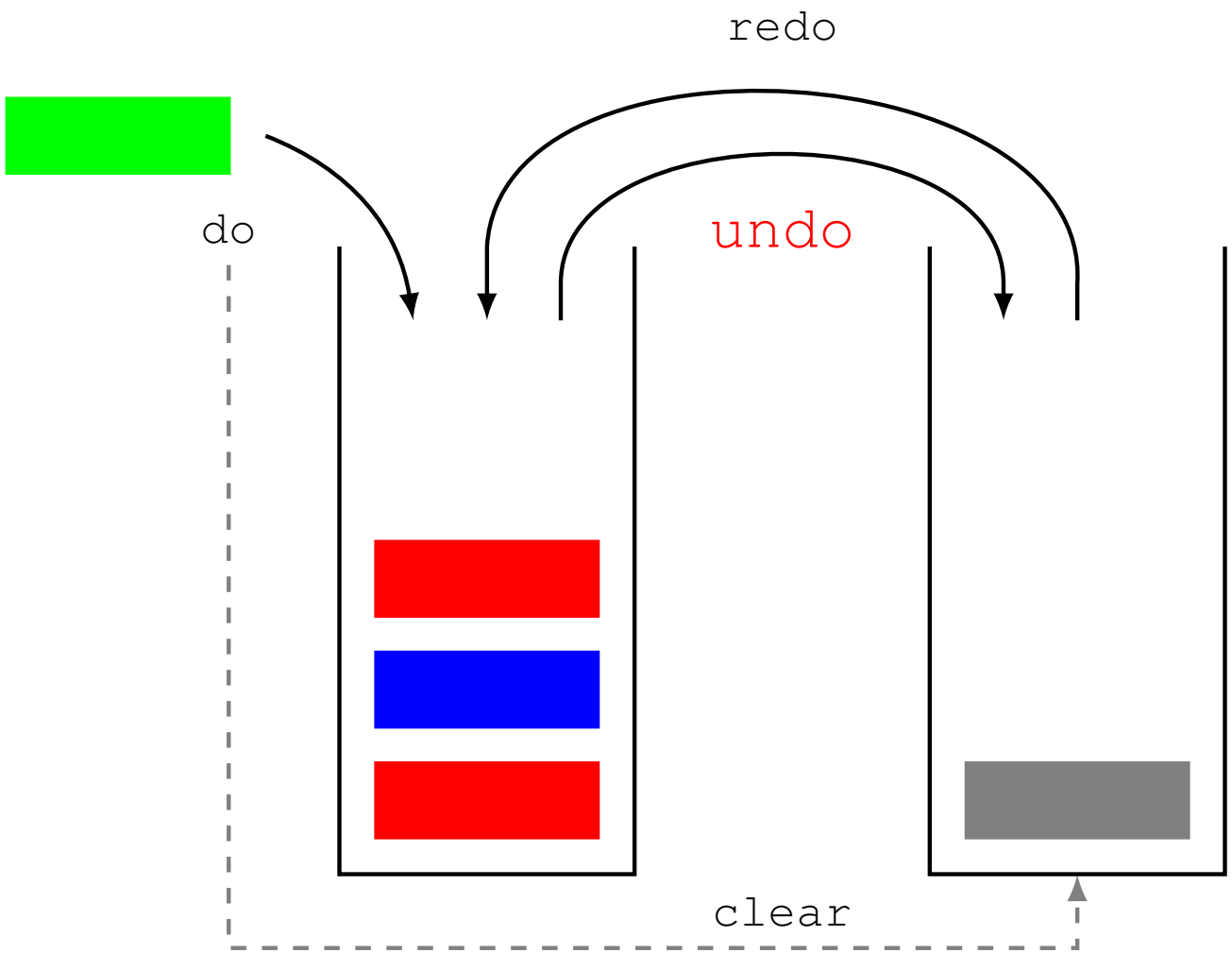

redo
do
undo
clear

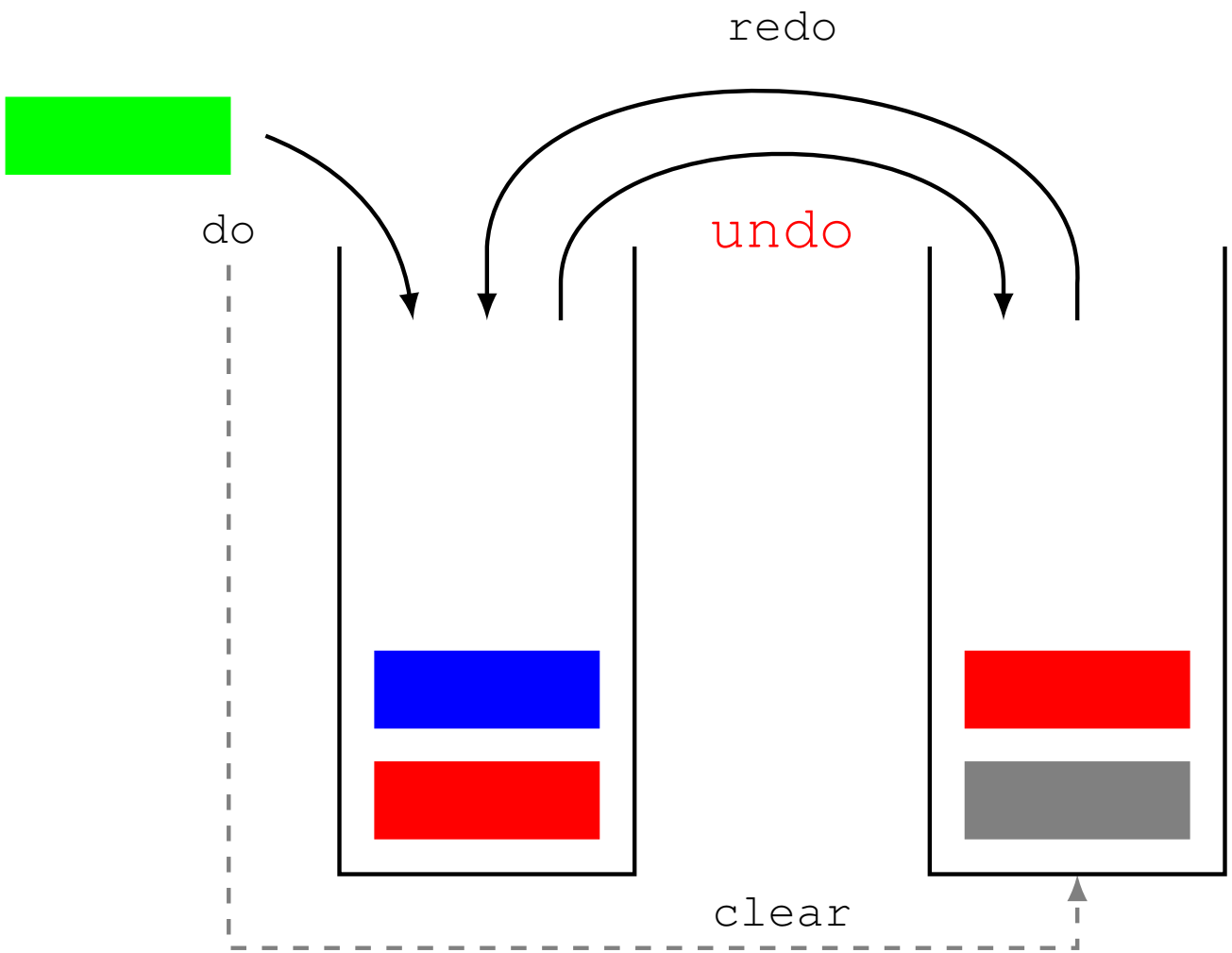

redo
do
undo
clear

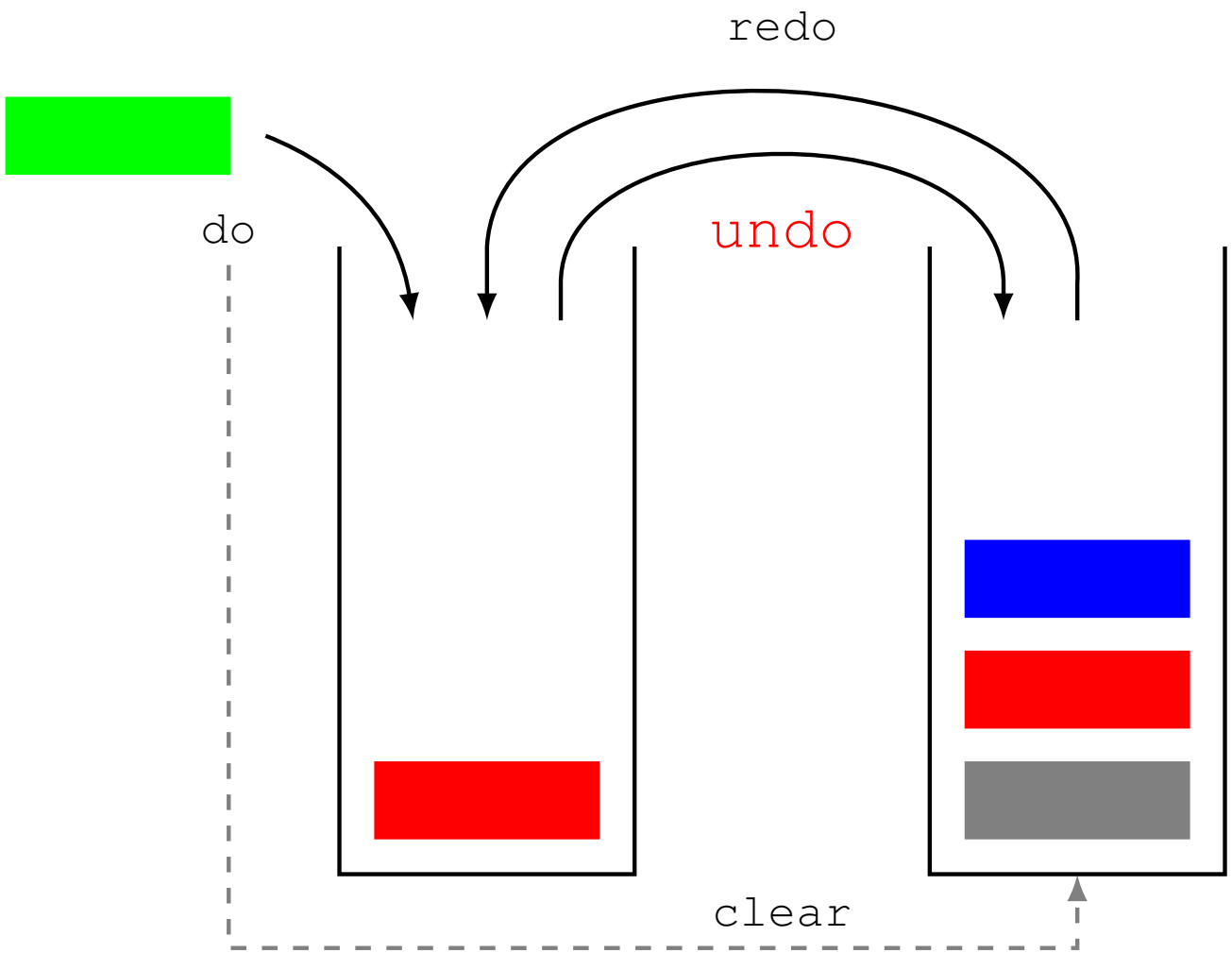

redo
do
undo
clear

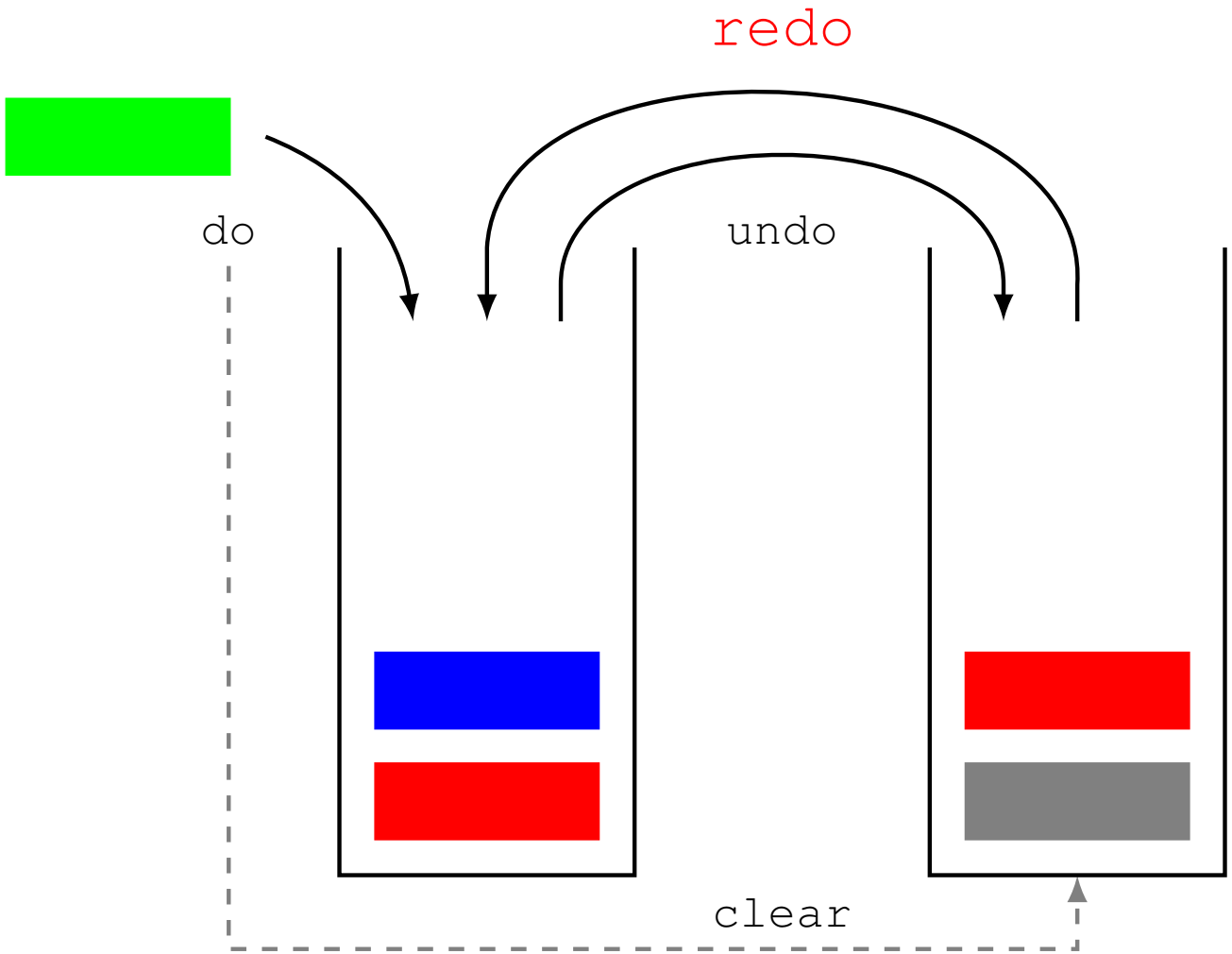

redo
do
undo
clear

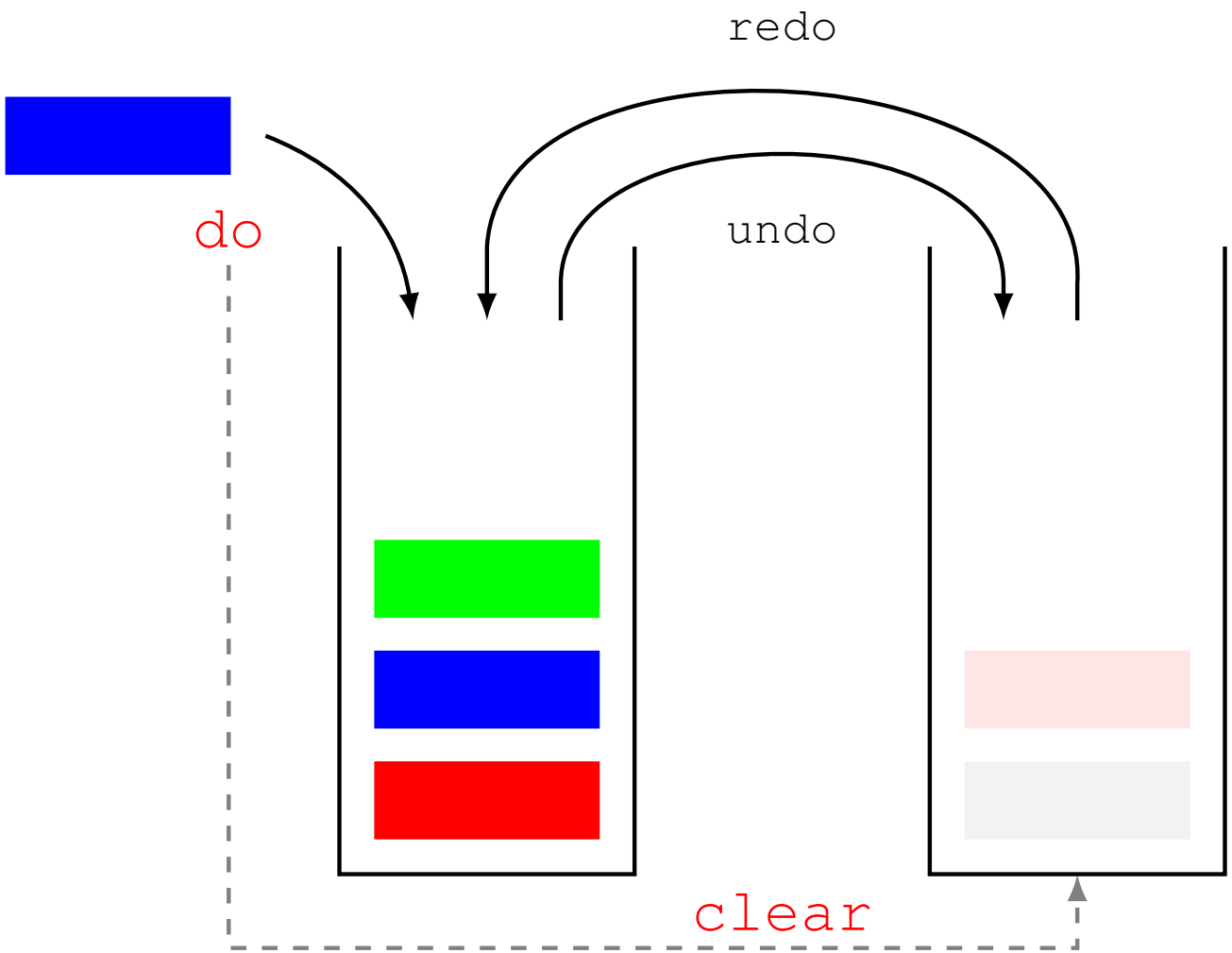

redo
do
undo
clear

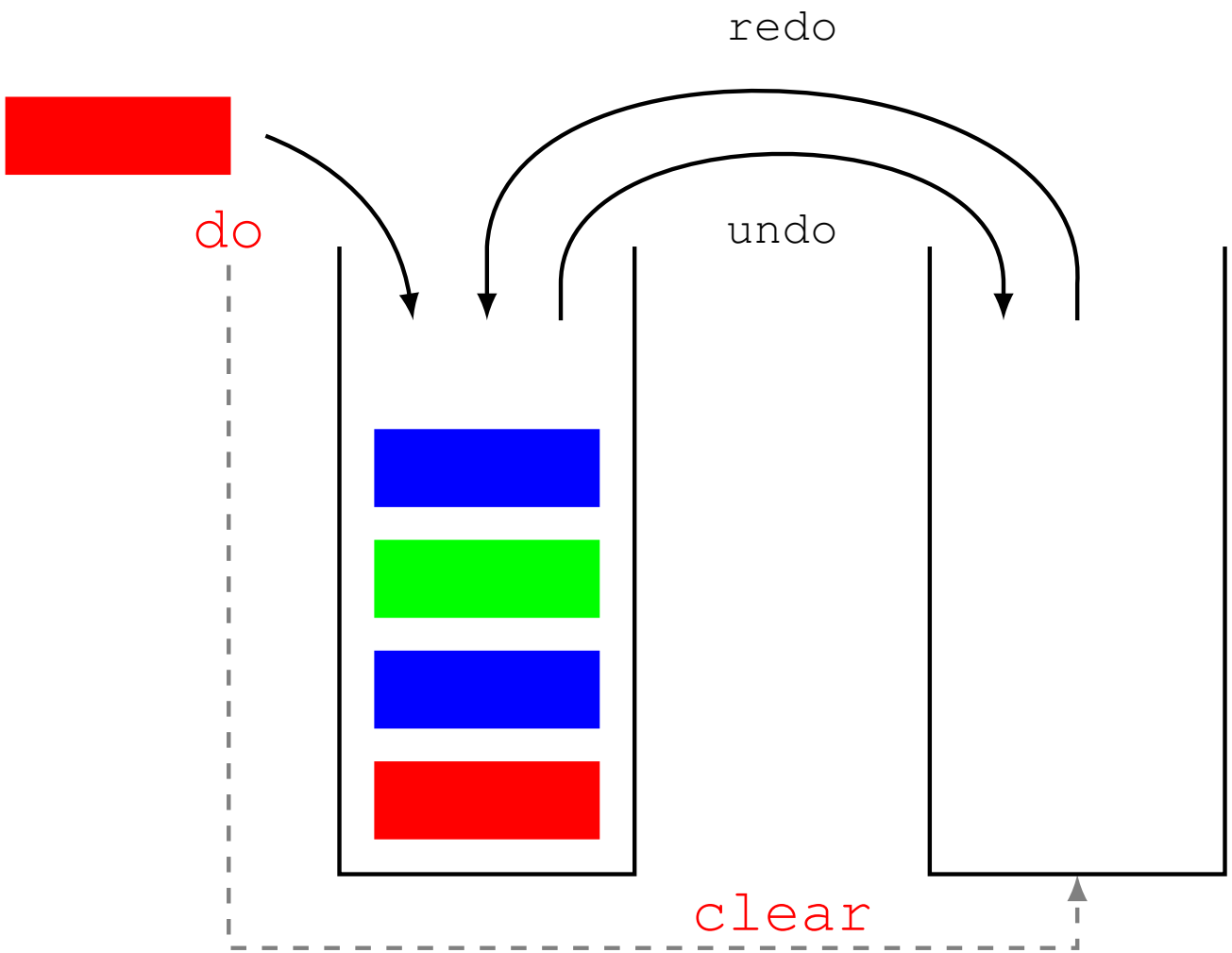

redo
do
undo
clear

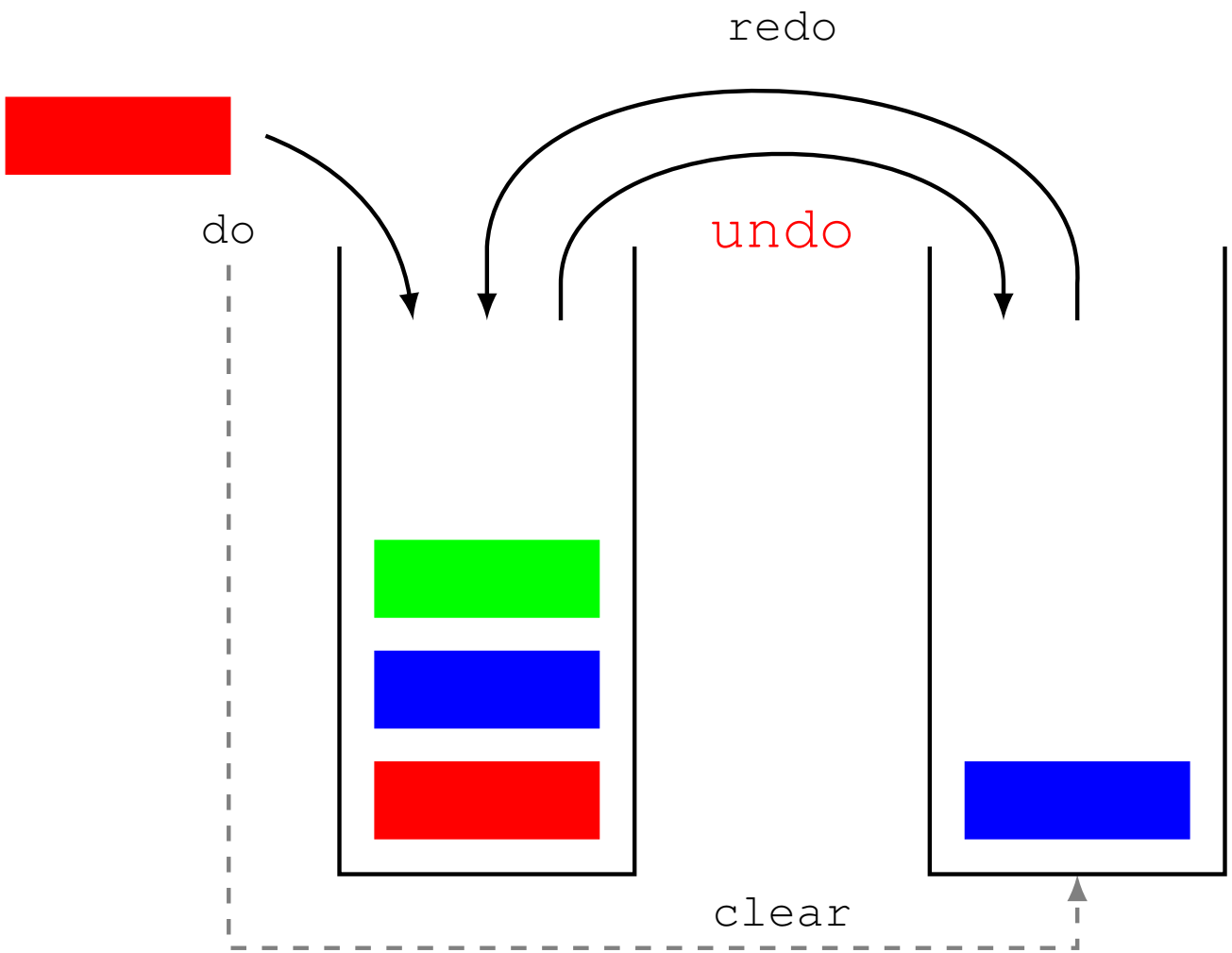

redo
do
undo
clear

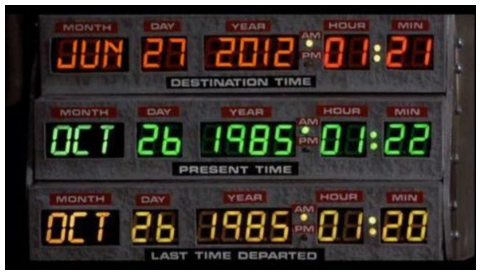

redo
do
undo
clear

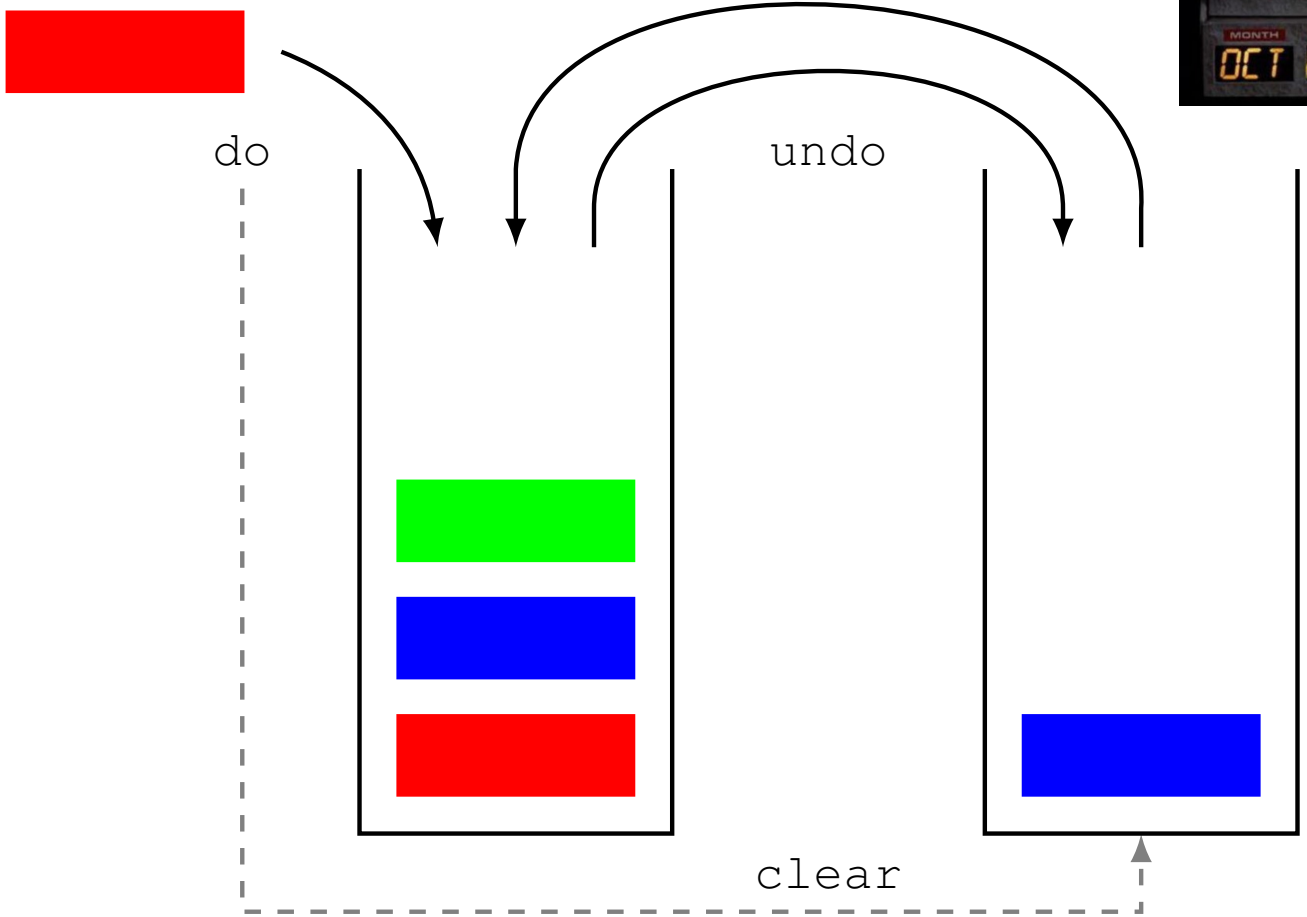

redo
do
undo
clear


redo
do
undo
clear


redo
do
undo
clear


redo
do
undo
clear


MONTH DAY YEAR HOUR MIN
JUN 27 2012 01:21
DESTINATION TIME
MONTH DAY YEAR HOUR MIN
OCT 26 1985 01:22
PRESENT TIME
MONTH DAY YEAR HOUR MIN
OCT 26 1985 01:20
LAST TIME DEPARTED


redo
do
undo
clear

Définition récursive de factorielle (en Caml)

```
let rec fact n =
  if n < 2 then 1 else fact(n - 1) * n;;
```

Compilation plus délicate :
utilise une pile en interne
(structure de données
gérée par le compilateur)

Compilation facile :
utilise des instructions
de branchement conditionnels

Définition itérative de factorielle (en C)

```
int fact (int n){
    int r = 1, i;
    for (i = 2; i <= n; i++)
        r = r * i;
    return r;
}
```

Définition récursive de factorielle (en Caml)

```
let rec fact n =
 if n < 2 then 1 else fact(n - 1) * n;;
```

Compilation plus délicate :
utilise une pile d'appels
(structure de données
gérée par le compilateur)

Compilation facile :
utilise des instructions
de branchement conditionnels

Définition itérative de factorielle (en C)

```
int fact (int n){
    int r = 1, i;
