Concepts Informatiques

2018–2019

Matthieu Picantin

Tests et examens

- CC : résultat des 3 tests (ou plus) effectués en TD
- E0 : partiel (samedi 23 février ou 2 mars, à confirmer)
- E1 : examen mi-mai
- E2 : examen mi-juin

Notes finales

- Note session 1 : 20% CC + 20% E0 + 60% E1
- Note session 2 : max(E2, 20% CC + 80% E2)

Rappel

pas de note $\Longrightarrow$ pas de moyenne $\Longrightarrow$ pas de semestre

`moodlesupd.script.univ-paris-diderot.fr`

Tests et examens

- CC : résultat des 3 tests (ou plus) effectués en TD
- E0 : partiel (samedi 23 février ou 2 mars, à confirmer)
- E1 : examen mi-mai
- E2 : examen mi-juin

Notes finales

- Note session 1 : 20% CC + 20% E0 + 60% E1
- Note session 2 : max(E2, 20% CC + 80% E2)

Rappel

pas de note $\Longrightarrow$ pas de moyenne $\Longrightarrow$ pas de semestre

`moodlesupd.script.univ-paris-diderot.fr`

Tests et examens

- CC : résultat des 3 tests (ou plus) effectués en TD
- E0 : partiel (samedi 23 février ou 2 mars, à confirmer)
- E1 : examen mi-mai
- E2 : examen mi-juin

Notes finales

- Note session 1 : 20% CC + 20% E0 + 60% E1
- Note session 2 : max(E2, 20% CC + 80% E2)

Rappel

pas de note $\Longrightarrow$ pas de moyenne $\Longrightarrow$ pas de semestre

moodlesupd.script.univ-paris-diderot.fr

Tests et examens

- CC : résultat des 3 tests (ou plus) effectués en TD
- E0 : partiel (samedi 23 février ou 2 mars, à confirmer)
- E1 : examen mi-mai
- E2 : examen mi-juin

Notes finales

- Note session 1 : 20% CC + 20% E0 + 60% E1
- Note session 2 : max(E2, 20% CC + 80% E2)

Rappel

pas de note $\Longrightarrow$ pas de moyenne $\Longrightarrow$ pas de semestre

`moodlesupd.script.univ-paris-diderot.fr`

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

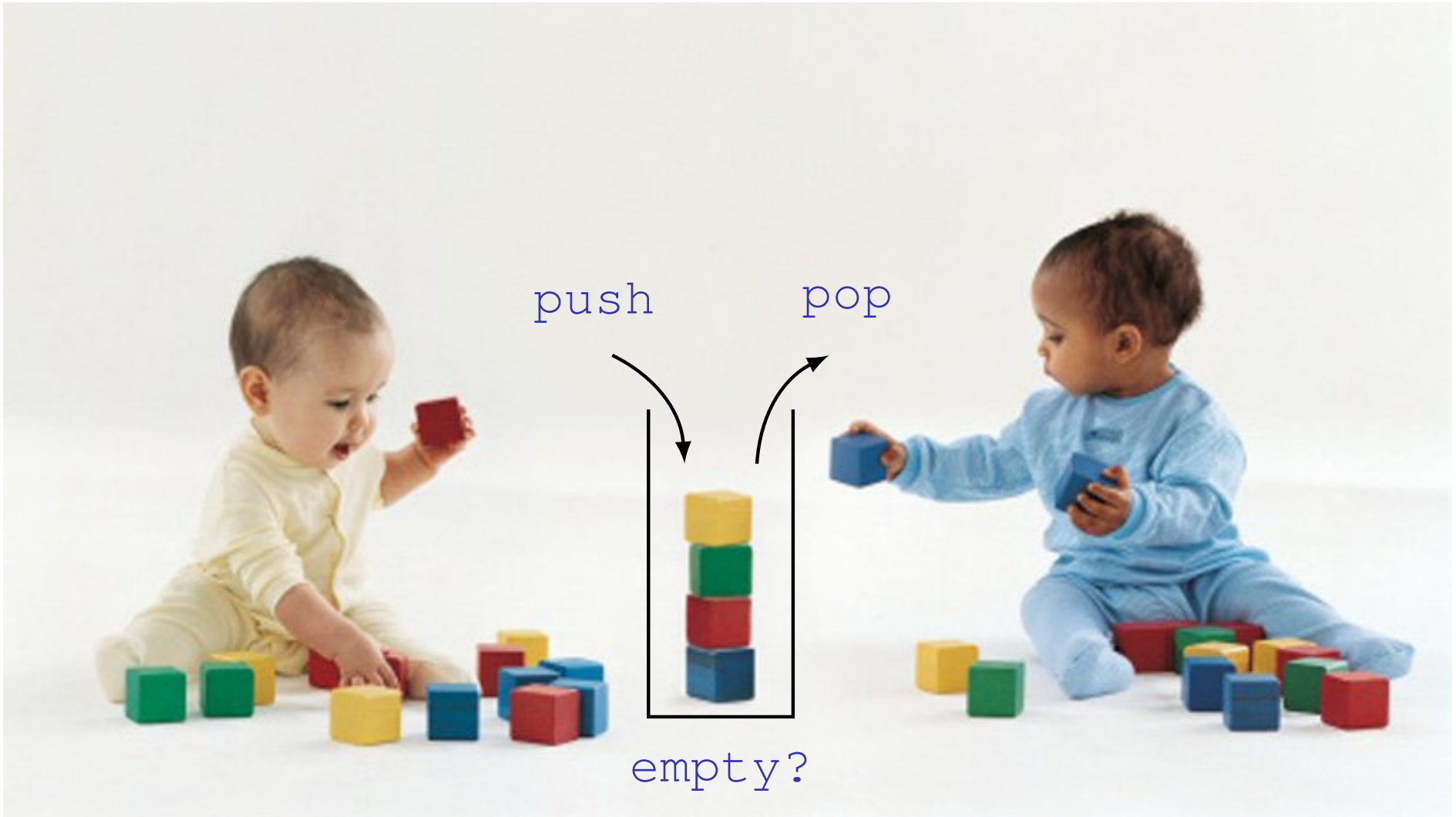
pensée

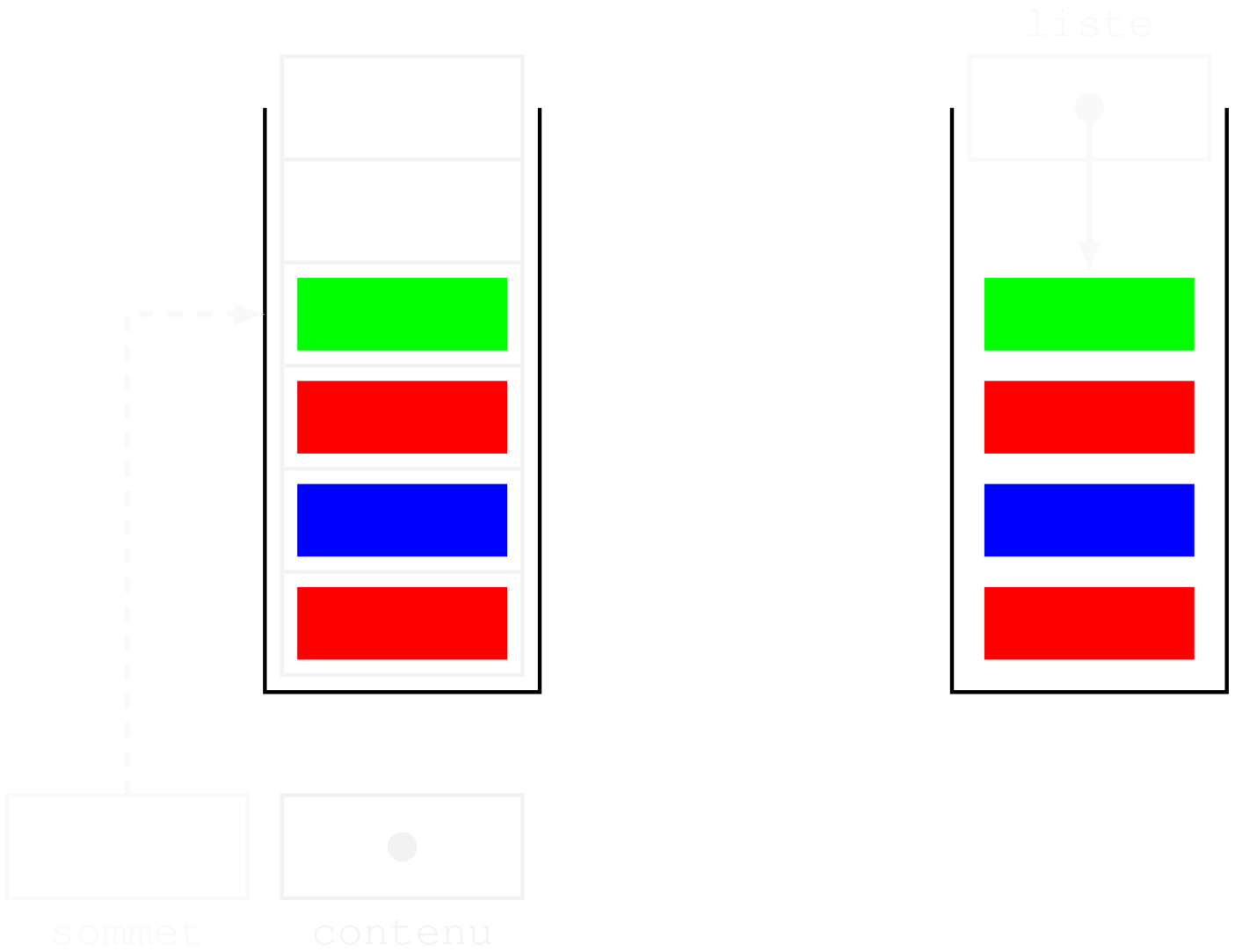
calcul
récursion
fonction
objet
⋮

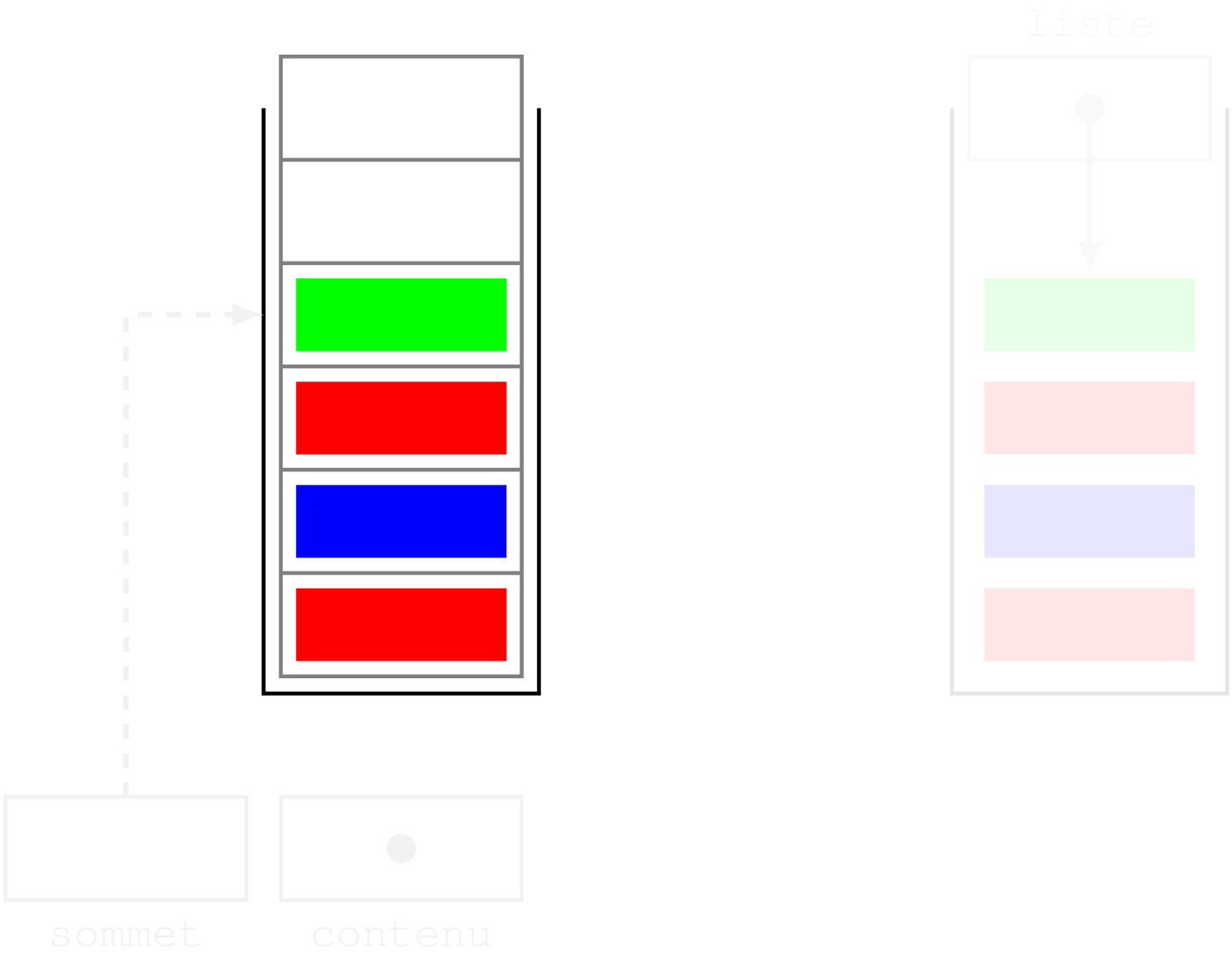
machine

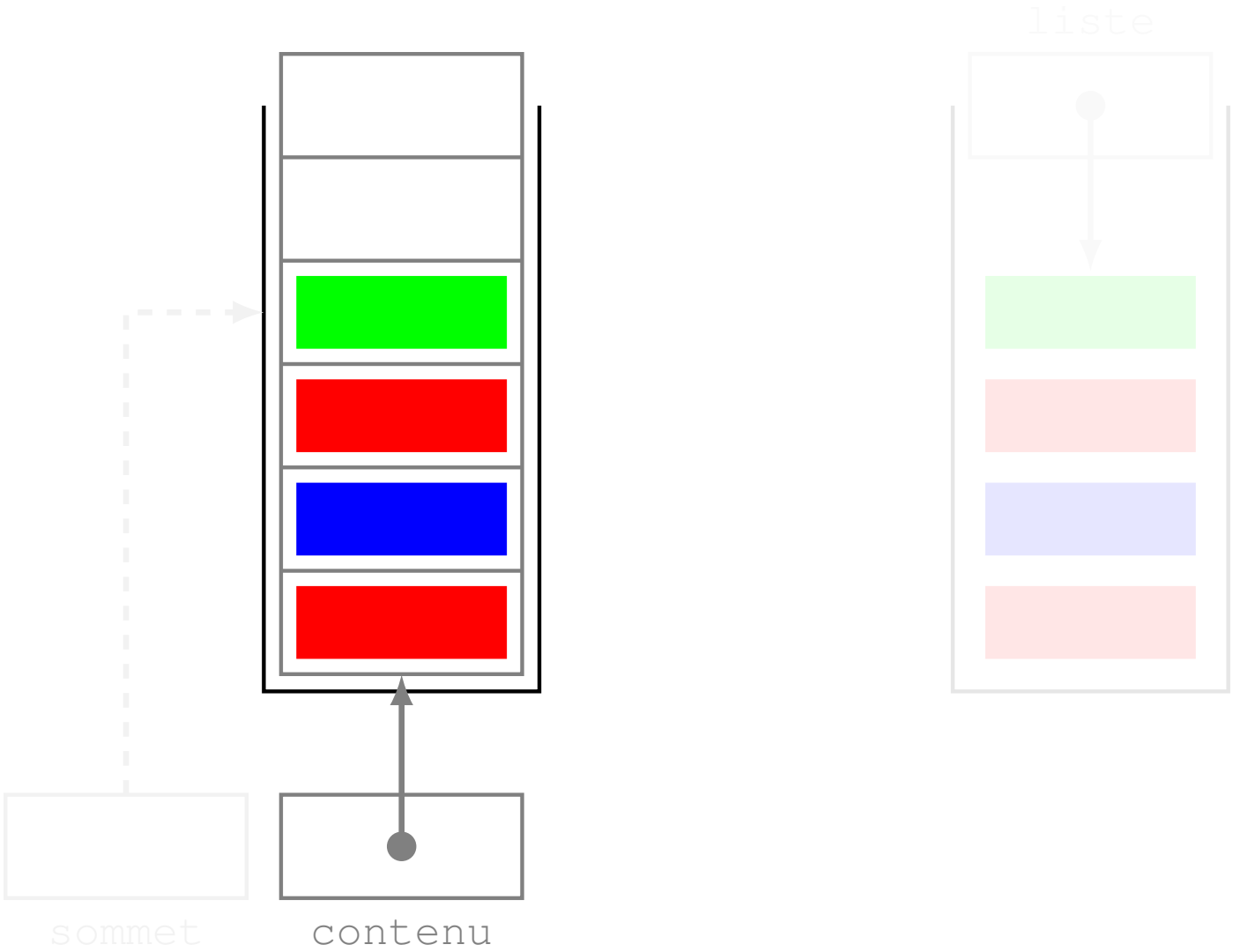
circuit
pile
registre
mémoire
⋮

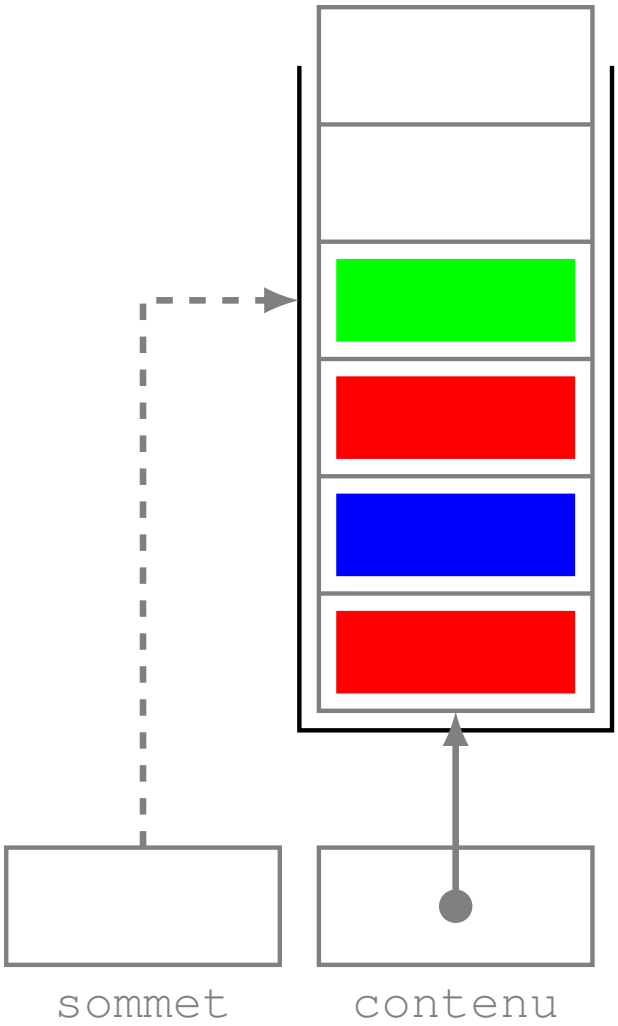
```
10111000 00000001 00000000
00000000 00000000 10111010
00000010 00000000 00000000
00000000 00111001 11011010
01111111 00000110 00001111
10101111 11000010 01000010
11101011 11110110 11000011
```


push
pop
empty?