    for (i = 2; i <= n; i++)
        r = r * i;
    return r;
}
```

Définition récursive de factorielle (en Caml)

```
let rec fact n =
 if n < 2 then 1 else fact(n - 1) * n;;
```

Compilation plus délicate :
utilise une pile d'appels
(structure de données
gérée par le compilateur)

Compilation facile :
utilise des instructions
de branchement conditionnels

Définition itérative de factorielle (en C)

```
int fact (int n){
    int r = 1, i;
    for (i = 2; i <= n; i++)
        r = r * i;
    return r;
}
```

Définition récursive de factorielle (en Caml)

```
let rec fact n =
 if n < 2 then 1 else fact(n - 1) * n;;
```

Compilation plus délicate :
utilise une pile d'appels
(structure de données
gérée par le compilateur)

Compilation facile :
utilise des instructions
de branchement conditionnels

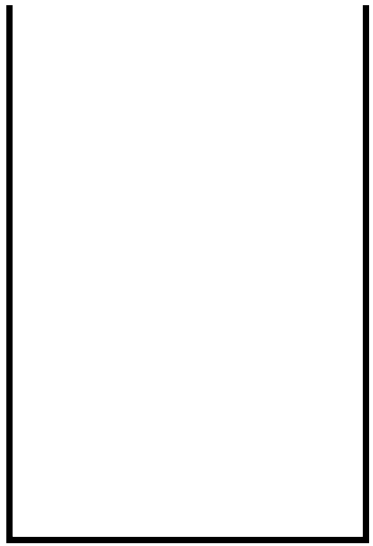
Définition itérative de factorielle (en C)

```
int fact (int n){
    int r = 1, i;
    for (i = 2; i <= n; i++)
        r = r * i;
    return r;
}
```

Définition récursive de factorielle (en Caml)

```
let rec fact n =
 if n < 2 then 1 else fact(n - 1) * n;;
```

Compilation plus délicate :
utilise une pile d'appels
(structure de données
gérée par le compilateur)

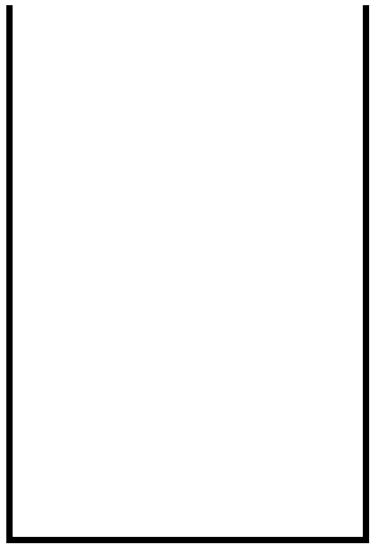


Définition récursive de factorielle (en Caml)

```
let rec fact n =
 if n < 2 then 1 else fact(n - 1) * n;;
```

Compilation plus délicate :
utilise une pile d'appels
(structure de données
gérée par le compilateur)

appel de `fact` avec `n=3`



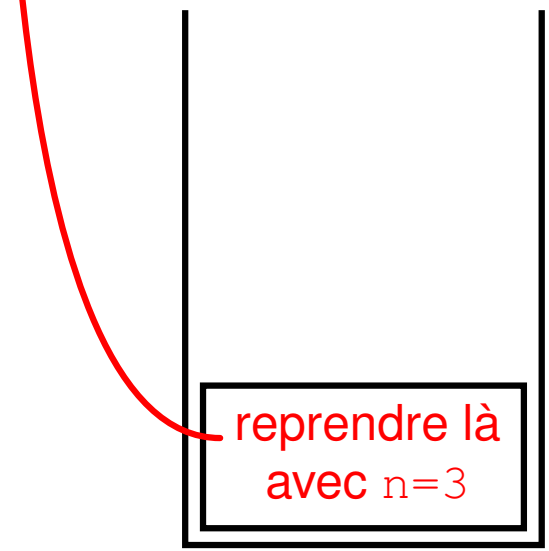
Définition récursive de factorielle (en Caml)

```
let rec fact n =
 if n < 2 then 1 else fact(n - 1) * n;;
```

Compilation plus délicate :
utilise une pile d'appels
(structure de données
gérée par le compilateur)

appel de `fact` avec `n=3`

appel de `fact` avec `n=2`



Définition récursive de factorielle (en Caml)

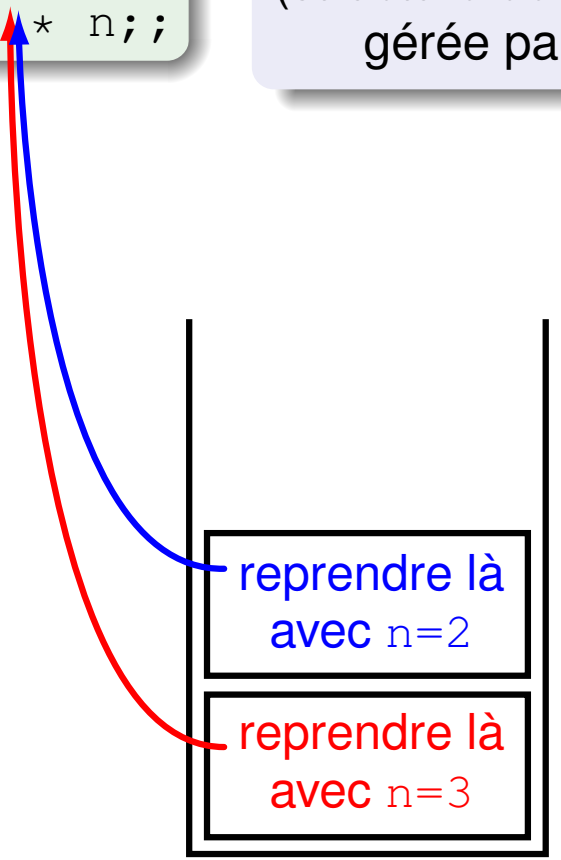
```
let rec fact n =
 if n < 2 then 1 else fact(n - 1) * n;;
```

Compilation plus délicate : utilise une pile d'appels (structure de données gérée par le compilateur)

appel de `fact` avec `n=3`

appel de `fact` avec `n=2`

appel de `fact` avec `n=1`

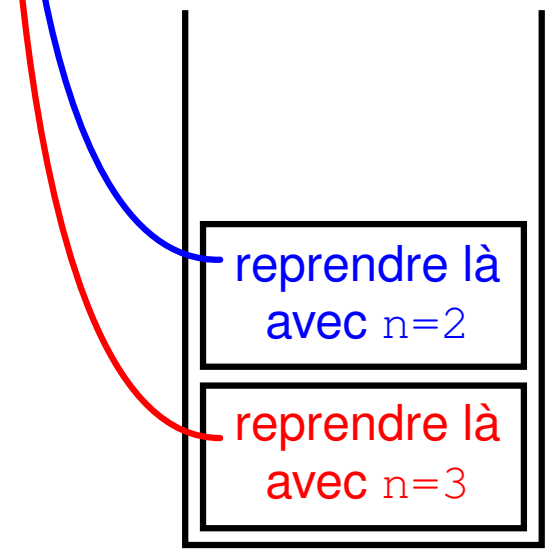


Définition récursive de factorielle (en Caml)

```
let rec fact n =
 if n < 2 then 1 else fact(n - 1) * n;;
```

Compilation plus délicate : utilise une pile d'appels (structure de données gérée par le compilateur)

appel de `fact` avec `n=3`
  appel de `fact` avec `n=2`
    appel de `fact` avec `n=1`
    retour avec `résultat=1`

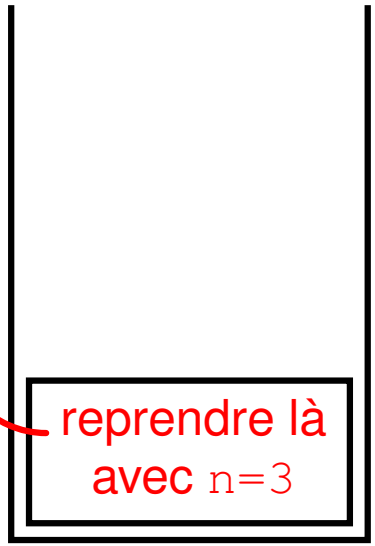


Définition récursive de factorielle (en Caml)

```
let rec fact n =
 if n < 2 then 1 else fact(n - 1) * n;;
```

Compilation plus délicate : utilise une pile d'appels (structure de données gérée par le compilateur)

appel de `fact` avec `n=3`
- appel de `fact` avec `n=2`
  - appel de `fact` avec `n=1`
  - retour avec `résultat=1`
- calcul de `1*n` avec `n=2`

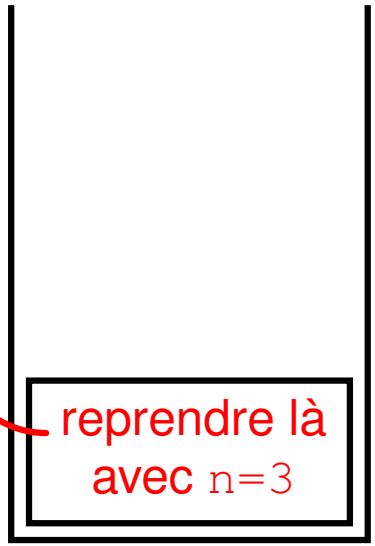


Définition récursive de factorielle (en Caml)

```
let rec fact n =
 if n < 2 then 1 else fact(n - 1) * n;;
```

Compilation plus délicate : utilise une pile d'appels (structure de données gérée par le compilateur)

appel de `fact` avec `n=3`
- appel de `fact` avec `n=2`
  - appel de `fact` avec `n=1`
  - retour avec `résultat=1`
- calcul de `1*n` avec `n=2`
- retour avec `résultat=2`



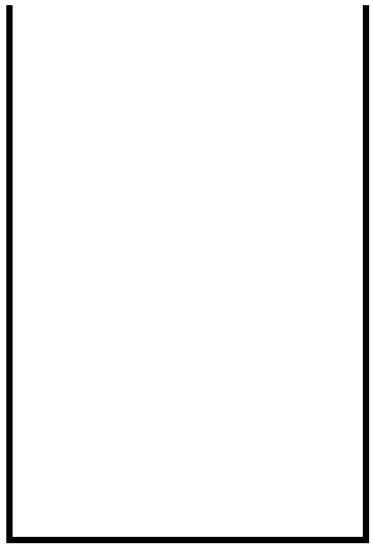
Définition récursive de factorielle (en Caml)

```
let rec fact n =
 if n < 2 then 1 else fact(n - 1) * n;;
```

Compilation plus délicate :
utilise une pile d'appels
(structure de données
gérée par le compilateur)

appel de `fact` avec `n=3`
- appel de `fact` avec `n=2`
  - appel de `fact` avec `n=1`
  - retour avec `résultat=1`
- calcul de `1*n` avec `n=2`
- retour avec `résultat=2`

calcul de `2*n` avec `n=3`



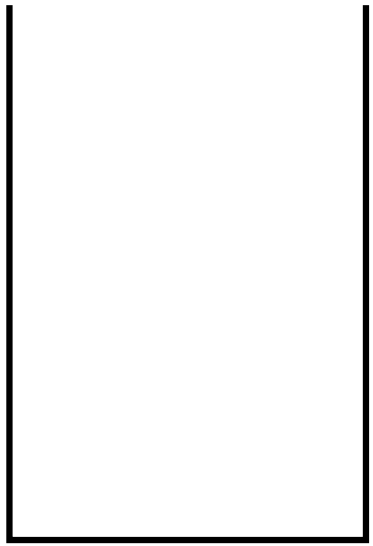
Définition récursive de factorielle (en Caml)

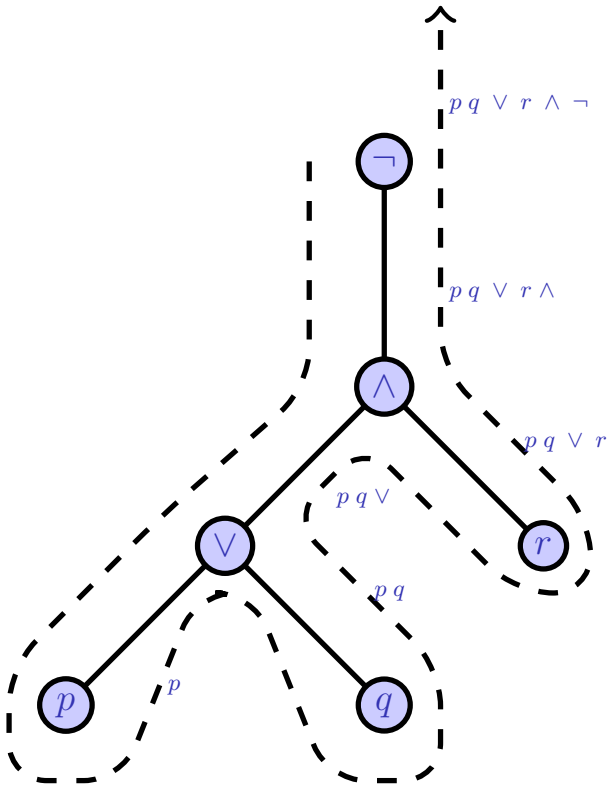
```
let rec fact n =
 if n < 2 then 1 else fact(n - 1) * n;;
```

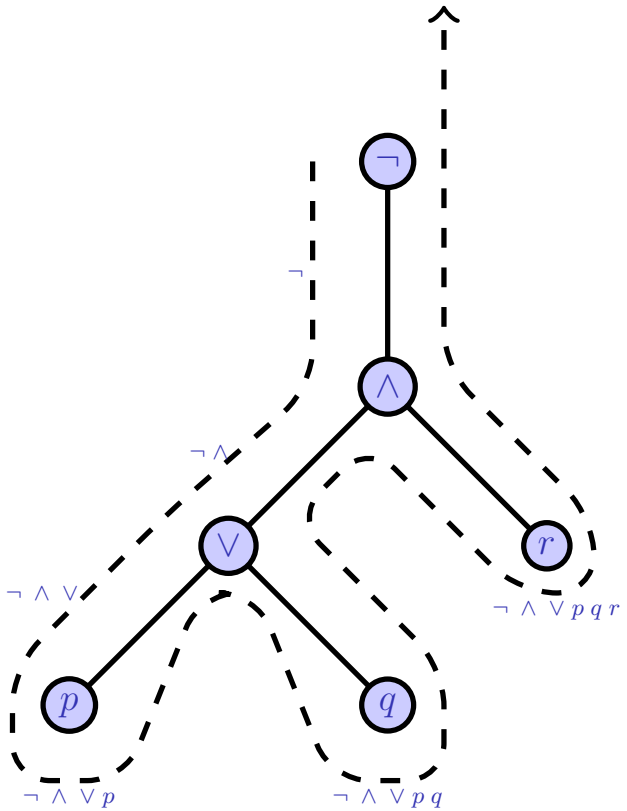
Compilation plus délicate :
utilise une pile d'appels
(structure de données
gérée par le compilateur)

appel de `fact` avec `n=3`
- appel de `fact` avec `n=2`
  - appel de `fact` avec `n=1`
  - retour avec `résultat=1`
- calcul de `1*n` avec `n=2`
- retour avec `résultat=2`

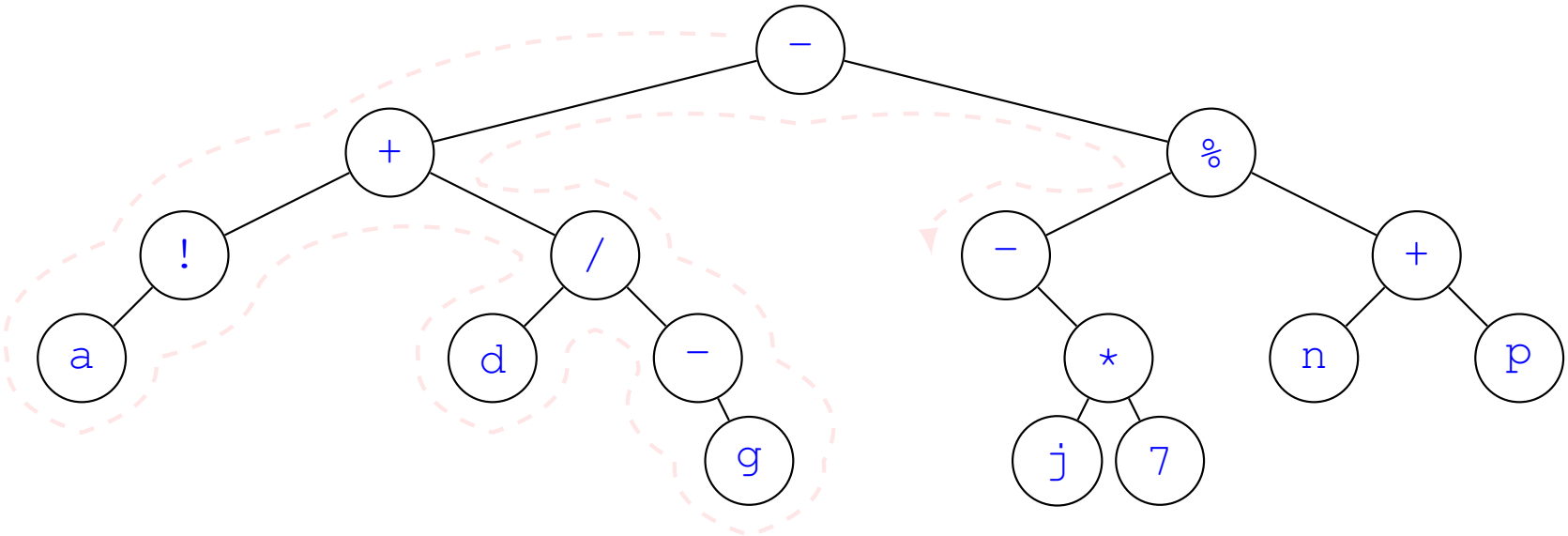
calcul de `2*n` avec `n=3`
arrêt avec `résultat=6`







(vu en PF1 — amphi 6)