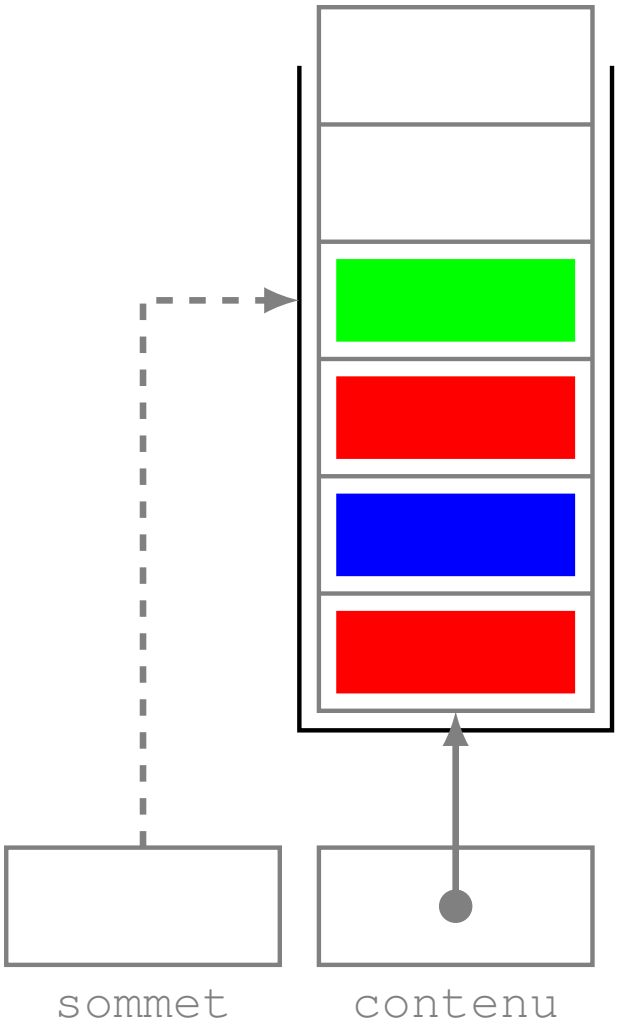

liste
sommet
contenu

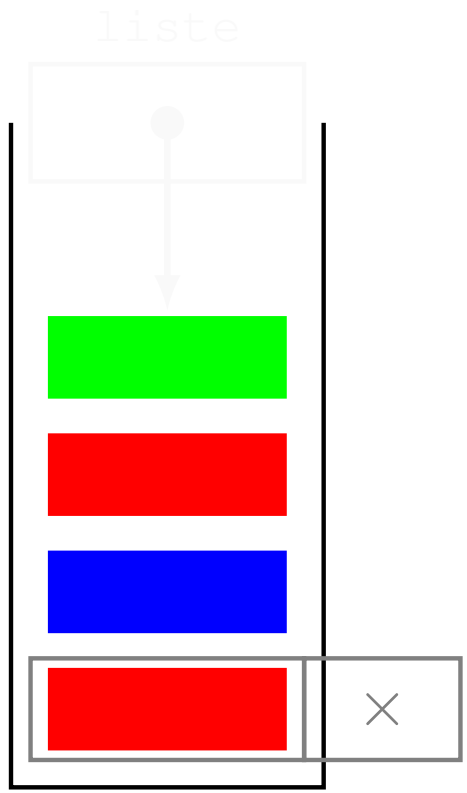

liste
sommet
contenu

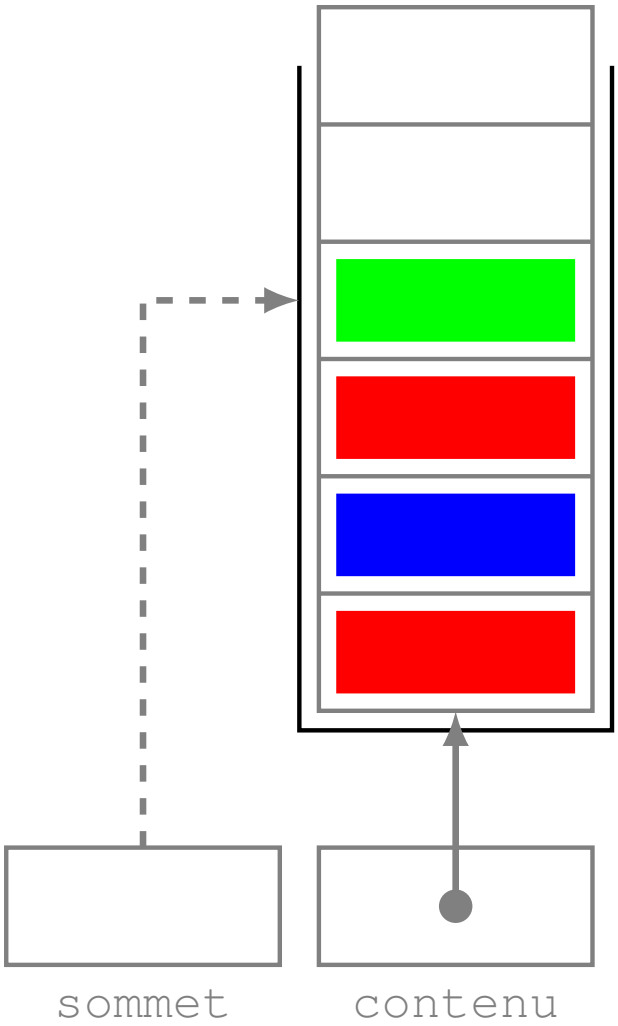

liste
sommet
contenu

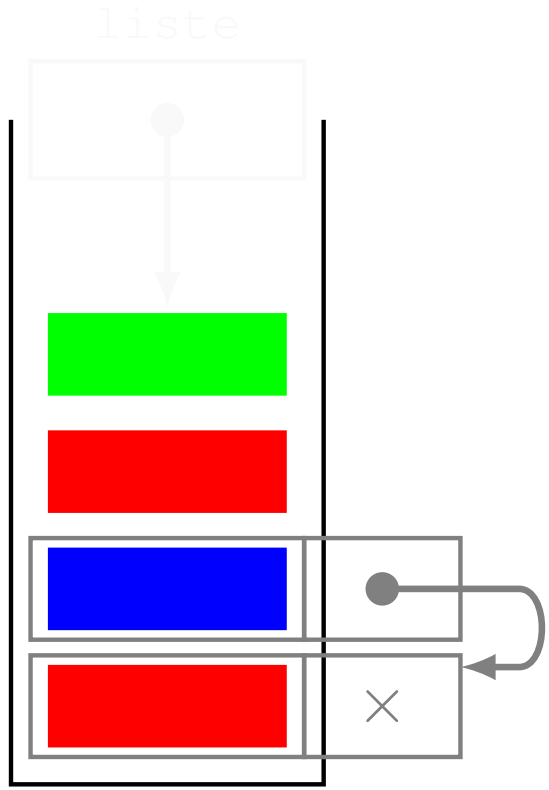

sommet
contenu