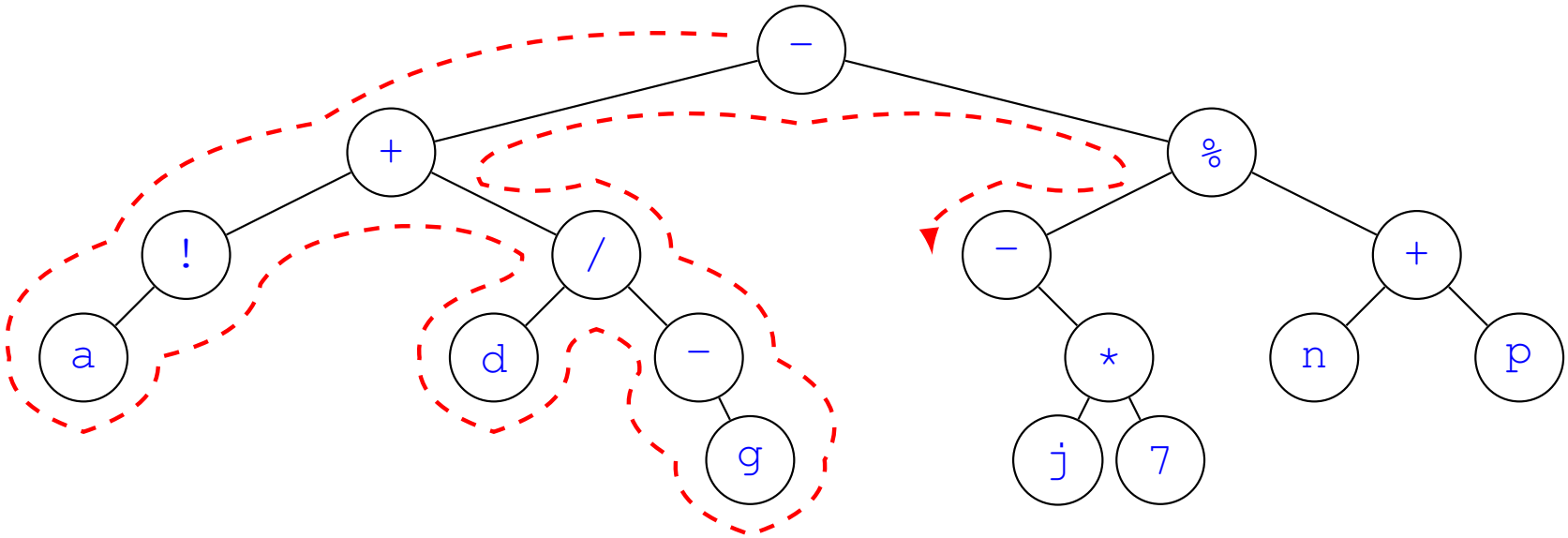


infixe

évaluation

préfixe

postfixe

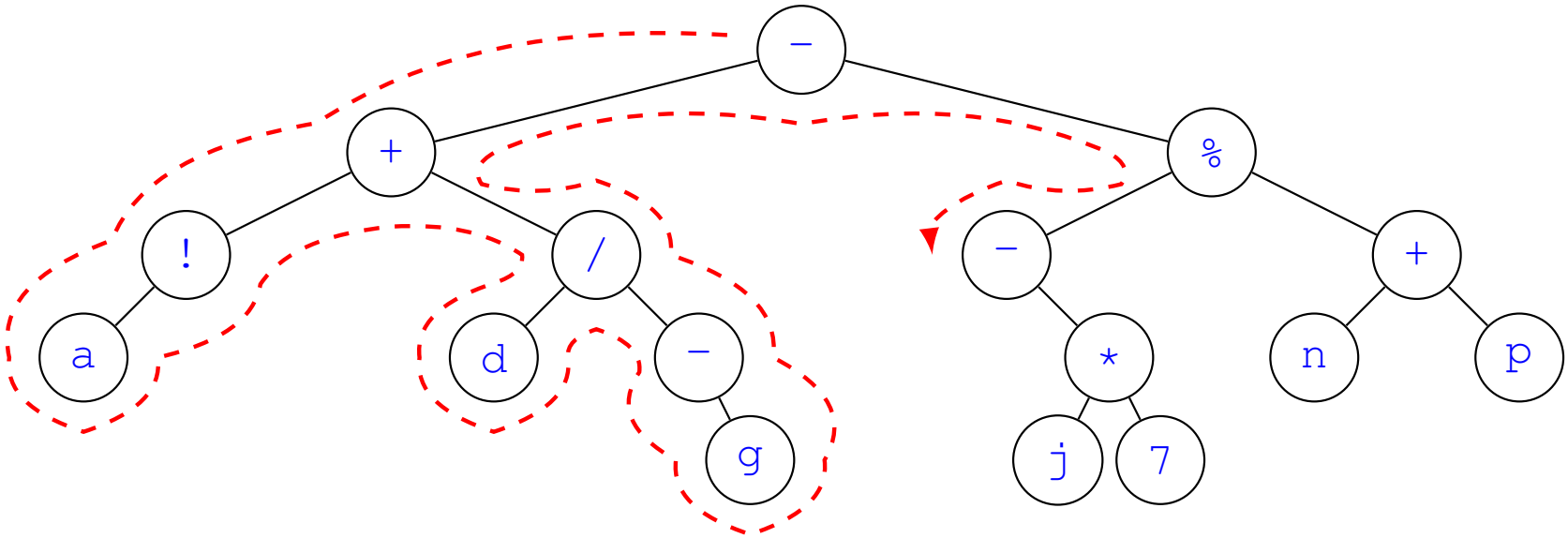


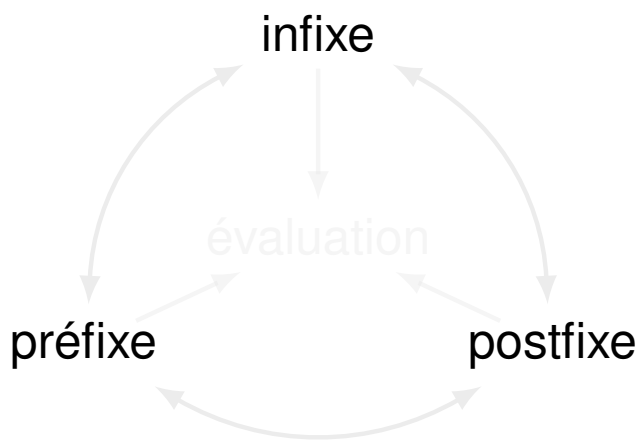
infixe

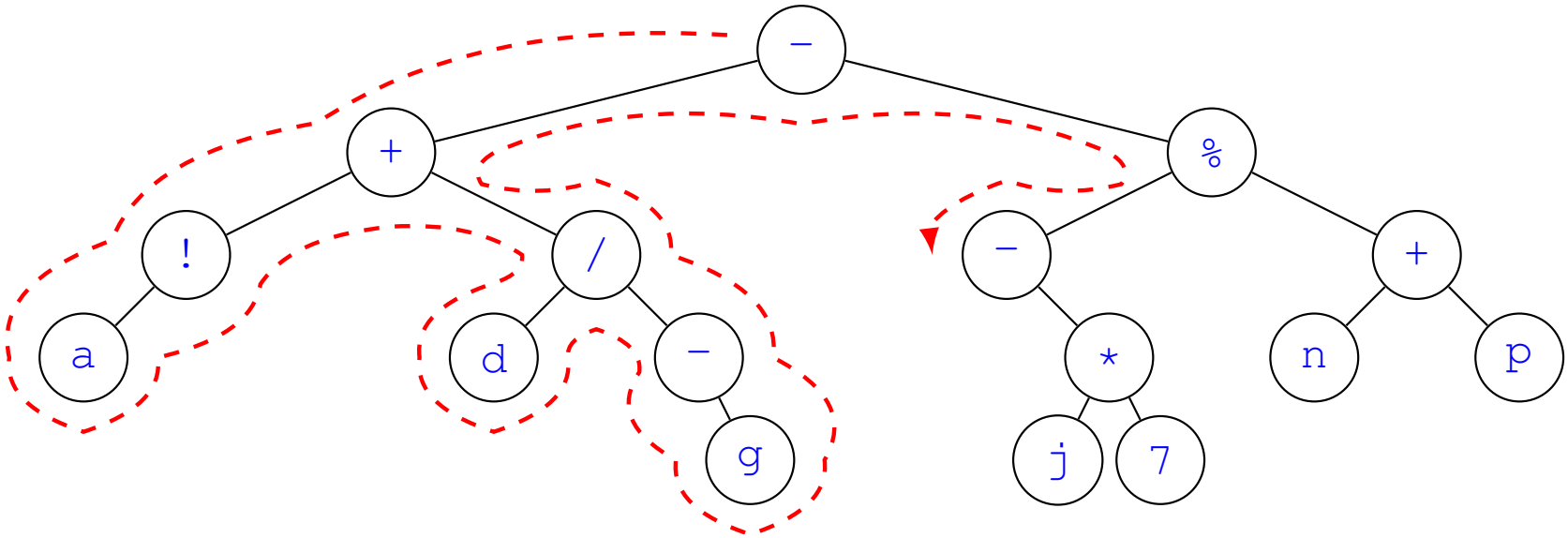
évaluation

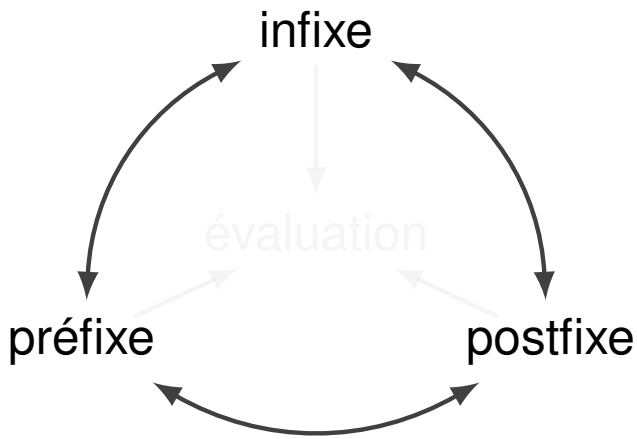
préfixe

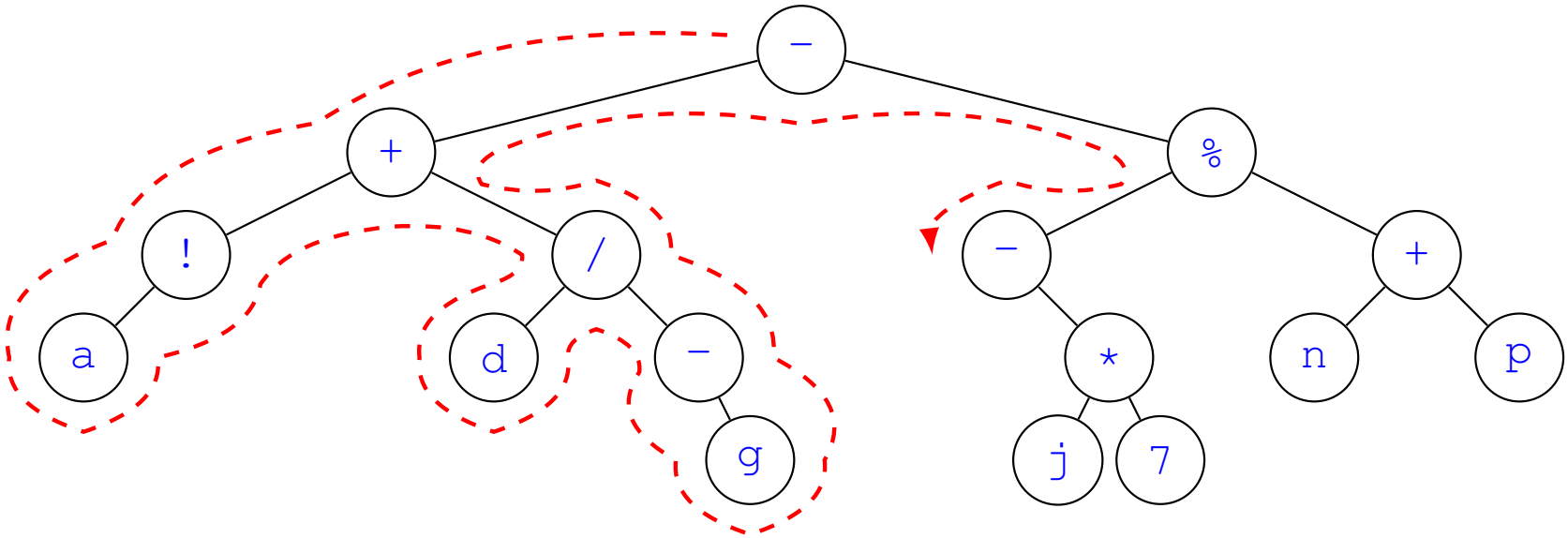
postfixe

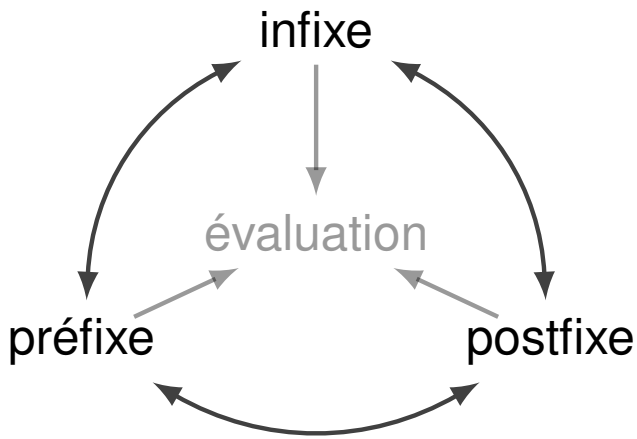

-
+
%
!
/
-
+
a
d
-
*
n
p
g
j
7


infixe
évaluation
préfixe
postfixe

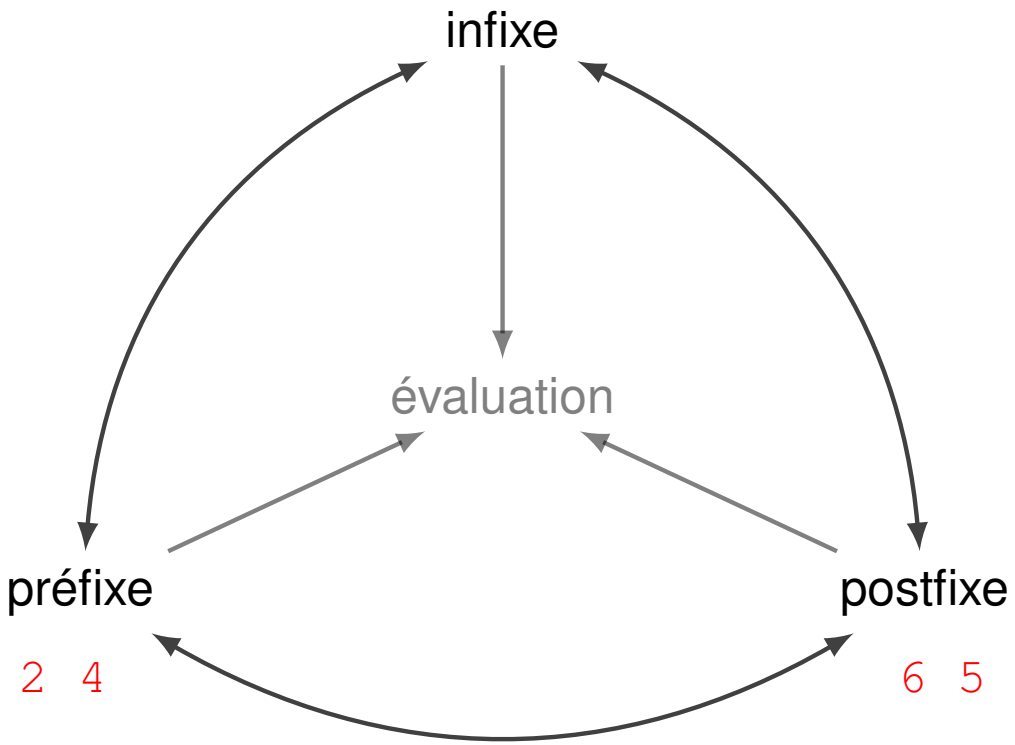

-
+
%
!
/
-
+
a
d
-
*
n
p
g
j
7


infixe
évaluation
préfixe
postfixe


-
+
%
!
/
-
+
a
d
-
*
n
p
g
j
7

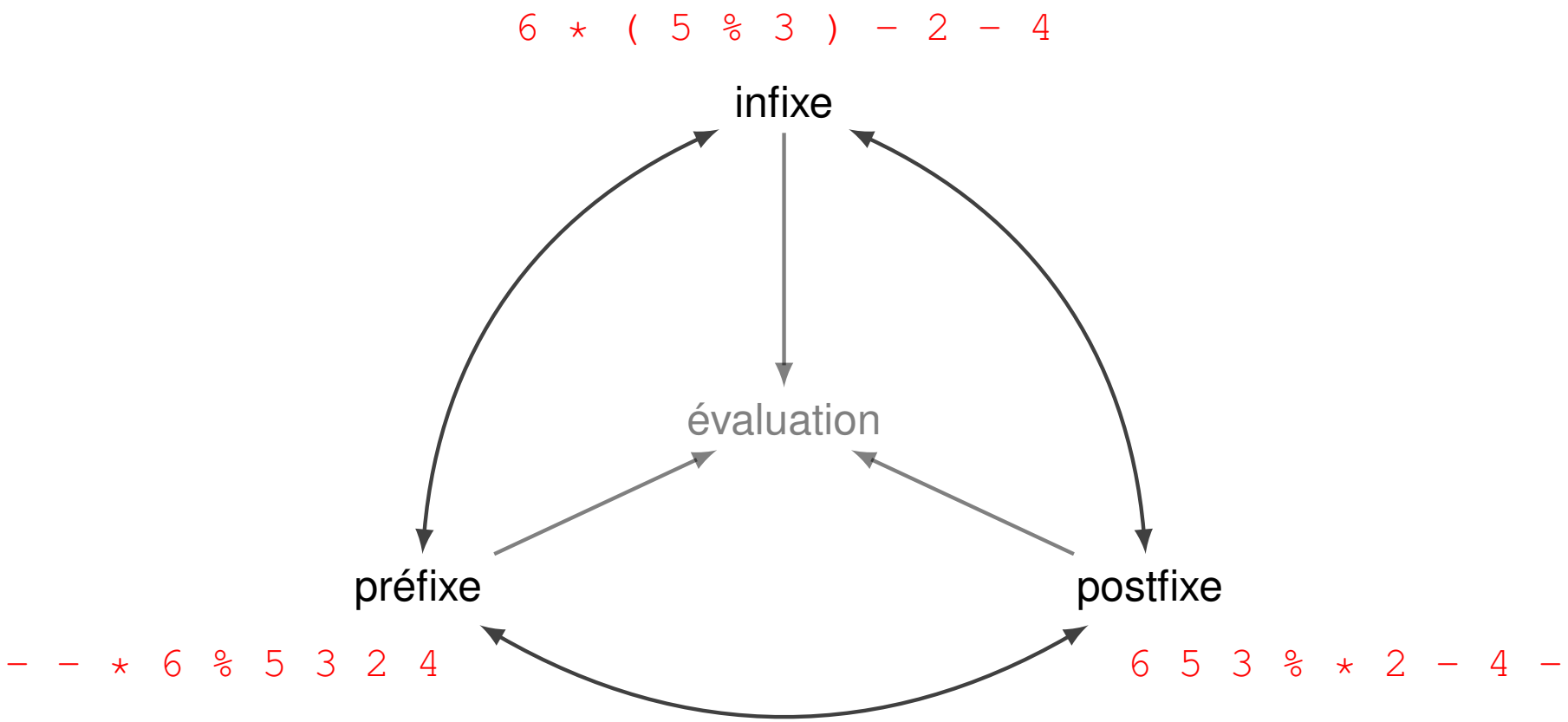

infixe
évaluation
préfixe
postfixe

6 * (5 % 3) - 2 - 4



- - * 6 % 5 3 2 4

6 5 3 % * 2 - 4 -

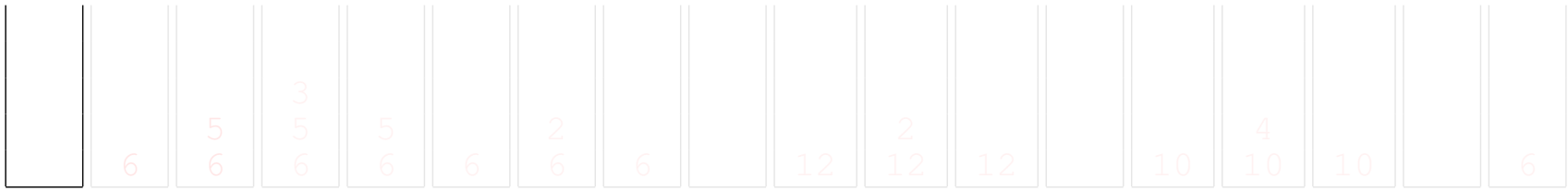

6 * (5 % 3) - 2 - 4
infixe
évaluation
préfixe
postfixe
- - * 6 % 5 3 2 4
6 5 3 % * 2 - 4 -

évaluation

postfixe

```
6 5 3 % * 2 - 4 -
```

```
/* on lit l'expression de gauche à droite
 * si le symbole est un opérande, on l'empile
 * sinon (c'est un opérateur (binaire))
 *   * on dépile (deux) symboles
 *   * on exécute l'opération
 *   * on empile son résultat
 */
```

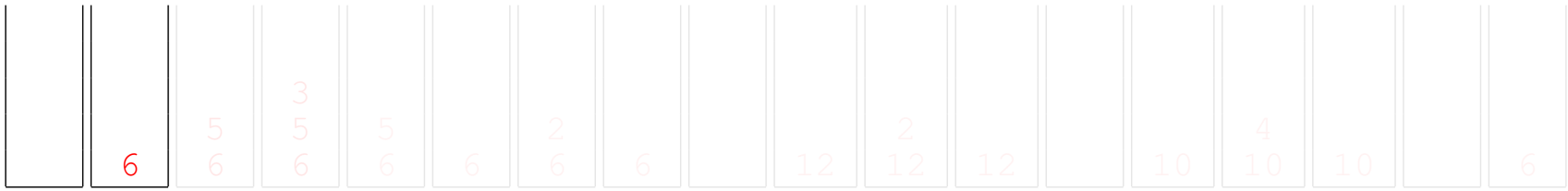


évaluation

postfixe

```
6 5 3 % * 2 - 4 -
▲
```

```
/* on lit l'expression de gauche à droite
 * si le symbole est un opérande, on l'empile
 * sinon (c'est un opérateur (binaire))
 *   * on dépile (deux) symboles
 *   * on exécute l'opération
 *   * on empile son résultat
 */
```

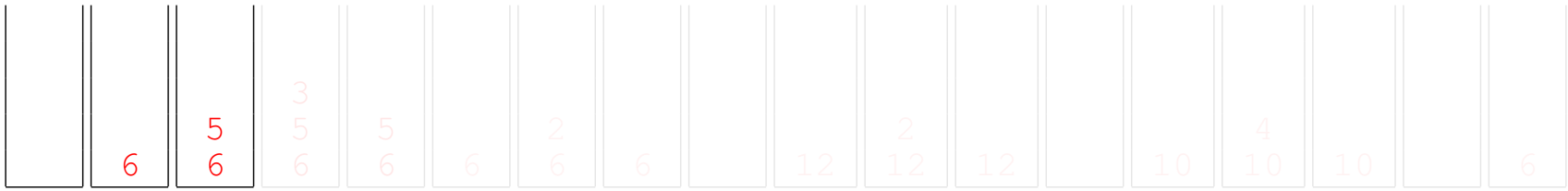


évaluation

postfixe

```
/* on lit l'expression de gauche à droite
 * si le symbole est un opérande, on l'empile
 * sinon (c'est un opérateur (binaire))
 *   * on dépile (deux) symboles
 *   * on exécute l'opération
 *   * on empile son résultat
 */
```

6 5 3 % * 2 - 4 -

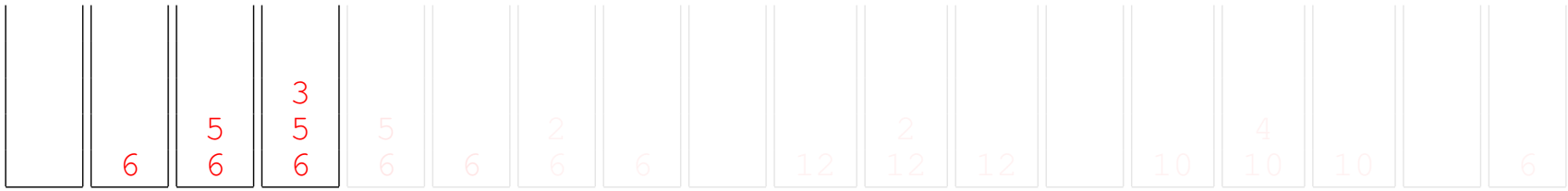


évaluation

postfixe

```
/* on lit l'expression de gauche à droite
 * si le symbole est un opérande, on l'empile
 * sinon (c'est un opérateur (binaire))
 *   * on dépile (deux) symboles
 *   * on exécute l'opération
 *   * on empile son résultat
 */
```

6 5 3 % * 2 - 4 -

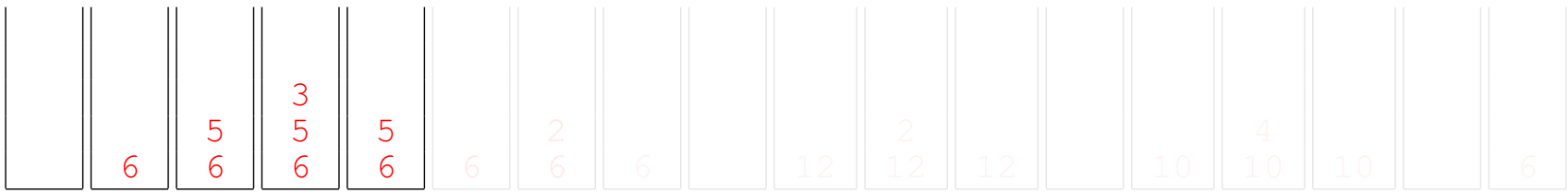


évaluation

postfixe

```
6 5 3 % * 2 - 4 -
      ▲
```

```
/* on lit l'expression de gauche à droite
 * si le symbole est un opérande, on l'empile
 * sinon (c'est un opérateur (binaire))
 *   * on dépile (deux) symboles
 *   * on exécute l'opération
 *   * on empile son résultat
 */
```

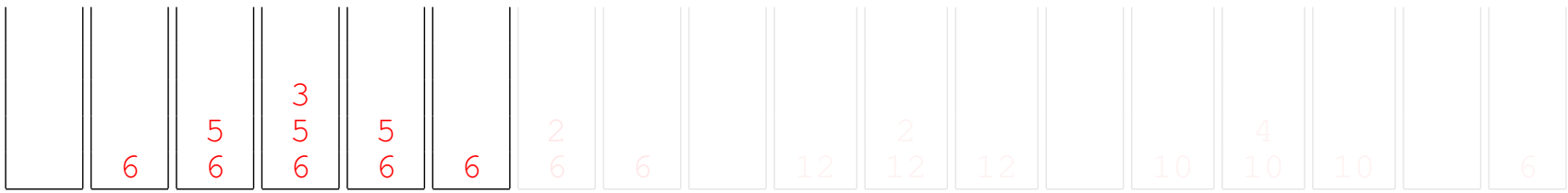


évaluation

postfixe

```
6 5 3 % * 2 - 4 -
```

```
/* on lit l'expression de gauche à droite
 * si le symbole est un opérande, on l'empile
 * sinon (c'est un opérateur (binaire))
 *   * on dépile (deux) symboles
 *   * on exécute l'opération
 *   * on empile son résultat
 */
```

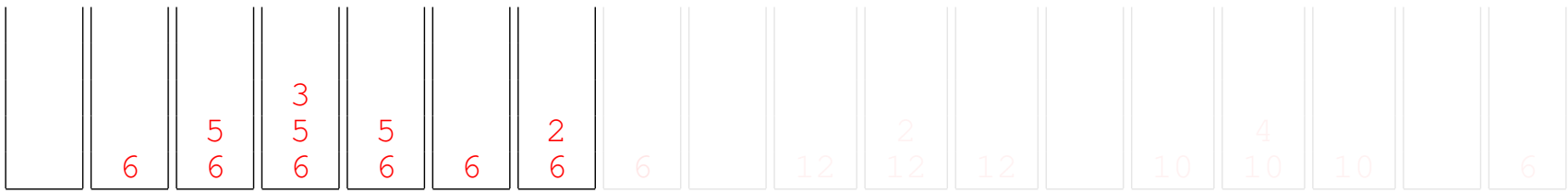


évaluation

postfixe

```
6 5 3 % * 2 - 4 -
      ▲
```

```
/* on lit l'expression de gauche à droite
 * si le symbole est un opérande, on l'empile
 * sinon (c'est un opérateur (binaire))
 *   * on dépile (deux) symboles
 *   * on exécute l'opération
 *   * on empile son résultat
 */
```

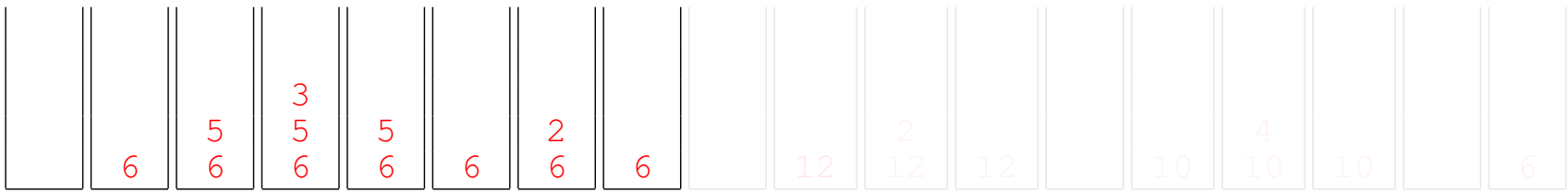


évaluation

postfixe

6 5 3 % * 2 - 4 -

```
/* on lit l'expression de gauche à droite
 * si le symbole est un opérande, on l'empile
 * sinon (c'est un opérateur (binaire))
 *   * on dépile (deux) symboles
 *   * on exécute l'opération
 *   * on empile son résultat
 */
```

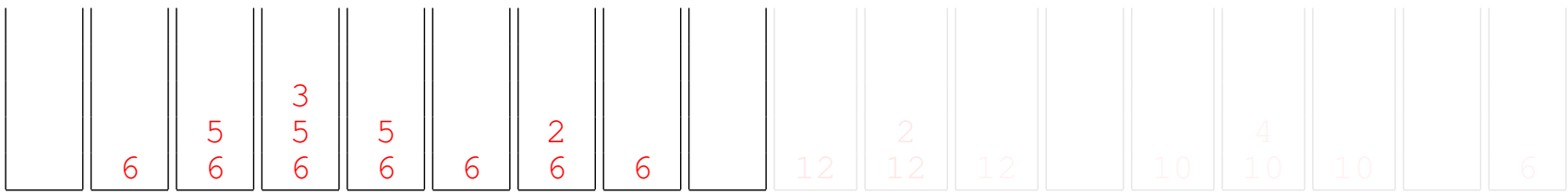


évaluation

postfixe

6 5 3 % * 2 - 4 -

```
/* on lit l'expression de gauche à droite
 * si le symbole est un opérande, on l'empile
 * sinon (c'est un opérateur (binaire))
 *   * on dépile (deux) symboles
 *   * on exécute l'opération
 *   * on empile son résultat
 */
```

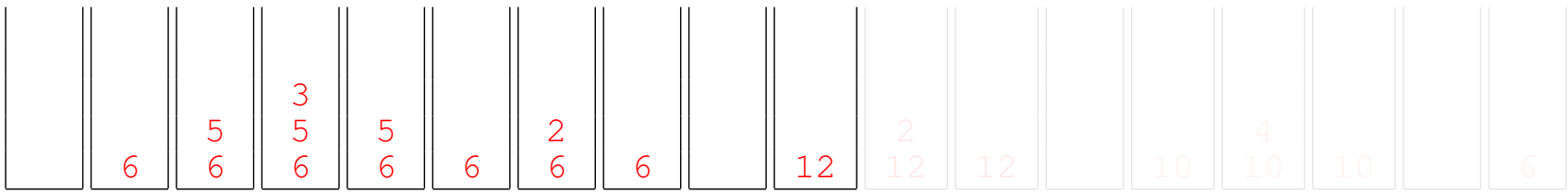


évaluation

postfixe

6 5 3 % * 2 - 4 -

```
/* on lit l'expression de gauche à droite
 * si le symbole est un opérande, on l'empile
 * sinon (c'est un opérateur (binaire))
 *   * on dépile (deux) symboles
 *   * on exécute l'opération
 *   * on empile son résultat
 */
```

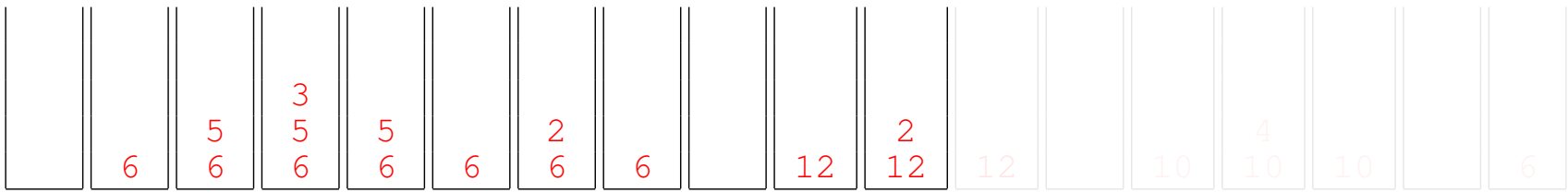


évaluation

postfixe

```
6 5 3 % * 2 - 4 -
```

```
/* on lit l'expression de gauche à droite
 * si le symbole est un opérande, on l'empile
 * sinon (c'est un opérateur (binaire))
 *   * on dépile (deux) symboles
 *   * on exécute l'opération
 *   * on empile son résultat
 */
```

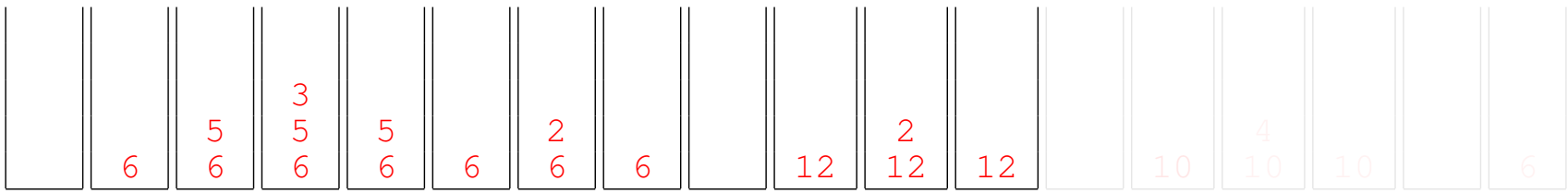


évaluation

postfixe

```
6 5 3 % * 2 - 4 -
        ▲
```

```
/* on lit l'expression de gauche à droite
 * si le symbole est un opérande, on l'empile
 * sinon (c'est un opérateur (binaire))
 *   * on dépile (deux) symboles
 *   * on exécute l'opération
 *   * on empile son résultat
 */
```

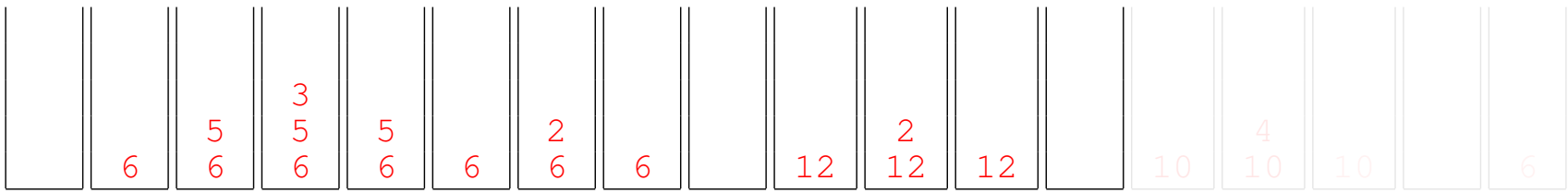


évaluation

postfixe

```
6 5 3 % * 2 - 4 -
          ▲
```

```
/* on lit l'expression de gauche à droite
 * si le symbole est un opérande, on l'empile
 * sinon (c'est un opérateur (binaire))
 *   * on dépile (deux) symboles
 *   * on exécute l'opération
 *   * on empile son résultat
 */
```

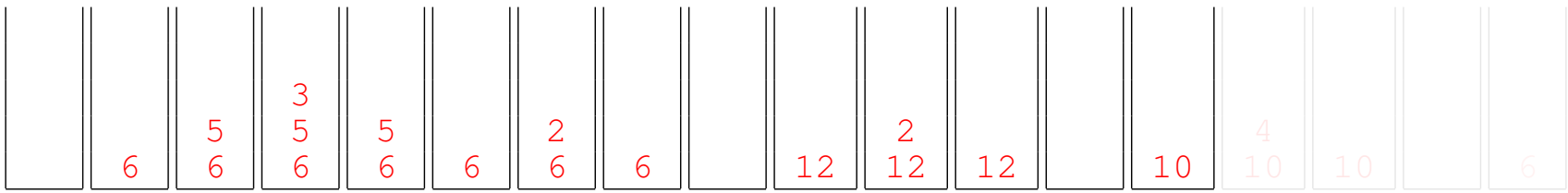


évaluation

postfixe

```
6 5 3 % * 2 - 4 -
          ▲
```

```
/* on lit l'expression de gauche à droite
 * si le symbole est un opérande, on l'empile
 * sinon (c'est un opérateur (binaire))
 *   * on dépile (deux) symboles
 *   * on exécute l'opération
 *   * on empile son résultat
 */
```

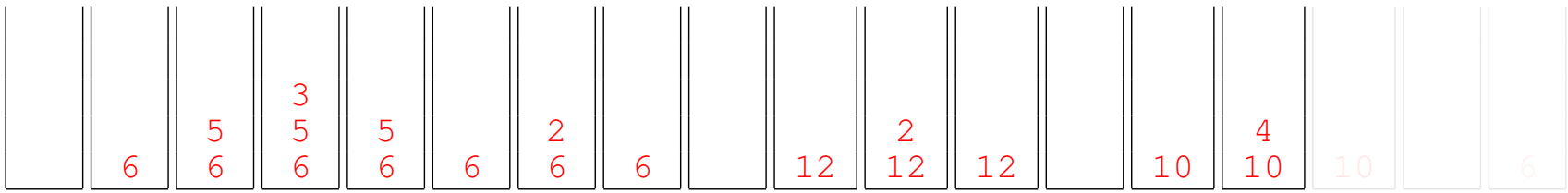


évaluation

postfixe

```
/* on lit l'expression de gauche à droite
 * si le symbole est un opérande, on l'empile
 * sinon (c'est un opérateur (binaire))
 *   * on dépile (deux) symboles
 *   * on exécute l'opération
 *   * on empile son résultat
 */
```

```
6 5 3 % * 2 - 4 -
  ▲
```