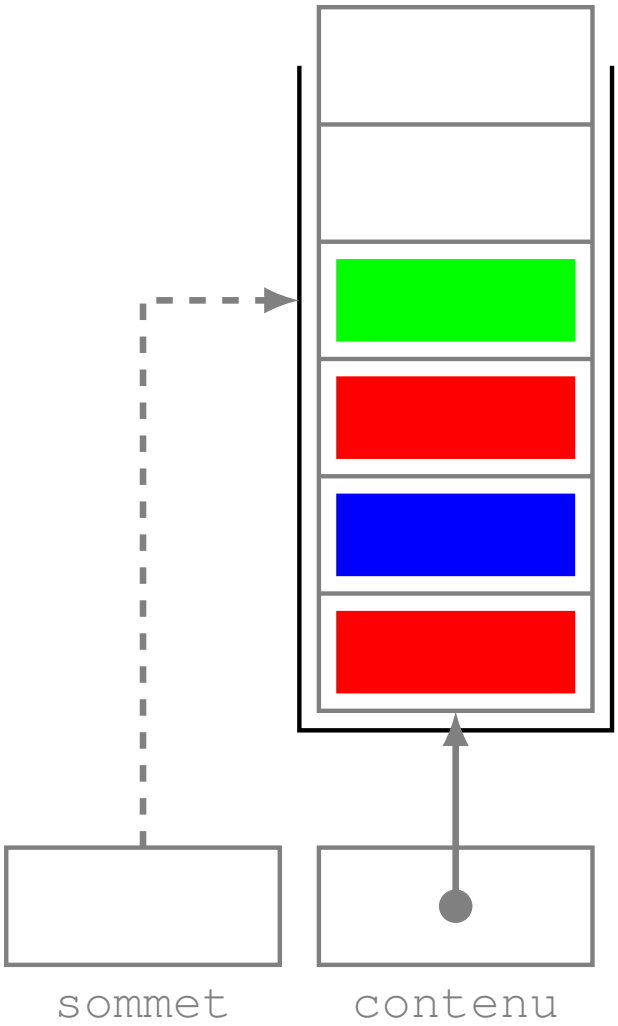
liste

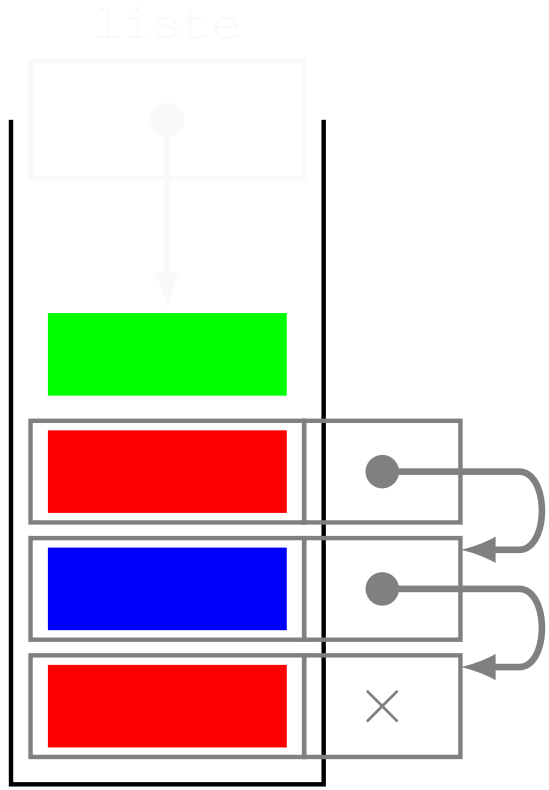

sommet
contenu

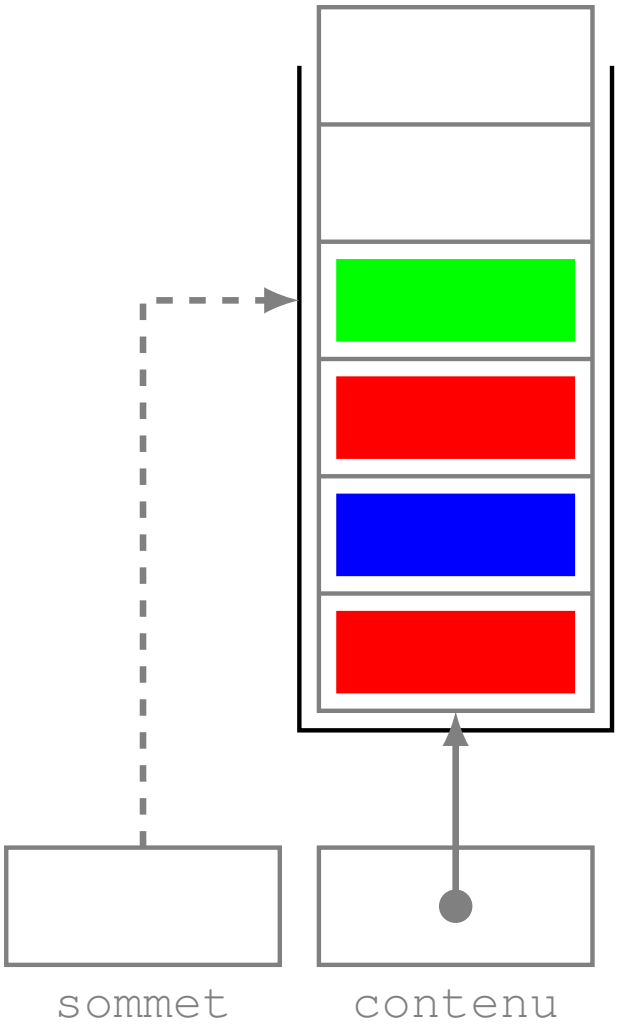

liste

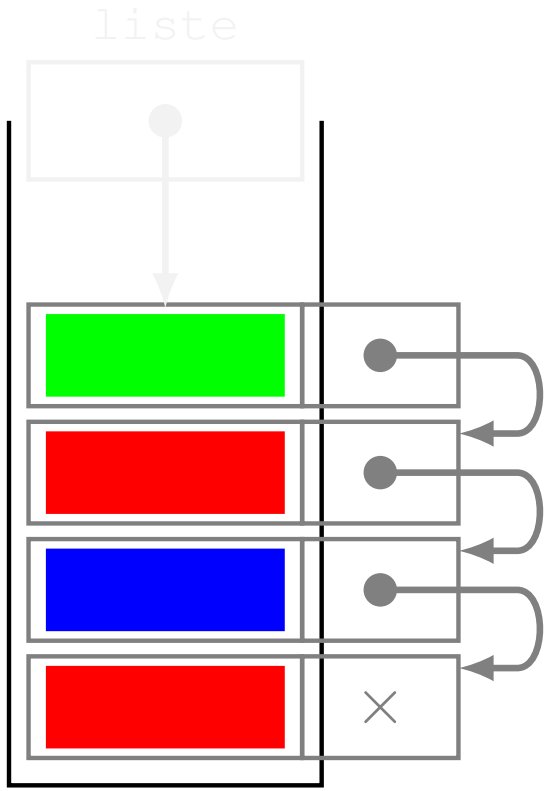

sommet
contenu

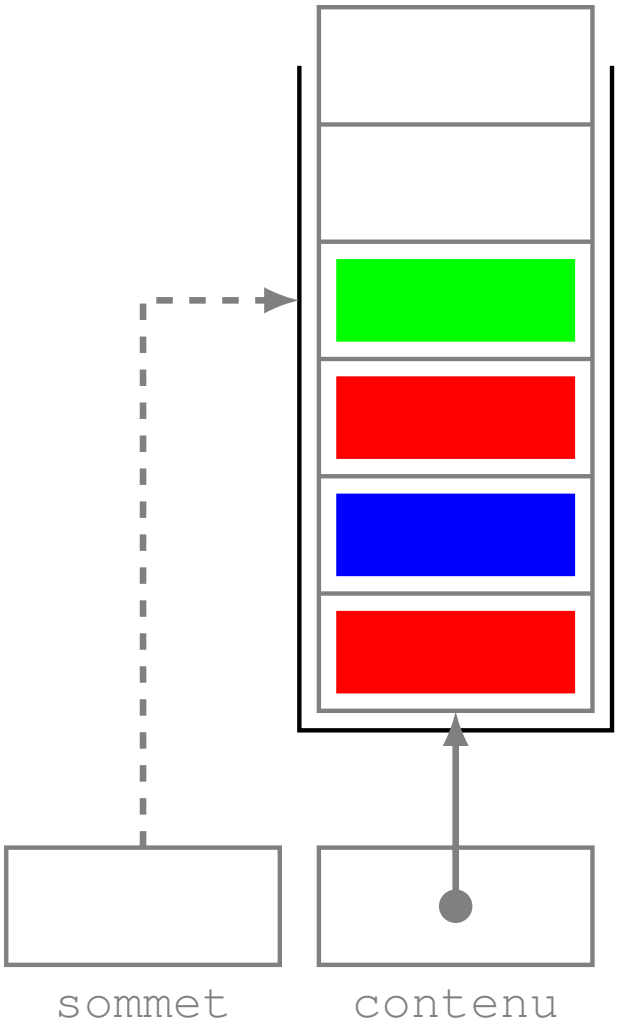

liste

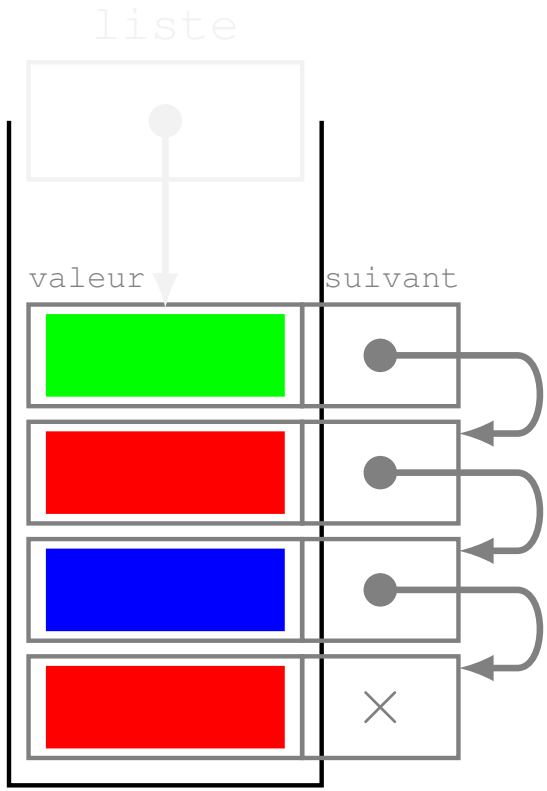

sommet
contenu

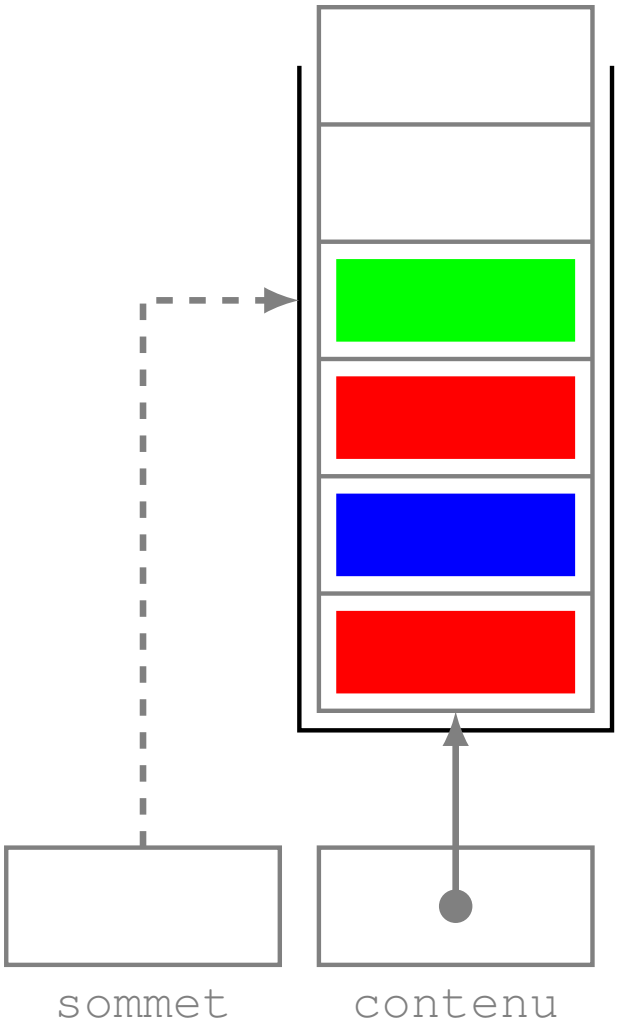

liste

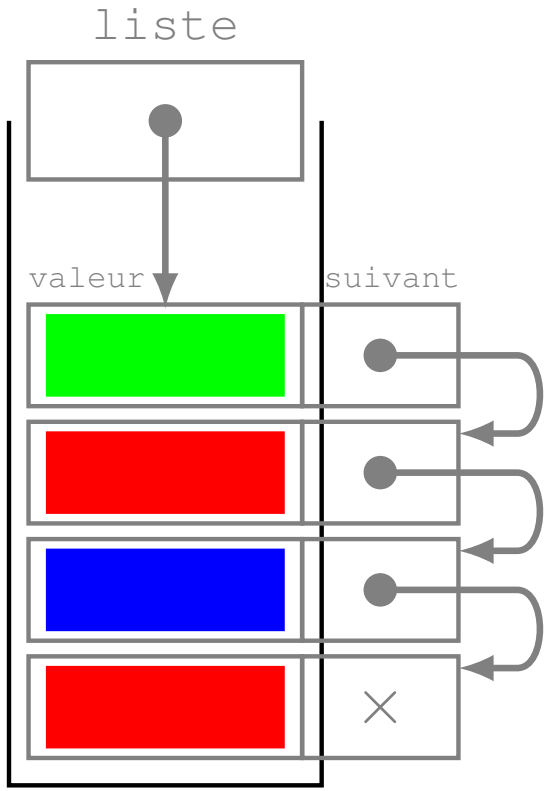

sommet
contenu

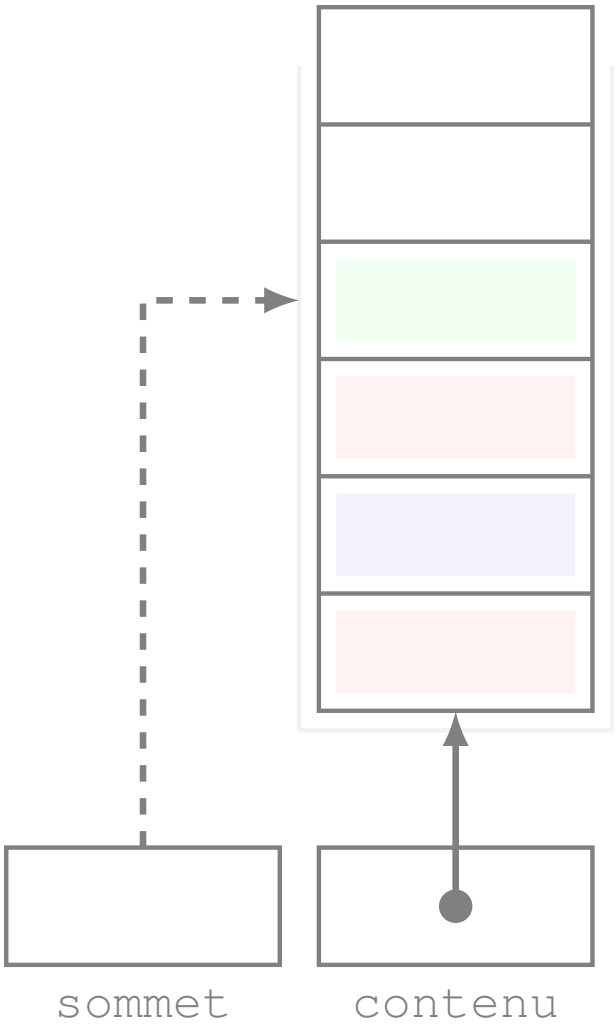

liste

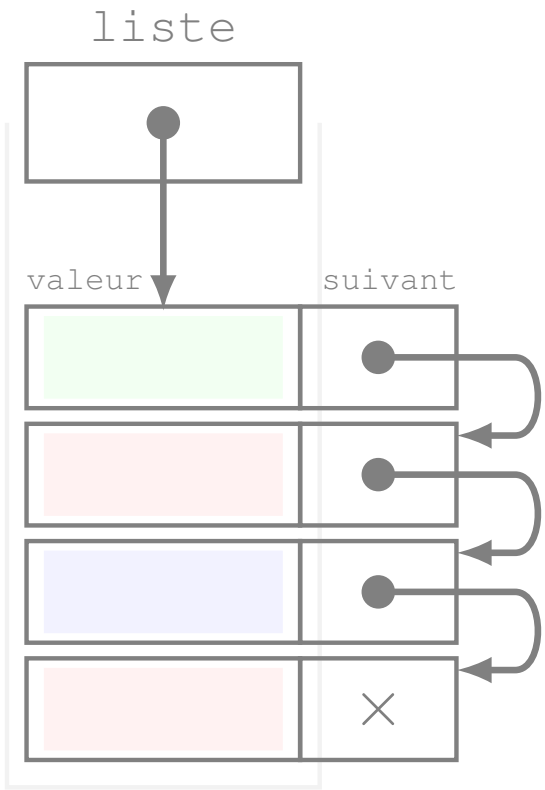

sommet
contenu