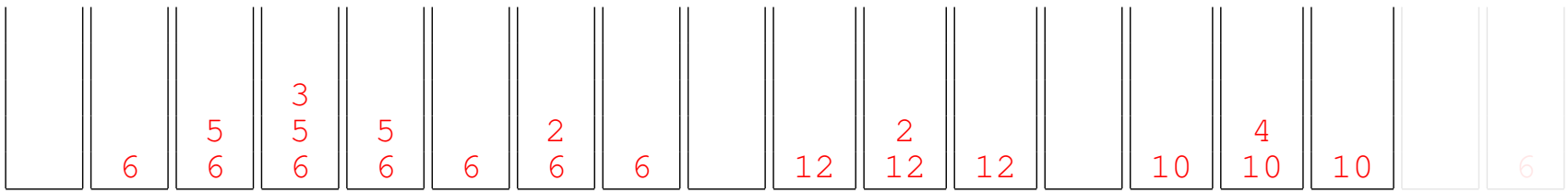


évaluation

postfixe

```
/* on lit l'expression de gauche à droite
 * si le symbole est un opérande, on l'empile
 * sinon (c'est un opérateur (binaire))
 *   * on dépile (deux) symboles
 *   * on exécute l'opération
 *   * on empile son résultat
 */
```

```
6 5 3 % * 2 - 4 -
  ▲
```

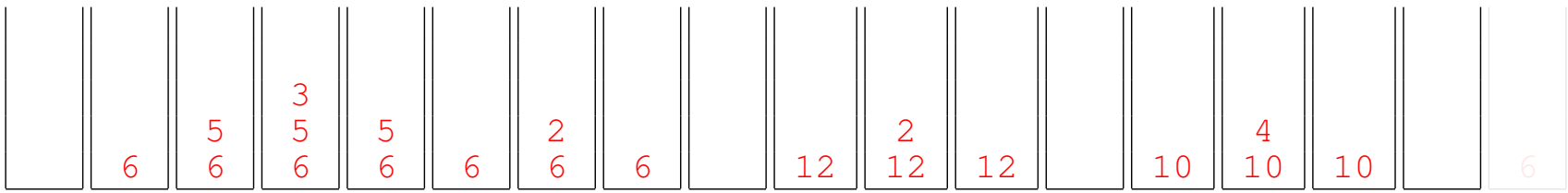


évaluation

postfixe

```
6 5 3 % * 2 - 4 -
▲
```

```
/* on lit l'expression de gauche à droite
 * si le symbole est un opérande, on l'empile
 * sinon (c'est un opérateur (binaire))
 *   * on dépile (deux) symboles
 *   * on exécute l'opération
 *   * on empile son résultat
 */
```

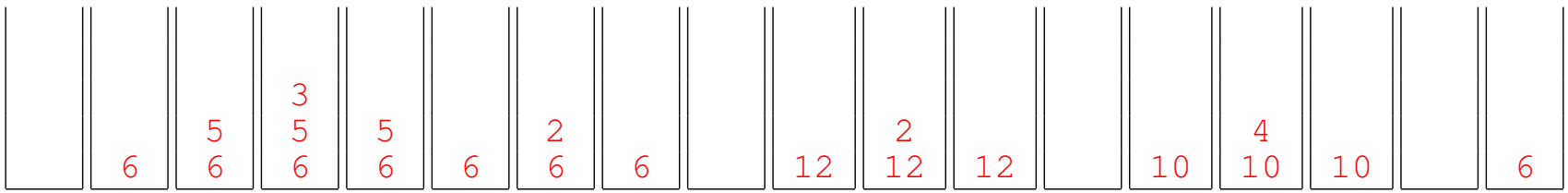


évaluation

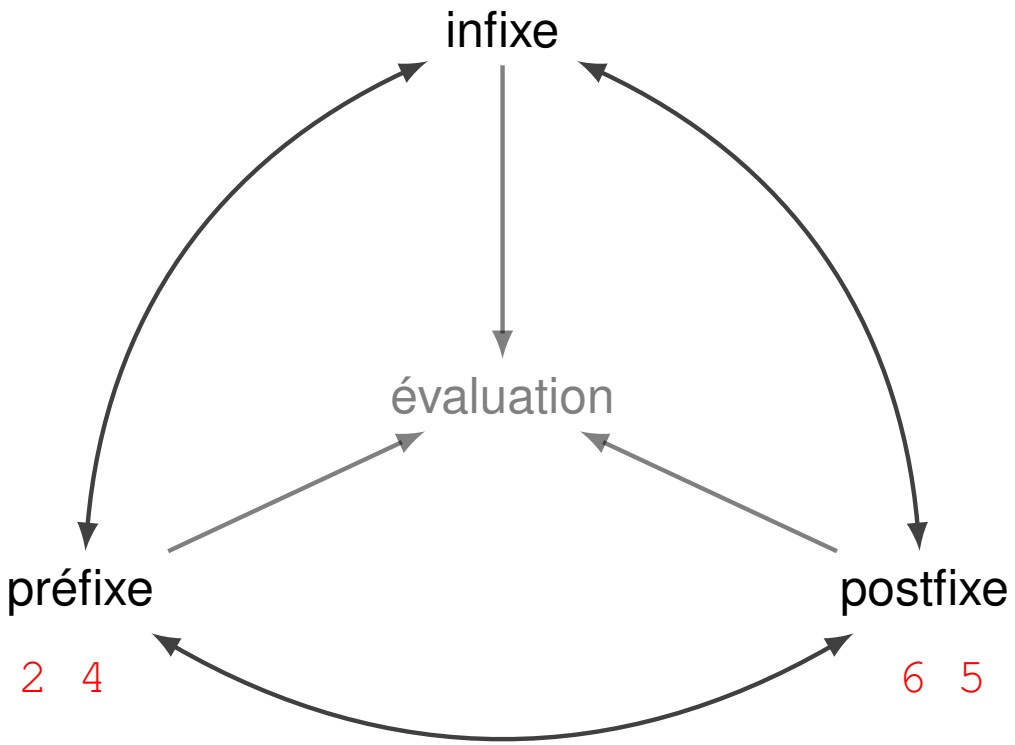
postfixe

```
6 5 3 % * 2 - 4 -
▲
```

```
/* on lit l'expression de gauche à droite
 * si le symbole est un opérande, on l'empile
 * sinon (c'est un opérateur (binaire))
 *   * on dépile (deux) symboles
 *   * on exécute l'opération
 *   * on empile son résultat
 */
```



6 * (5 % 3) - 2 - 4

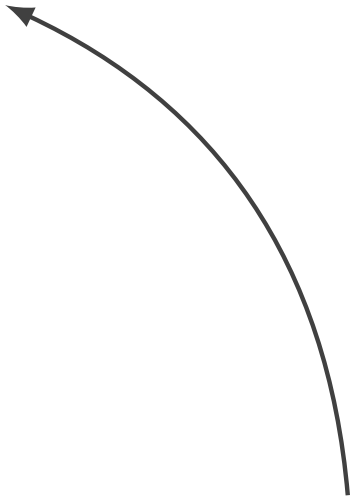


- - * 6 % 5 3 2 4

6 5 3 % * 2 - 4 -

```
6 * ( 5 % 3 ) - 2 - 4
```

infixe



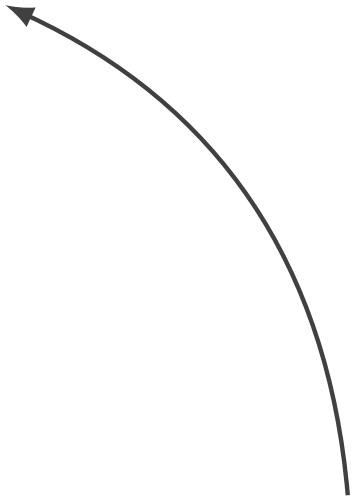
postfixe

```
6 5 3 % * 2 - 4 -
```

```
/* on lit l'expression de gauche à droite
 * si le symbole est un opérande, on l'empile
 * sinon (c'est un opérateur (binaire))
 *   * on dépile (deux) opérandes
 *   * on construit l'infixe du nouvel opérande
 *     entouré de parenthèses
 *   * on l'empile
 */
```

```
6 * ( 5 % 3 ) - 2 - 4
```

infixe



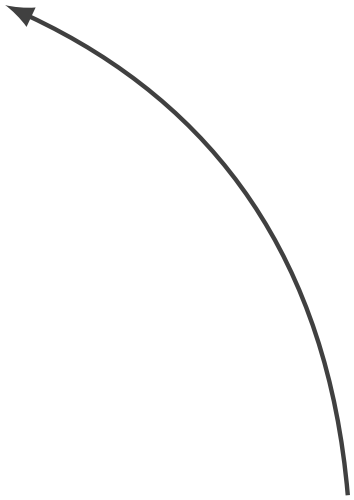
postfixe

```
6 5 3 % * 2 - 4 -
```

```
/* on lit l'expression de gauche à droite
 * si le symbole est un opérande, on l'empile
 * sinon (c'est un opérateur (binaire))
 *   * on dépile (deux) opérandes
 *   * on construit l'infixe du nouvel opérande
 *     entouré de parenthèses
 *   * on l'empile
 */
```

```
6 * ( 5 % 3 ) - 2 - 4
```

infixe



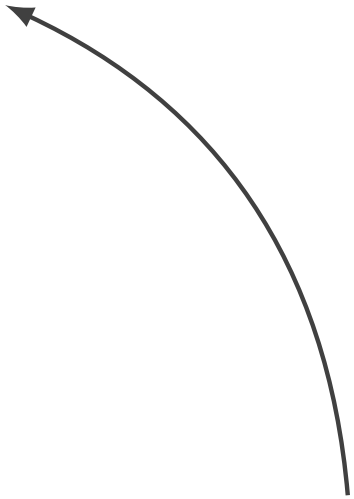
postfixe

```
6 5 3 % * 2 - 4 -
```

```
/* on lit l'expression de gauche à droite
 * si le symbole est un opérande, on l'empile
 * sinon (c'est un opérateur (binaire))
 *   * on dépile (deux) opérandes
 *   * on construit l'infixe du nouvel opérande
 *     entouré de parenthèses
 *   * on l'empile
 */
```

```
6 * ( 5 % 3 ) - 2 - 4
```

infixe



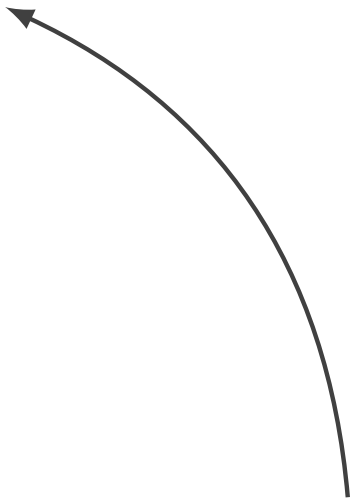
postfixe

```
6 5 3 % * 2 - 4 -
```

```
/* on lit l'expression de gauche à droite
 * si le symbole est un opérande, on l'empile
 * sinon (c'est un opérateur (binaire))
 *   * on dépile (deux) opérandes
 *   * on construit l'infixe du nouvel opérande
 *     entouré de parenthèses
 *   * on l'empile
 */
```

```
6 * ( 5 % 3 ) - 2 - 4
```

infixe



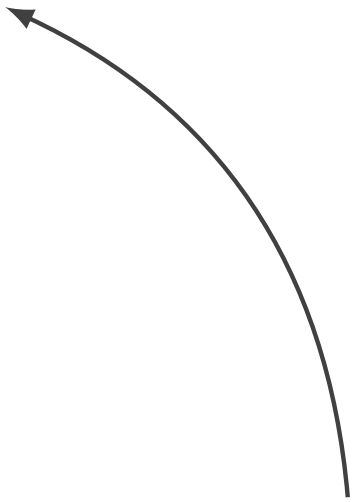
postfixe

```
6 5 3 % * 2 - 4 -
```

```
/* on lit l'expression de gauche à droite
 * si le symbole est un opérande, on l'empile
 * sinon (c'est un opérateur (binaire))
 *   * on dépile (deux) opérandes
 *   * on construit l'infixe du nouvel opérande
 *     entouré de parenthèses
 *   * on l'empile
 */
```

```
6 * ( 5 % 3 ) - 2 - 4
```

infixe



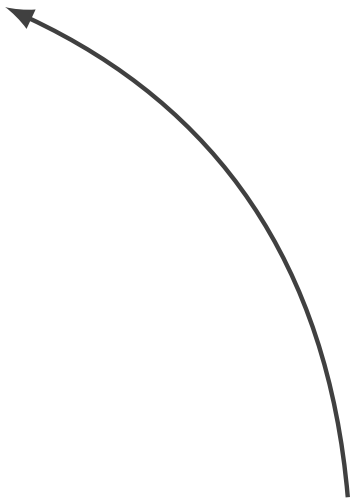
postfixe

```
6 5 3 % * 2 - 4 -
```

```
/* on lit l'expression de gauche à droite
 * si le symbole est un opérande, on l'empile
 * sinon (c'est un opérateur (binaire))
 *   * on dépile (deux) opérandes
 *   * on construit l'infixe du nouvel opérande
 *     entouré de parenthèses
 *   * on l'empile
 */
```

```
6 * ( 5 % 3 ) - 2 - 4
```

infixe



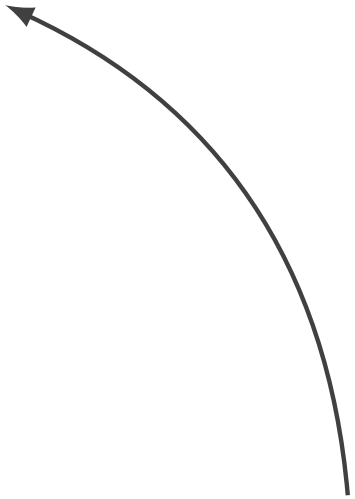
postfixe

```
6 5 3 % * 2 - 4 -
```

```
/* on lit l'expression de gauche à droite
 * si le symbole est un opérande, on l'empile
 * sinon (c'est un opérateur (binaire))
 *   * on dépile (deux) opérandes
 *   * on construit l'infixe du nouvel opérande
 *     entouré de parenthèses
 *   * on l'empile
 */
```

```
6 * ( 5 % 3 ) - 2 - 4
```

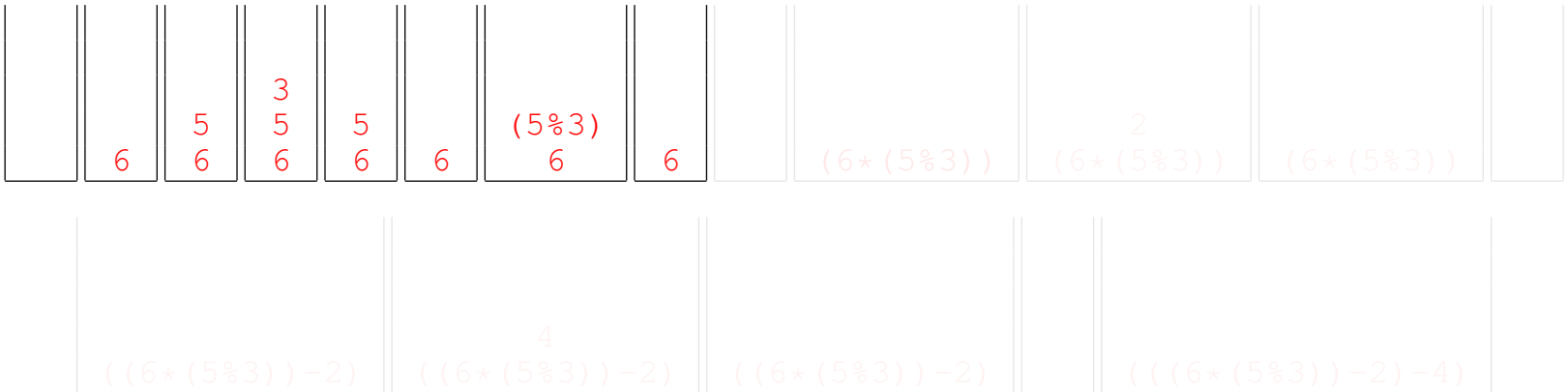
infixe



postfixe

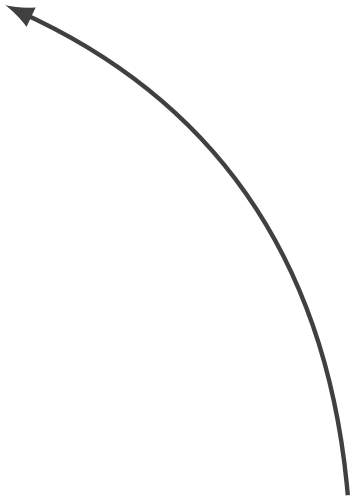
```
6 5 3 % * 2 - 4 -
```

```
/* on lit l'expression de gauche à droite
 * si le symbole est un opérande, on l'empile
 * sinon (c'est un opérateur (binaire))
 *   * on dépile (deux) opérandes
 *   * on construit l'infixe du nouvel opérande
 *     entouré de parenthèses
 *   * on l'empile
 */
```



```
6 * ( 5 % 3 ) - 2 - 4
```