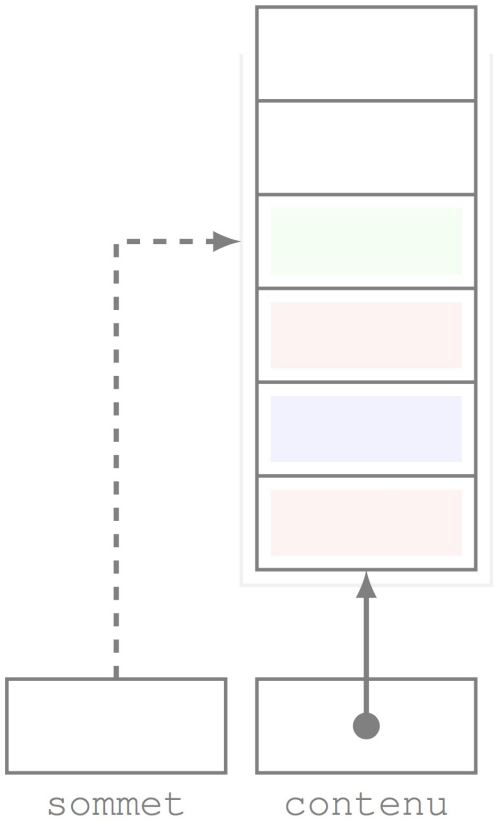

liste
valeur
suivant

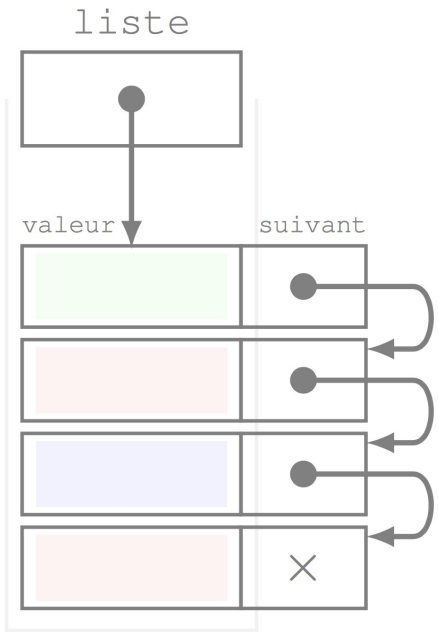

sommet
contenu


liste
valeur
suivant

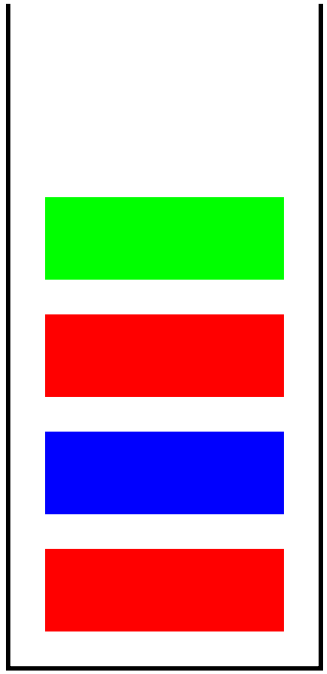

sommet
contenu

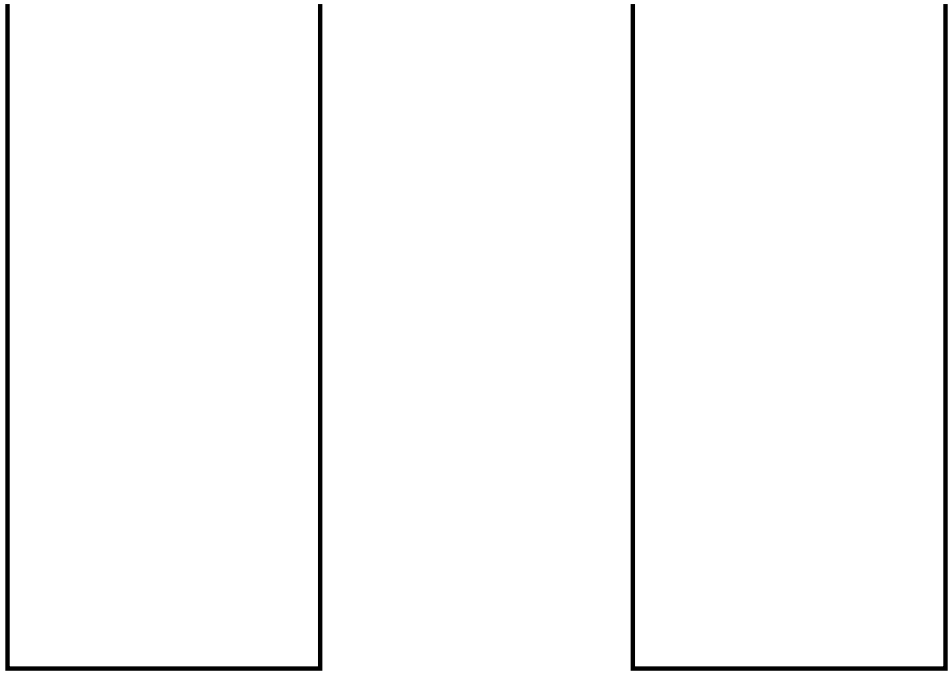

liste
valeur
suivant

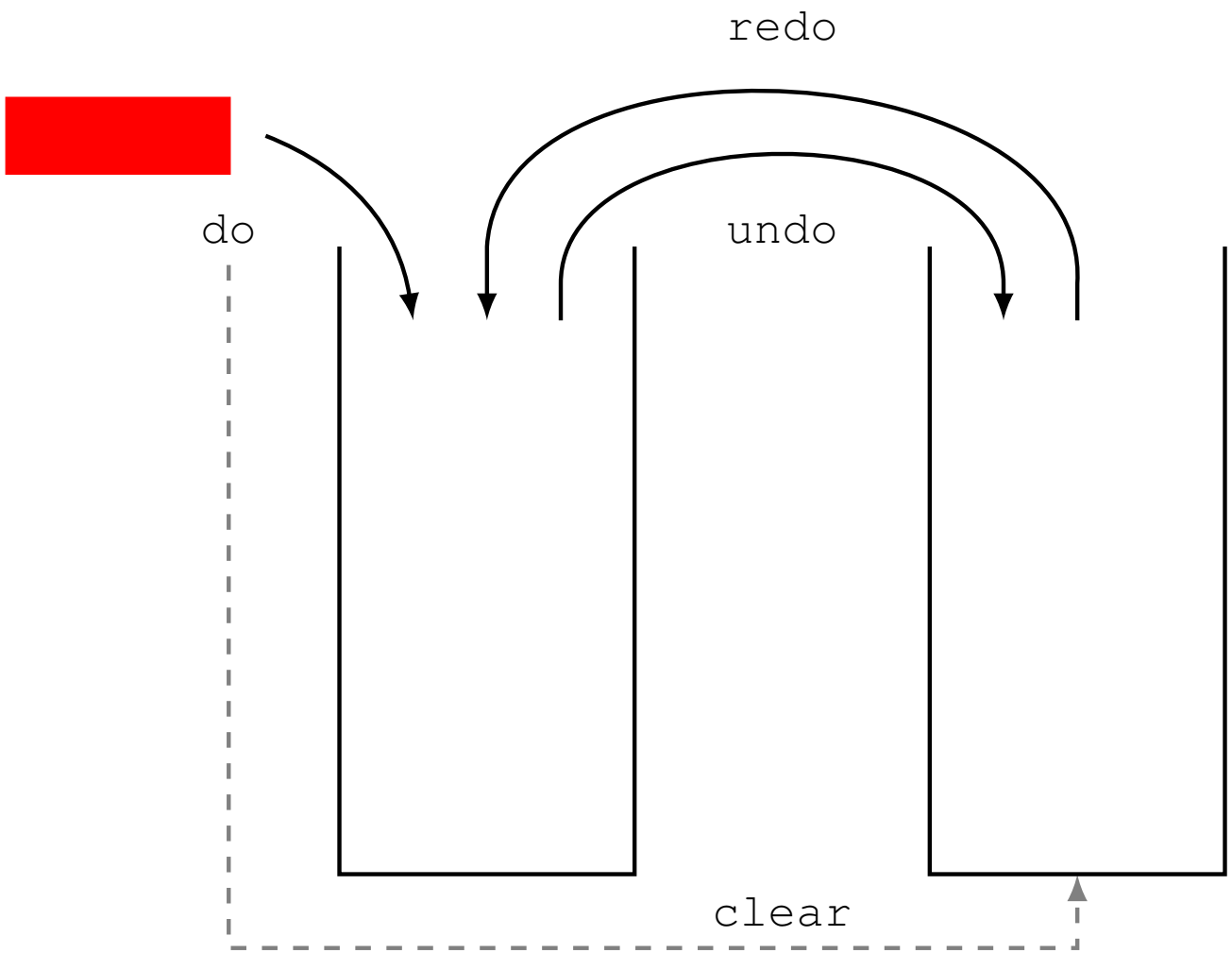

sommet
contenu

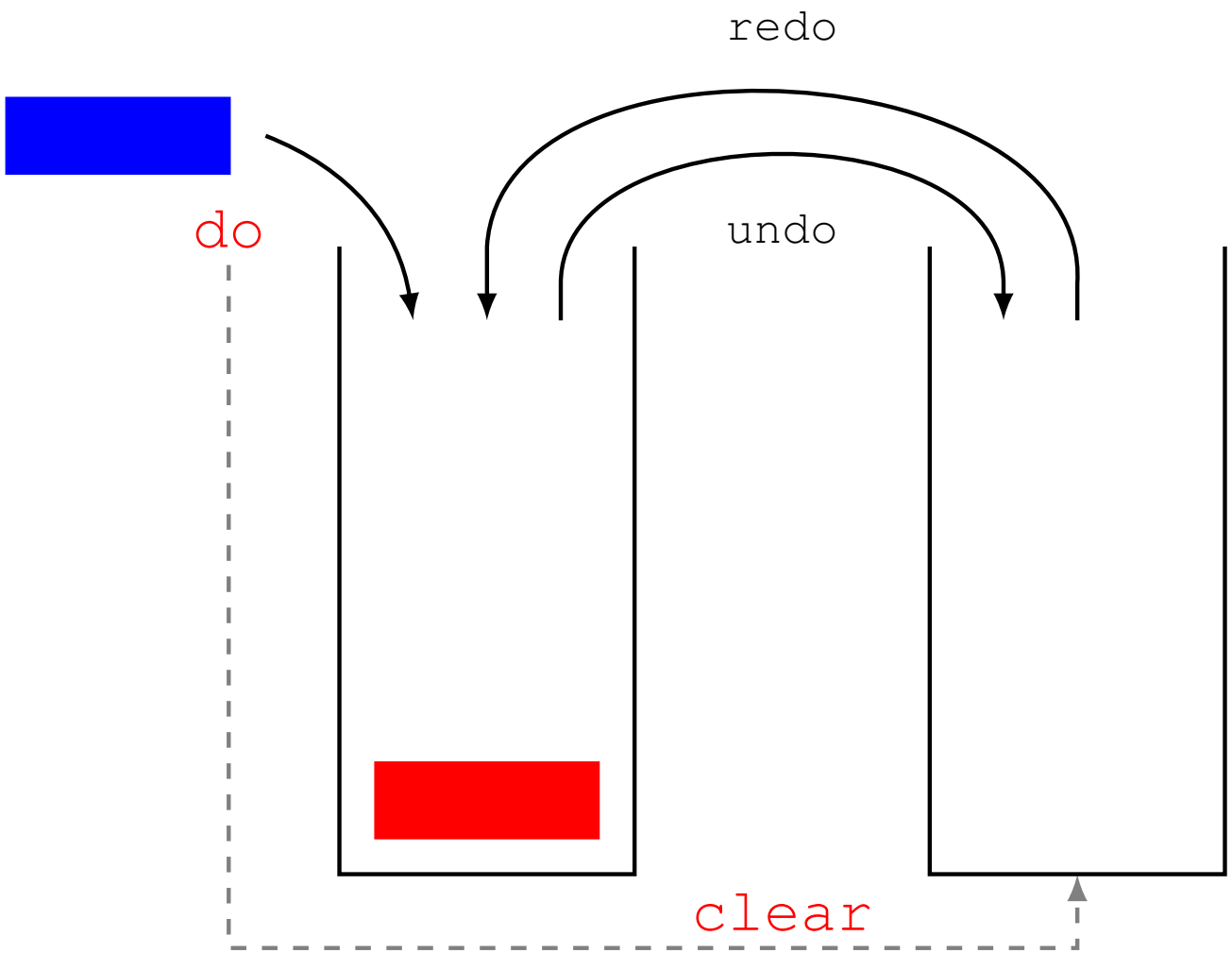

liste
valeur
suivant

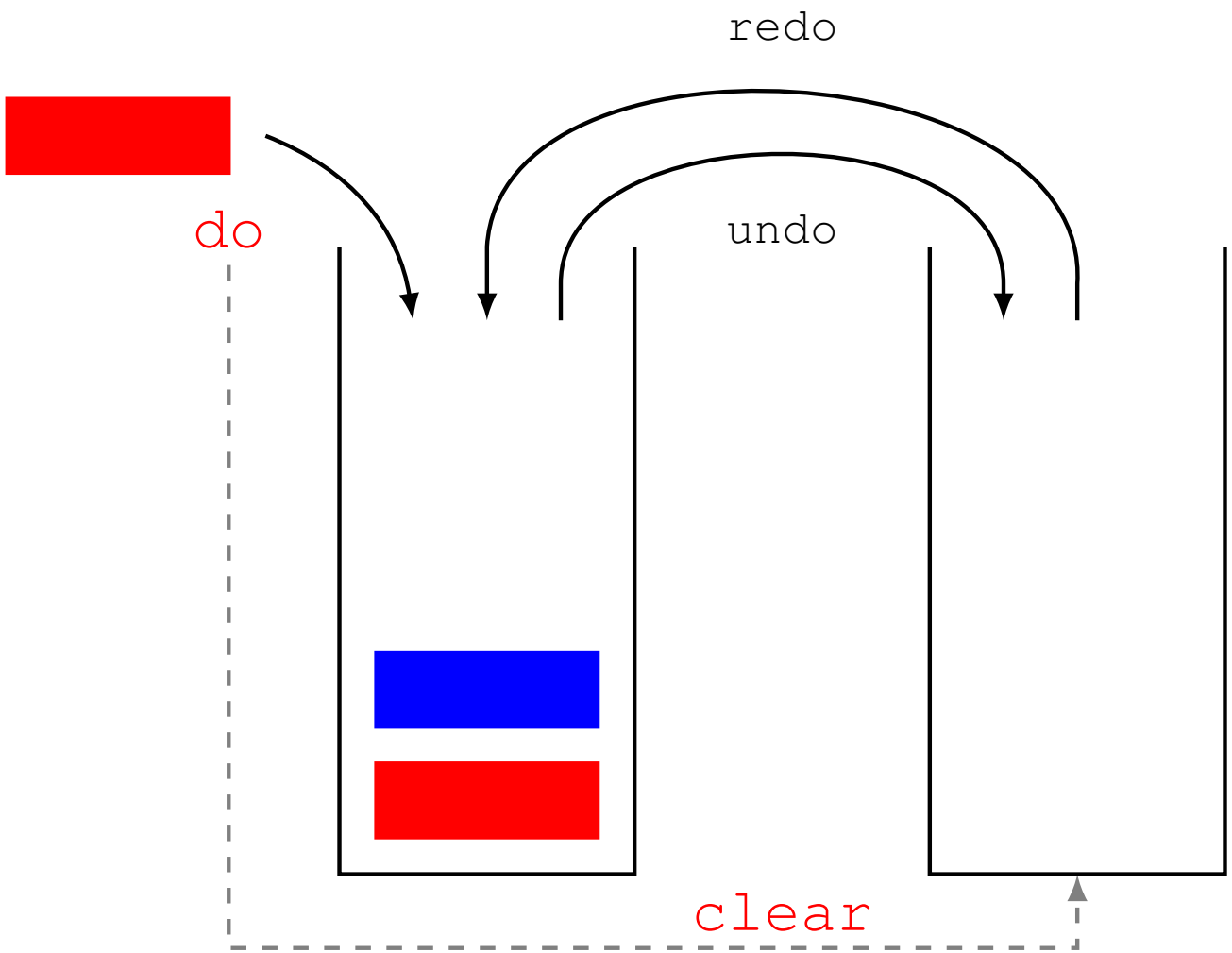

these implementations are not the ones you are looking for

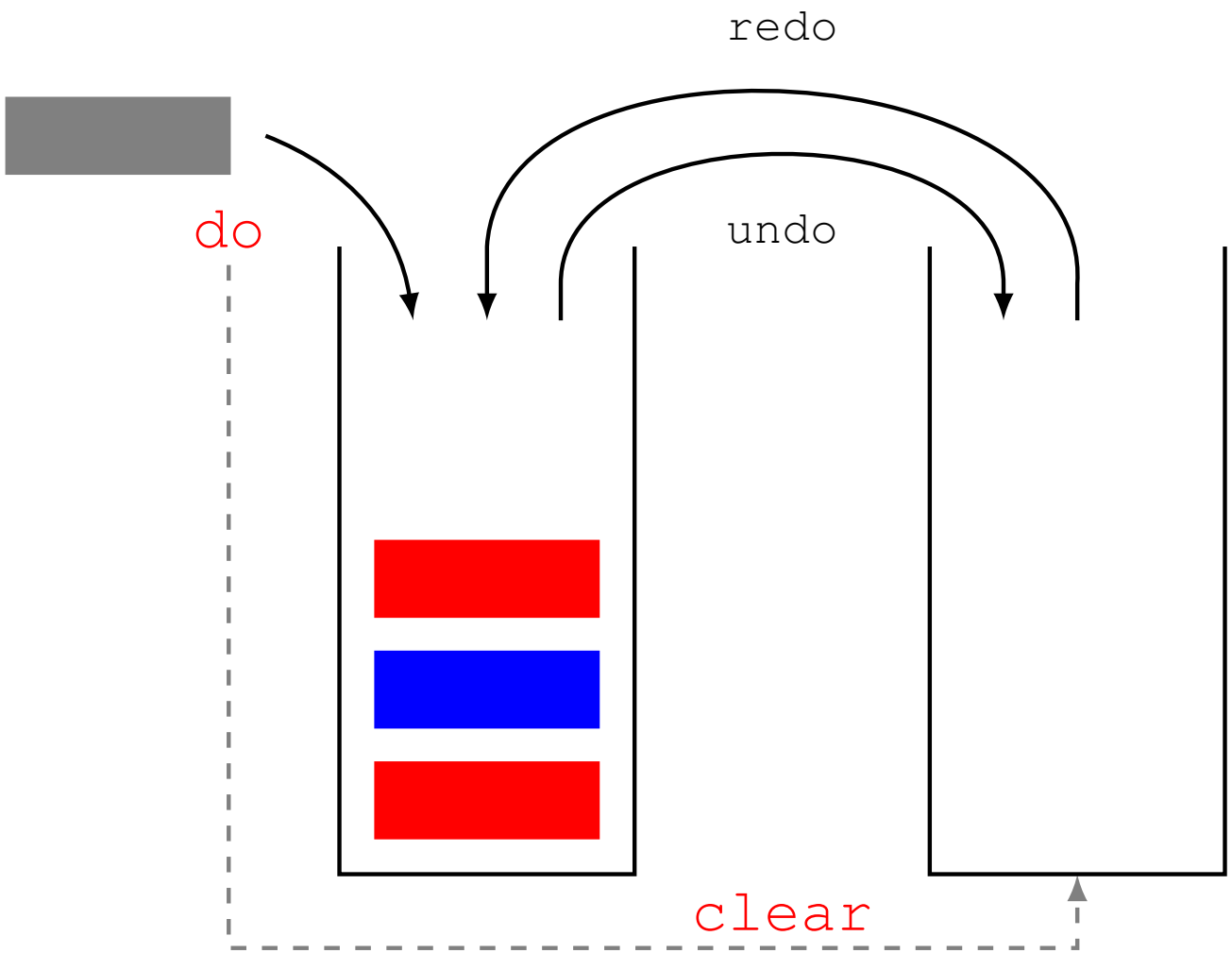


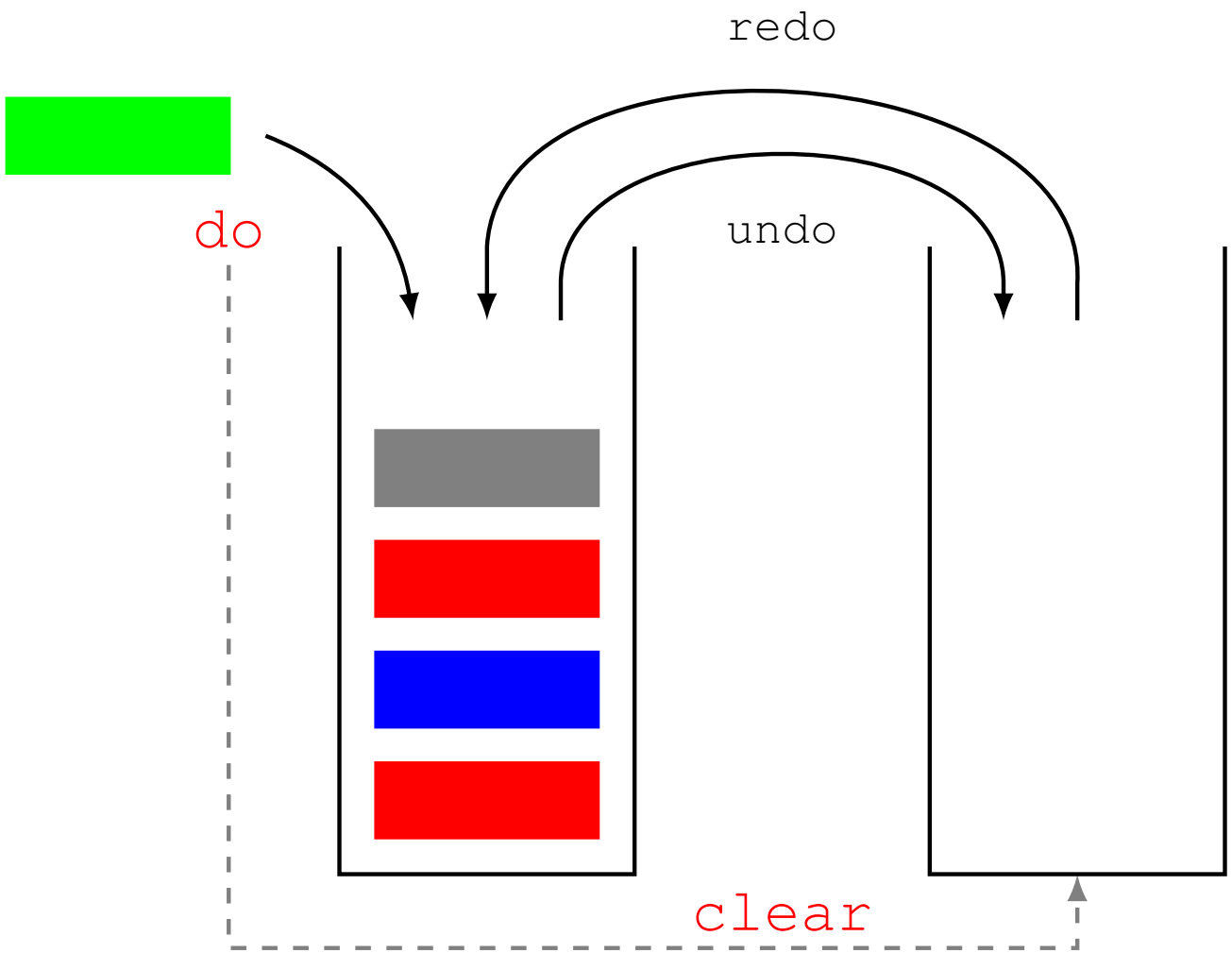