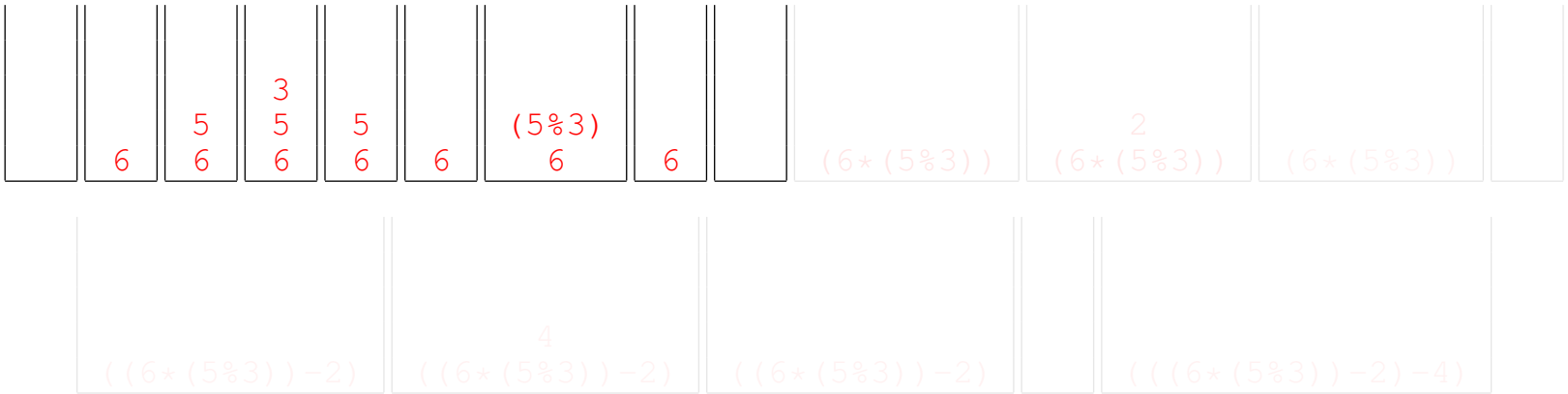
infixe



postfixe

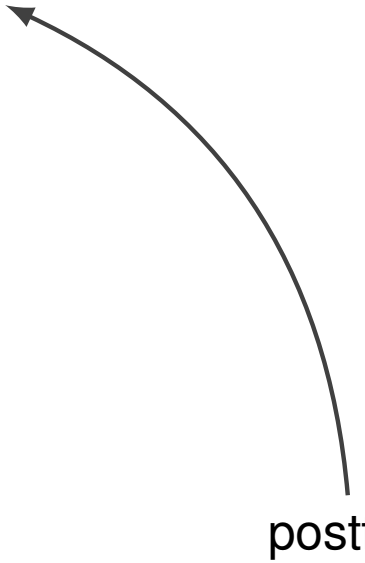
```
6 5 3 % * 2 - 4 -
```

```
/* on lit l'expression de gauche à droite
 * si le symbole est un opérande, on l'empile
 * sinon (c'est un opérateur (binaire))
 *   * on dépile (deux) opérandes
 *   * on construit l'infixe du nouvel opérande
 *     entouré de parenthèses
 *   * on l'empile
 */
```



```
6 * ( 5 % 3 ) - 2 - 4
```

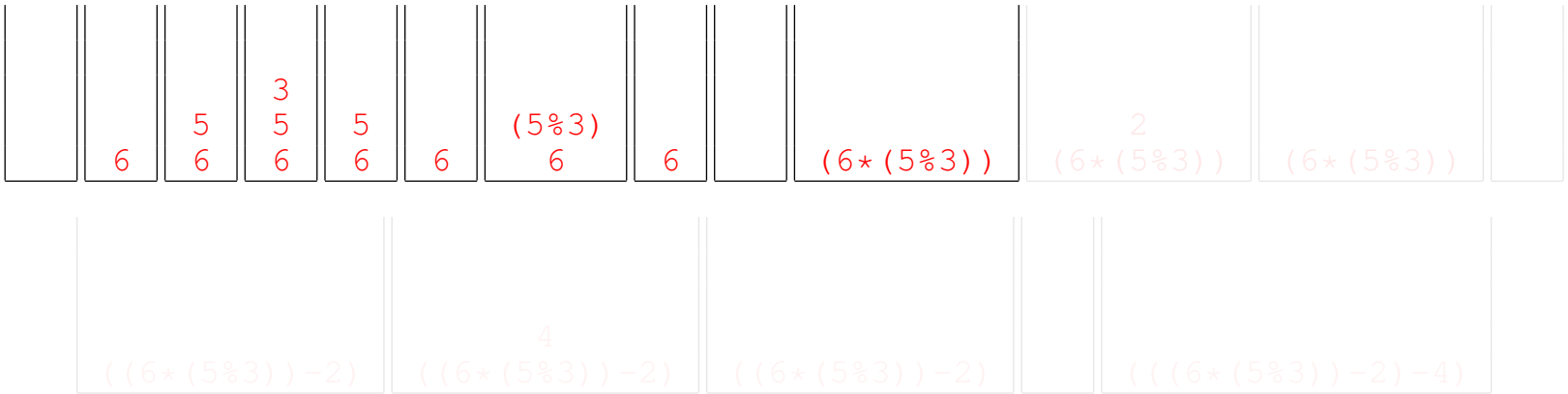
infixe



postfixe

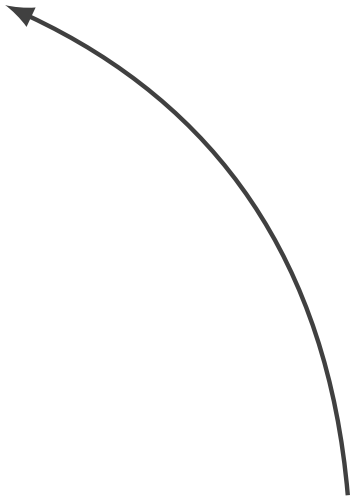
```
6 5 3 % * 2 - 4 -
```

```
/* on lit l'expression de gauche à droite
 * si le symbole est un opérande, on l'empile
 * sinon (c'est un opérateur (binaire))
 *   * on dépile (deux) opérandes
 *   * on construit l'infixe du nouvel opérande
 *     entouré de parenthèses
 *   * on l'empile
 */
```



```
6 * ( 5 % 3 ) - 2 - 4
```

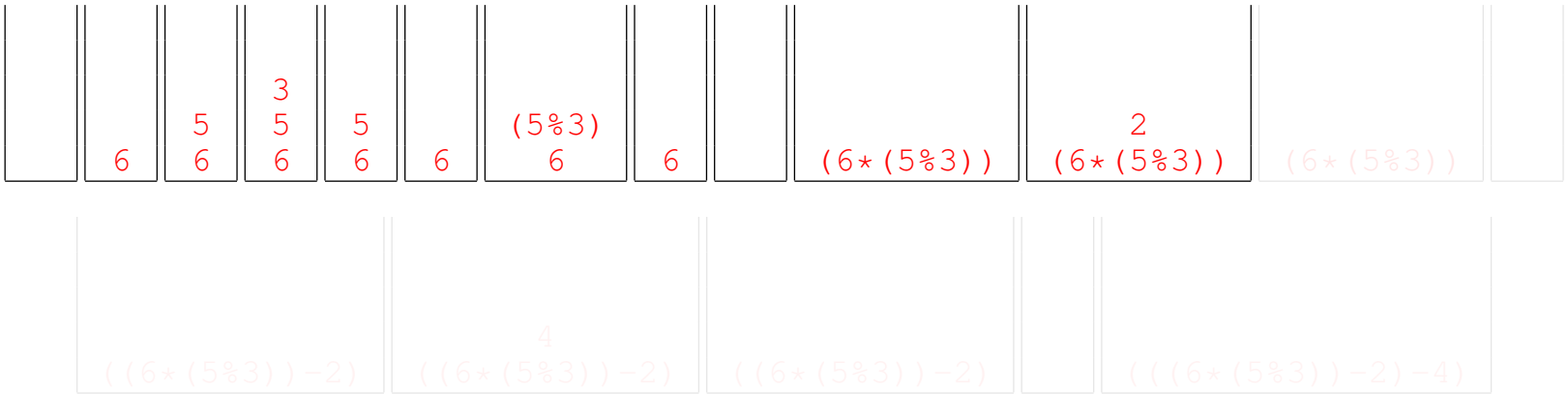
infixe



postfixe

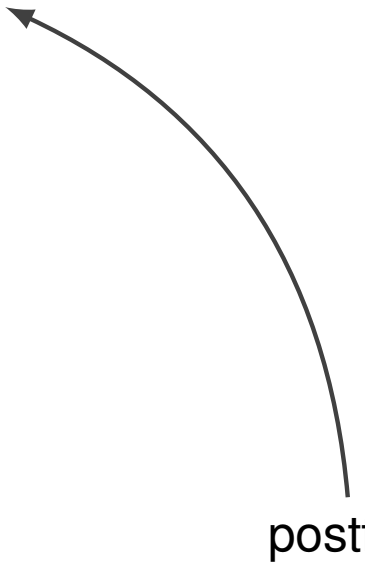
```
6 5 3 % * 2 - 4 -
```

```
/* on lit l'expression de gauche à droite
 * si le symbole est un opérande, on l'empile
 * sinon (c'est un opérateur (binaire))
 *   * on dépile (deux) opérandes
 *   * on construit l'infixe du nouvel opérande
 *     entouré de parenthèses
 *   * on l'empile
 */
```



```
6 * ( 5 % 3 ) - 2 - 4
```

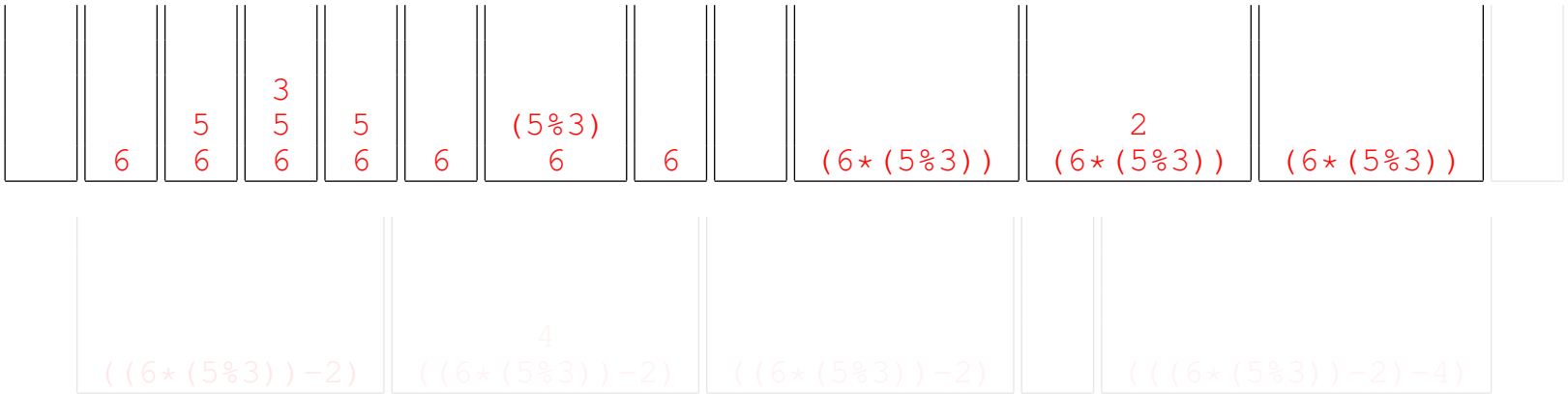
infixe



postfixe

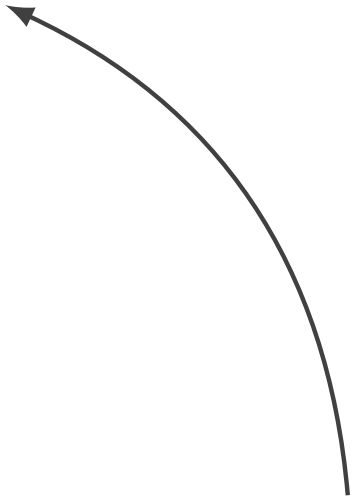
```
6 5 3 % * 2 - 4 -
```

```
/* on lit l'expression de gauche à droite
 * si le symbole est un opérande, on l'empile
 * sinon (c'est un opérateur (binaire))
 *   * on dépile (deux) opérandes
 *   * on construit l'infixe du nouvel opérande
 *     entouré de parenthèses
 *   * on l'empile
 */
```



```
6 * ( 5 % 3 ) - 2 - 4
```

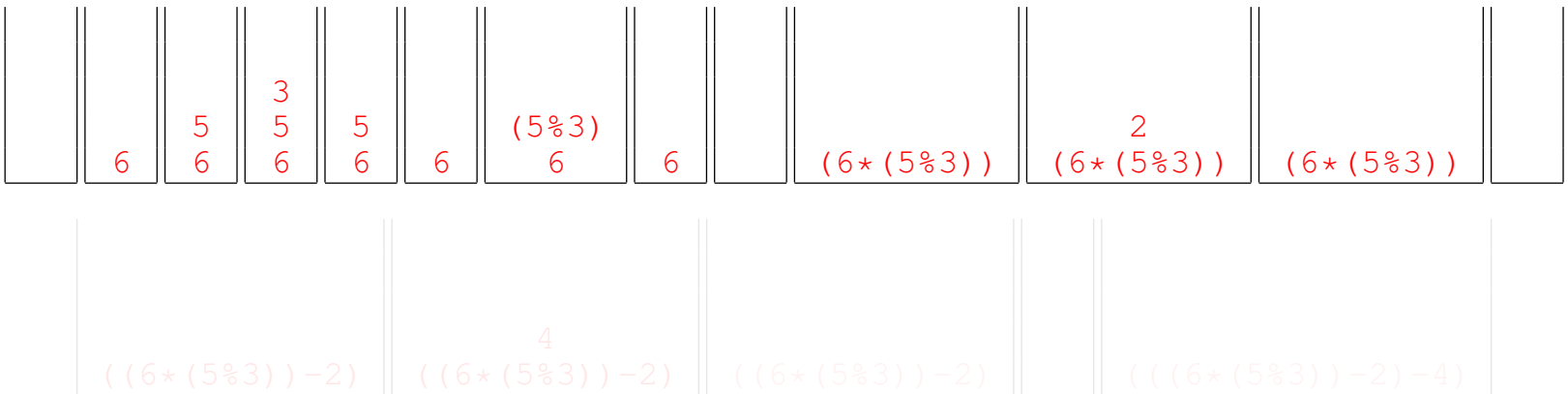
infixe



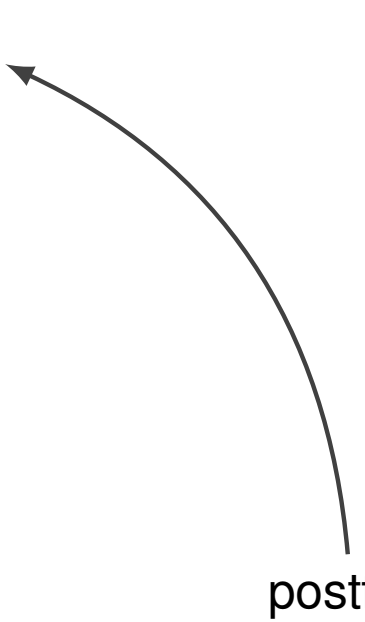
postfixe

```
6 5 3 % * 2 - 4 -
```

```
/* on lit l'expression de gauche à droite
 * si le symbole est un opérande, on l'empile
 * sinon (c'est un opérateur (binaire))
 *   * on dépile (deux) opérandes
 *   * on construit l'infixe du nouvel opérande
 *     entouré de parenthèses
 *   * on l'empile
 */
```

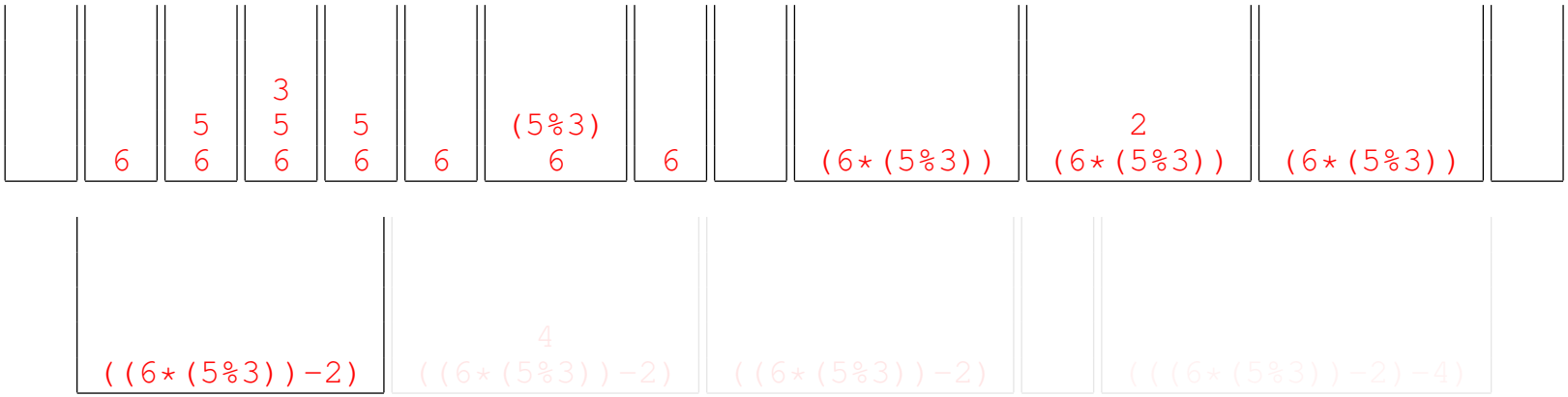


```
6 * ( 5 % 3 ) - 2 - 4
```

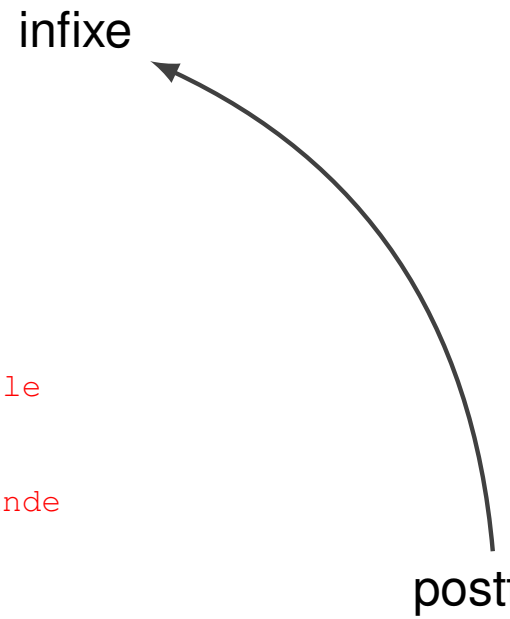



```
/* on lit l'expression de gauche à droite
 * si le symbole est un opérande, on l'empile
 * sinon (c'est un opérateur (binaire))
 *   * on dépile (deux) opérandes
 *   * on construit l'infixe du nouvel opérande
 *     entouré de parenthèses
 *   * on l'empile
 */
```

```
6 5 3 % * 2 - 4 -
      ▲
```

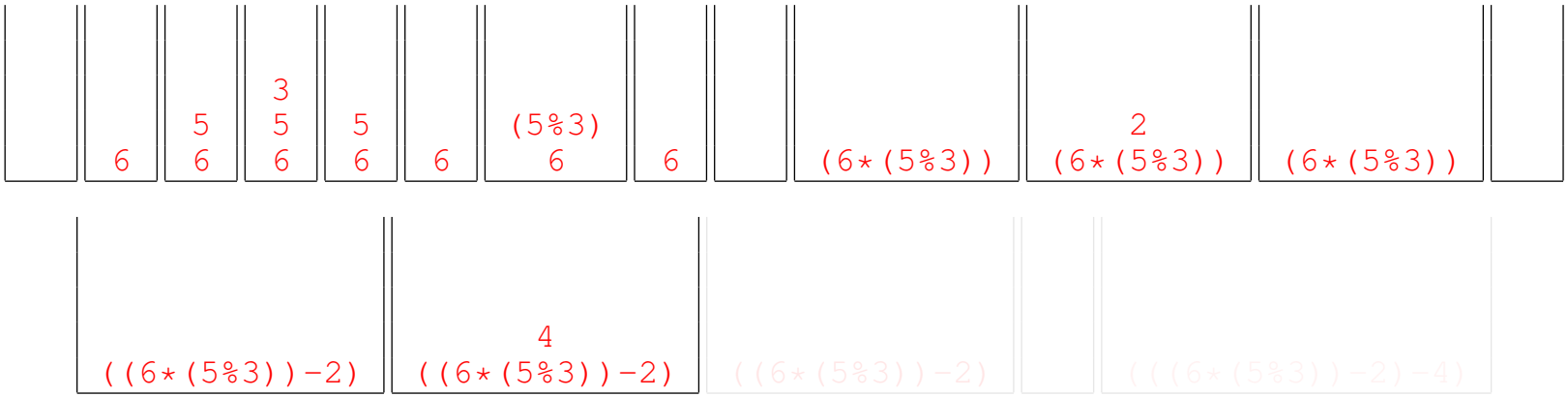


```
6 * ( 5 % 3 ) - 2 - 4
```



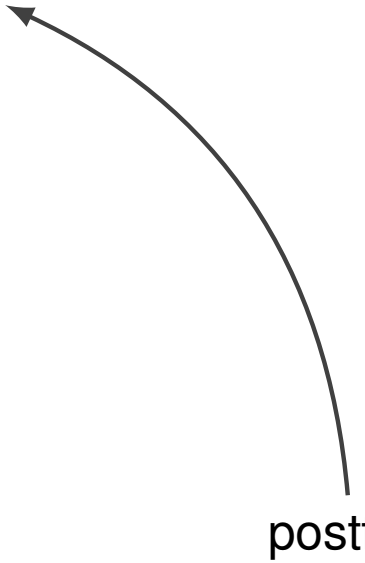

```
/* on lit l'expression de gauche à droite
 * si le symbole est un opérande, on l'empile
 * sinon (c'est un opérateur (binaire))
 *   * on dépile (deux) opérandes
 *   * on construit l'infixe du nouvel opérande
 *     entouré de parenthèses
 *   * on l'empile
 */
```

```
6 5 3 % * 2 - 4 -
      ▲
```



```
6 * ( 5 % 3 ) - 2 - 4
```

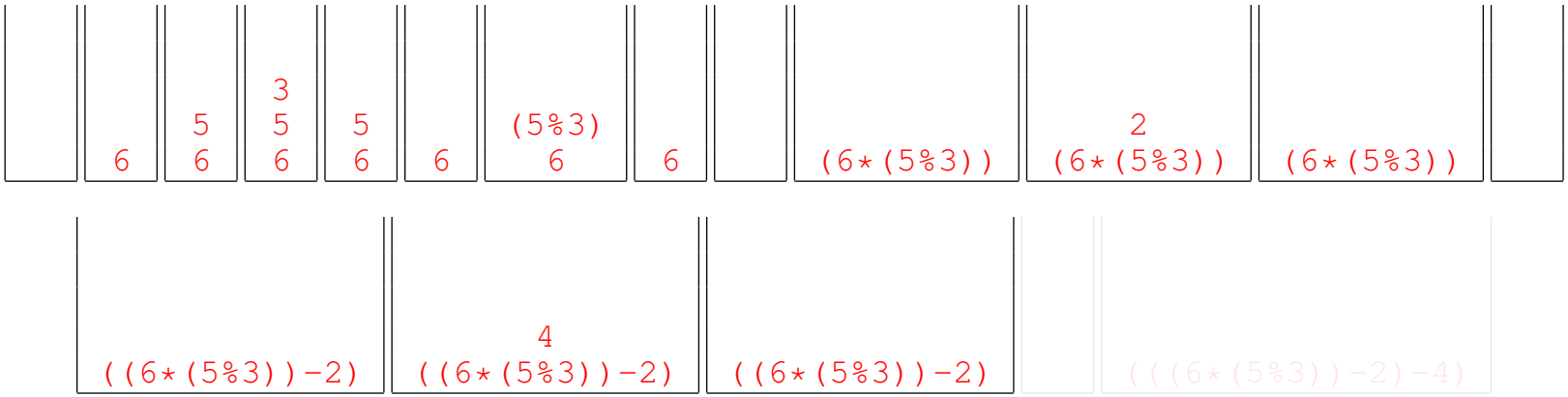
infixe



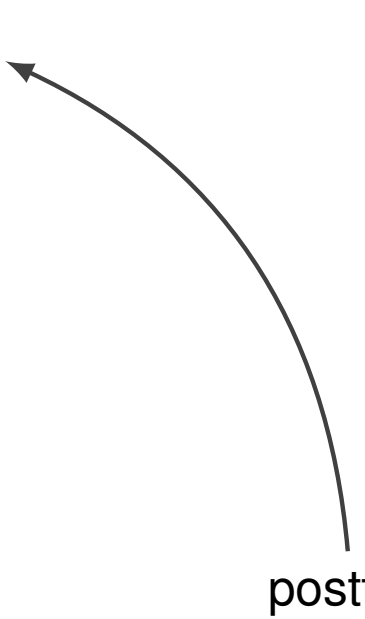
postfixe

```
6 5 3 % * 2 - 4 -
```

```
/* on lit l'expression de gauche à droite
 * si le symbole est un opérande, on l'empile
 * sinon (c'est un opérateur (binaire))
 *   * on dépile (deux) opérandes
 *   * on construit l'infixe du nouvel opérande
 *     entouré de parenthèses
 *   * on l'empile
 */
```

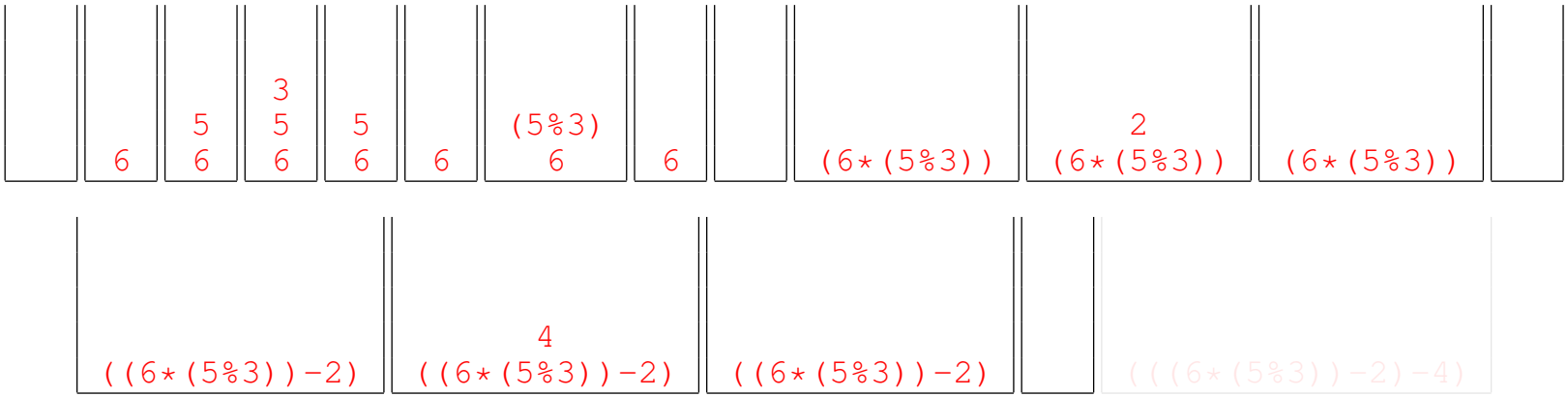


```
6 * ( 5 % 3 ) - 2 - 4
```

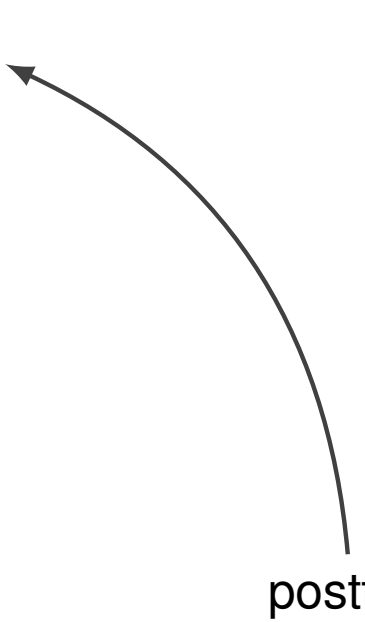



```
/* on lit l'expression de gauche à droite
 * si le symbole est un opérande, on l'empile
 * sinon (c'est un opérateur (binaire))
 *   * on dépile (deux) opérandes
 *   * on construit l'infixe du nouvel opérande
 *     entouré de parenthèses
 *   * on l'empile
 */
```

```
6 5 3 % * 2 - 4 -
  ▲
```

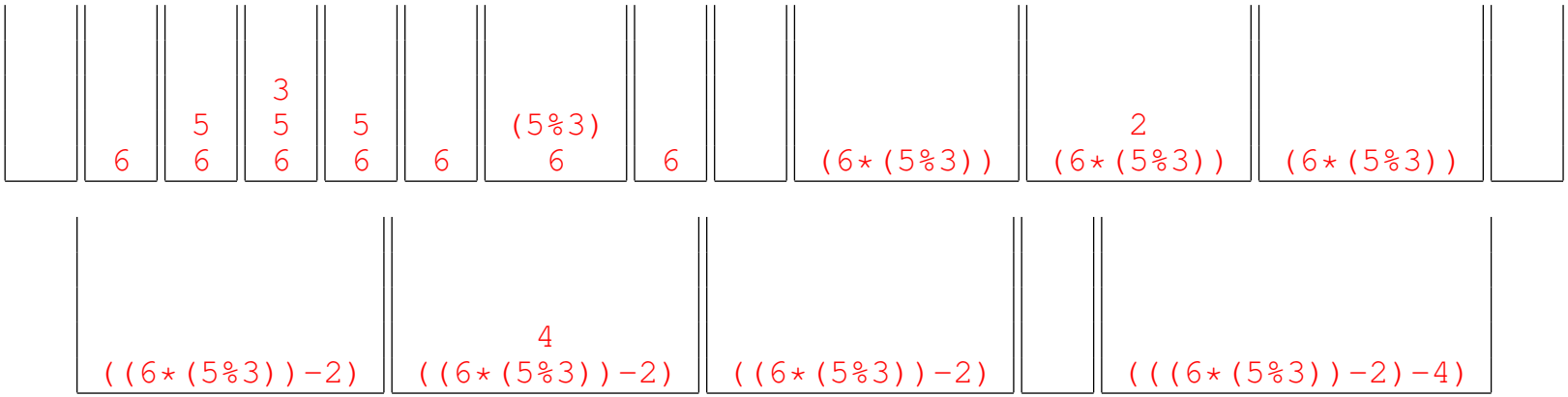


```
6 * ( 5 % 3 ) - 2 - 4
```

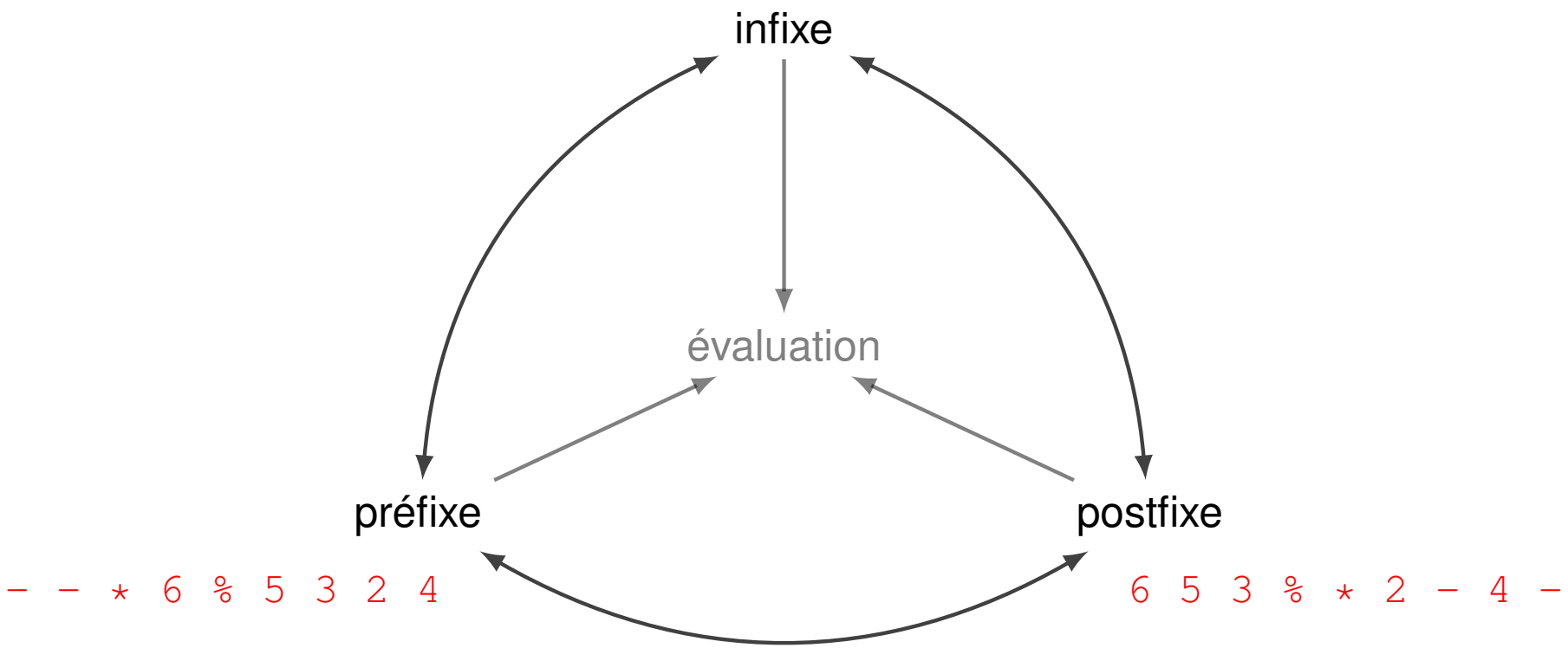



```
/* on lit l'expression de gauche à droite
 * si le symbole est un opérande, on l'empile
 * sinon (c'est un opérateur (binaire))
 *   * on dépile (deux) opérandes
 *   * on construit l'infixe du nouvel opérande
 *     entouré de parenthèses
 *   * on l'empile
 */
```

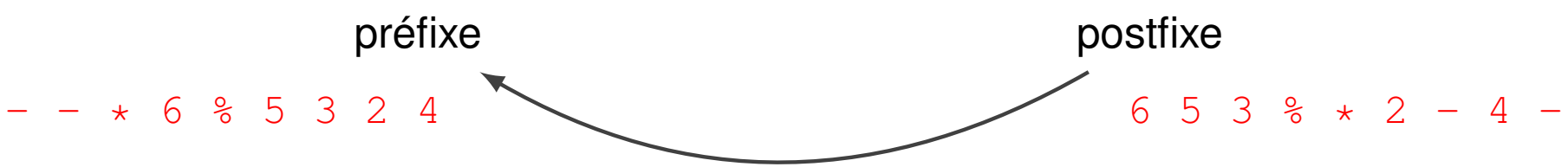
```
6 5 3 % * 2 - 4 -
  ▲
```

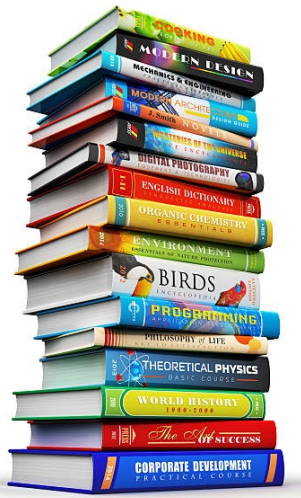


6 * (5 % 3) - 2 - 4



```
/* on lit l'expression de gauche à droite
 * si le symbole est un opérande, on l'empile
 * sinon (c'est un opérateur (binaire))
 *   * on dépile (deux) opérandes
 *   * on construit la préfixe du nouvel opérande
 *   * on l'empile
 */
```





pile
(stack)

tas
(heap)