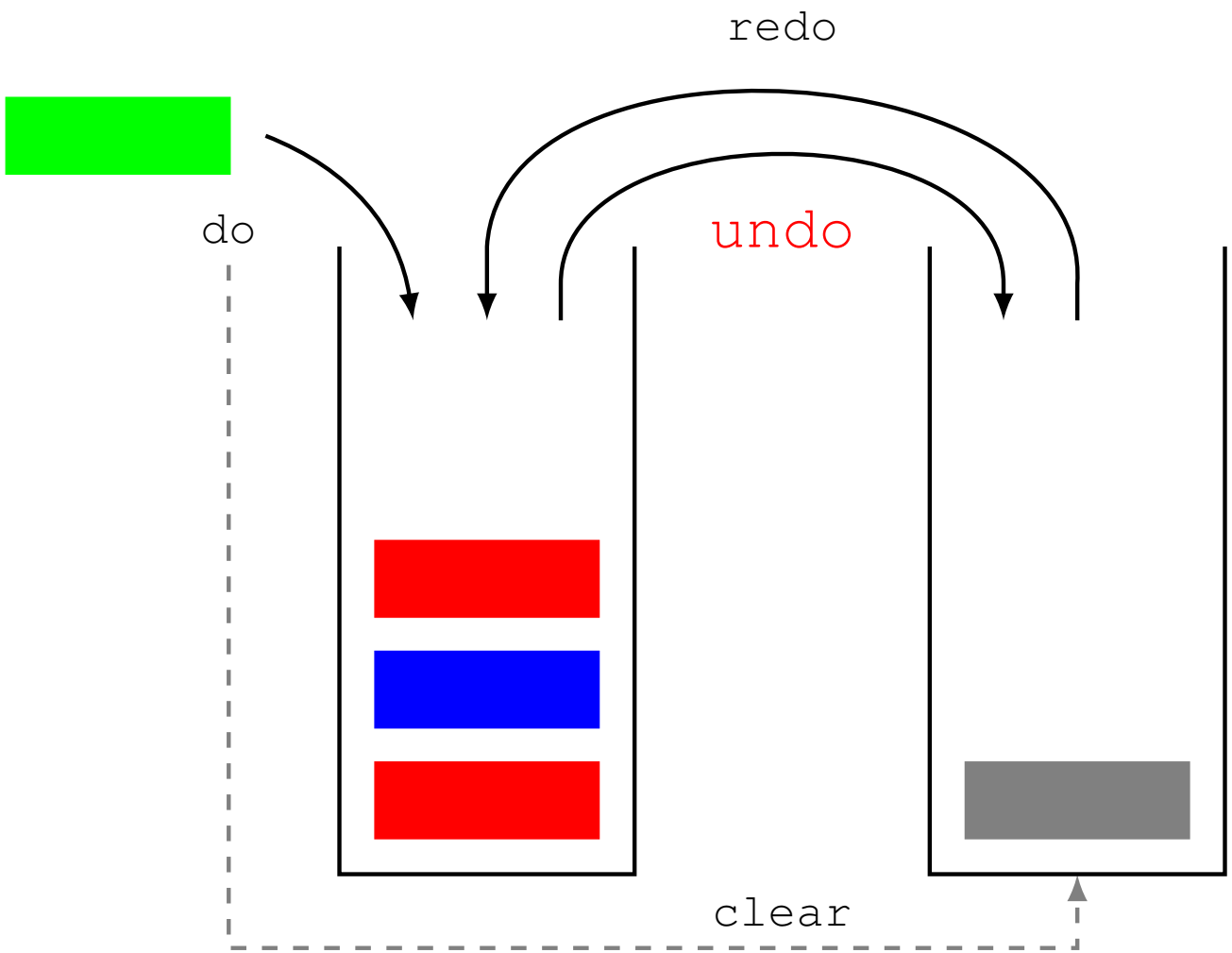

redo
do
undo
clear

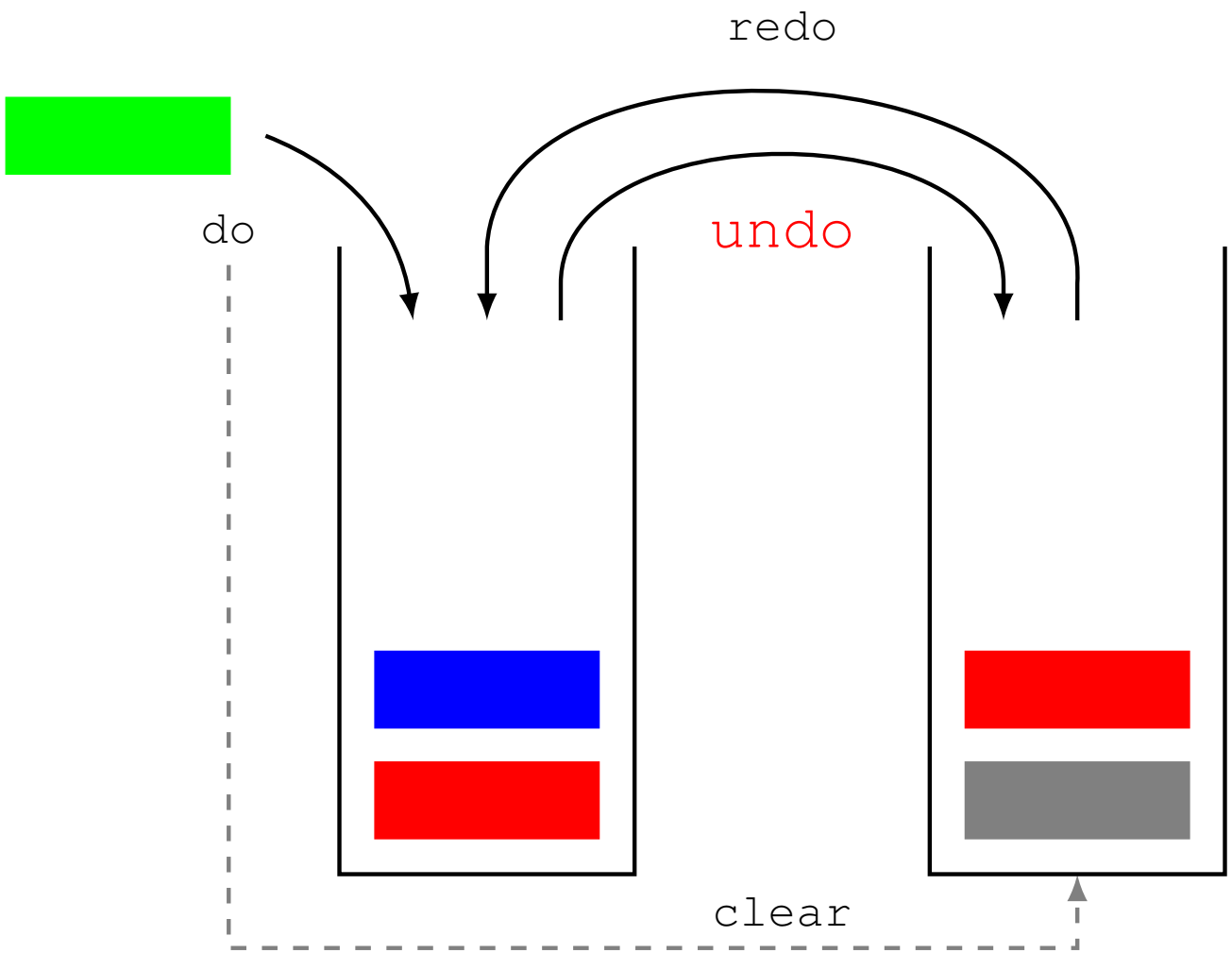

redo
do
undo
clear

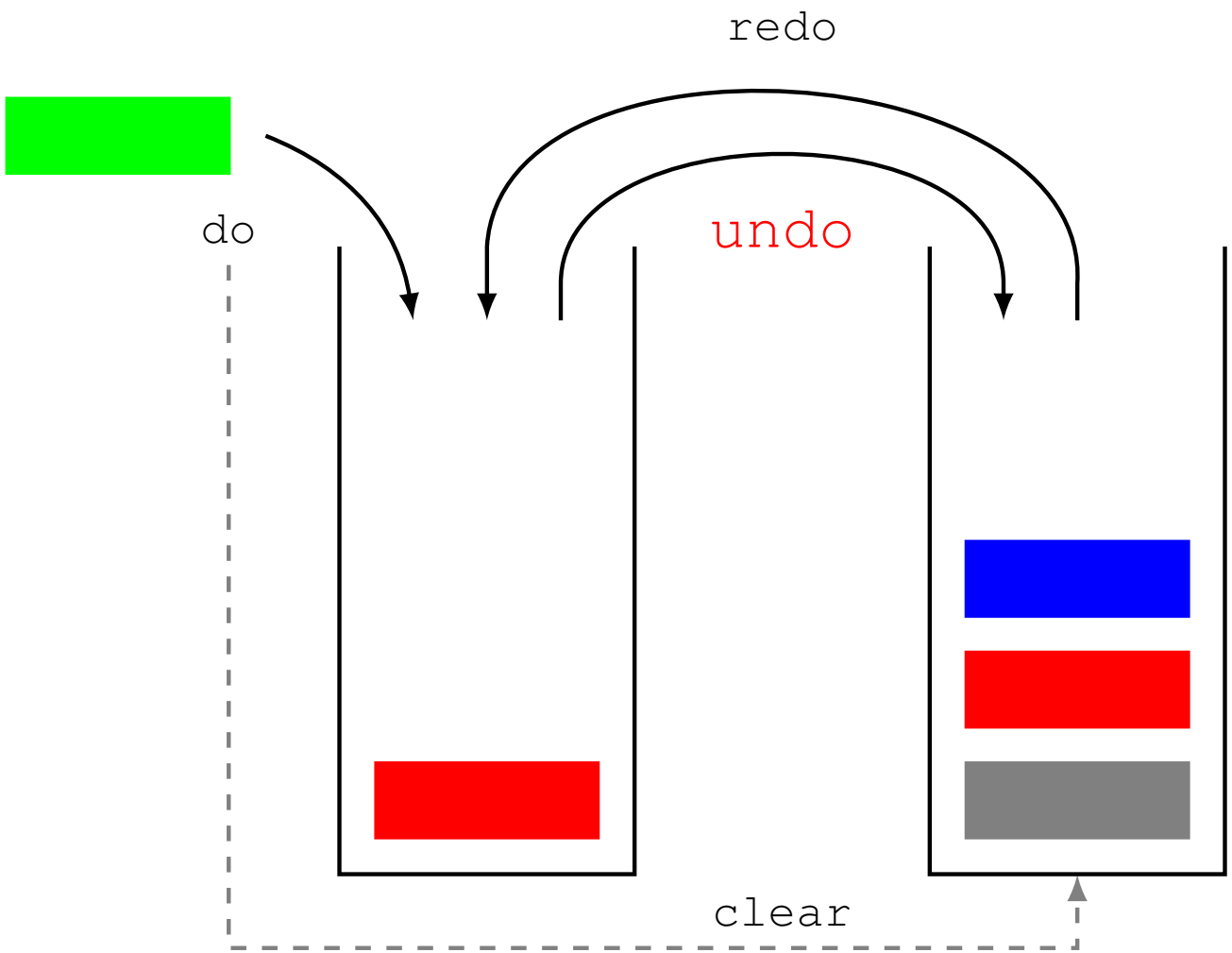

redo
do
undo
clear

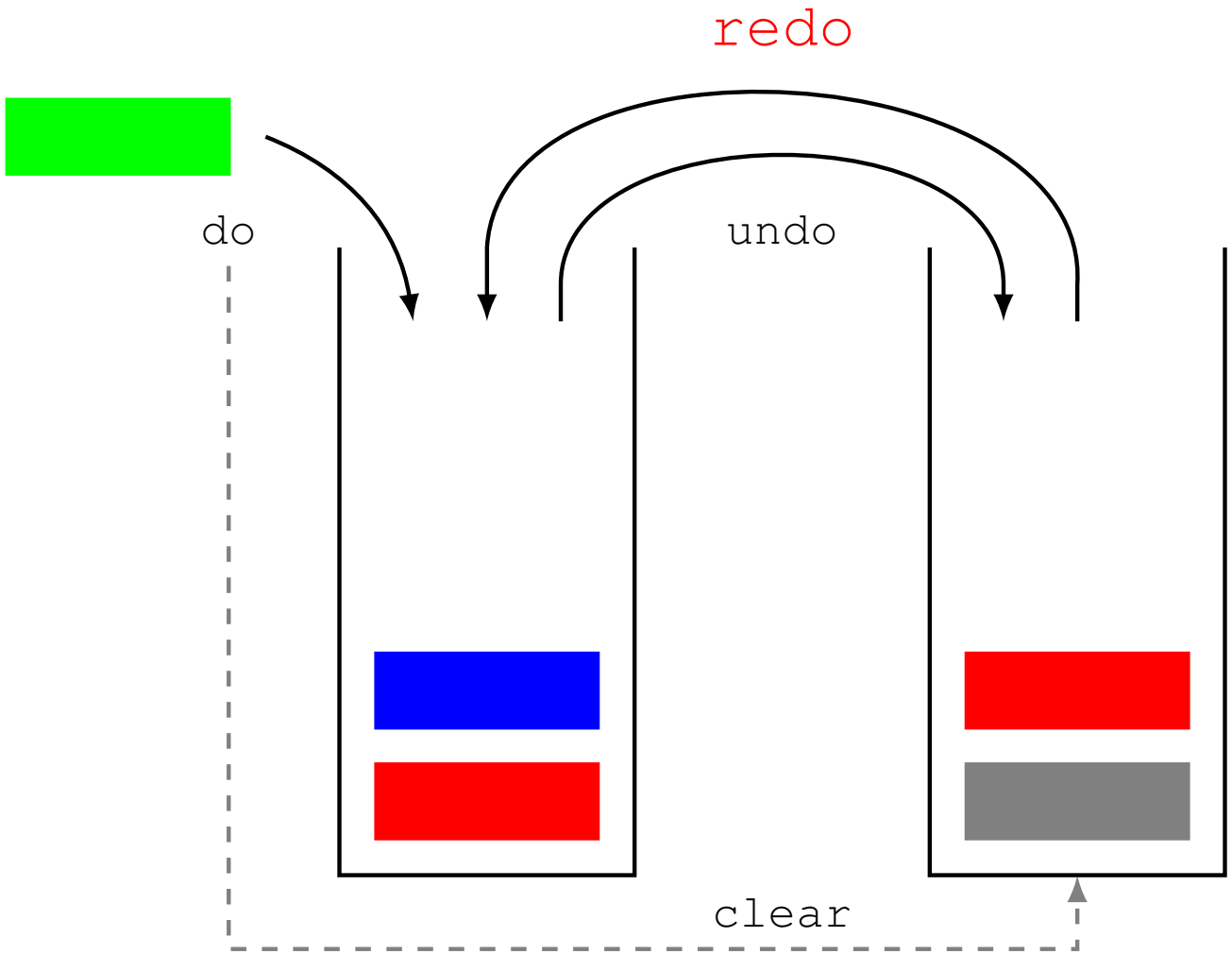

redo
do
undo
clear

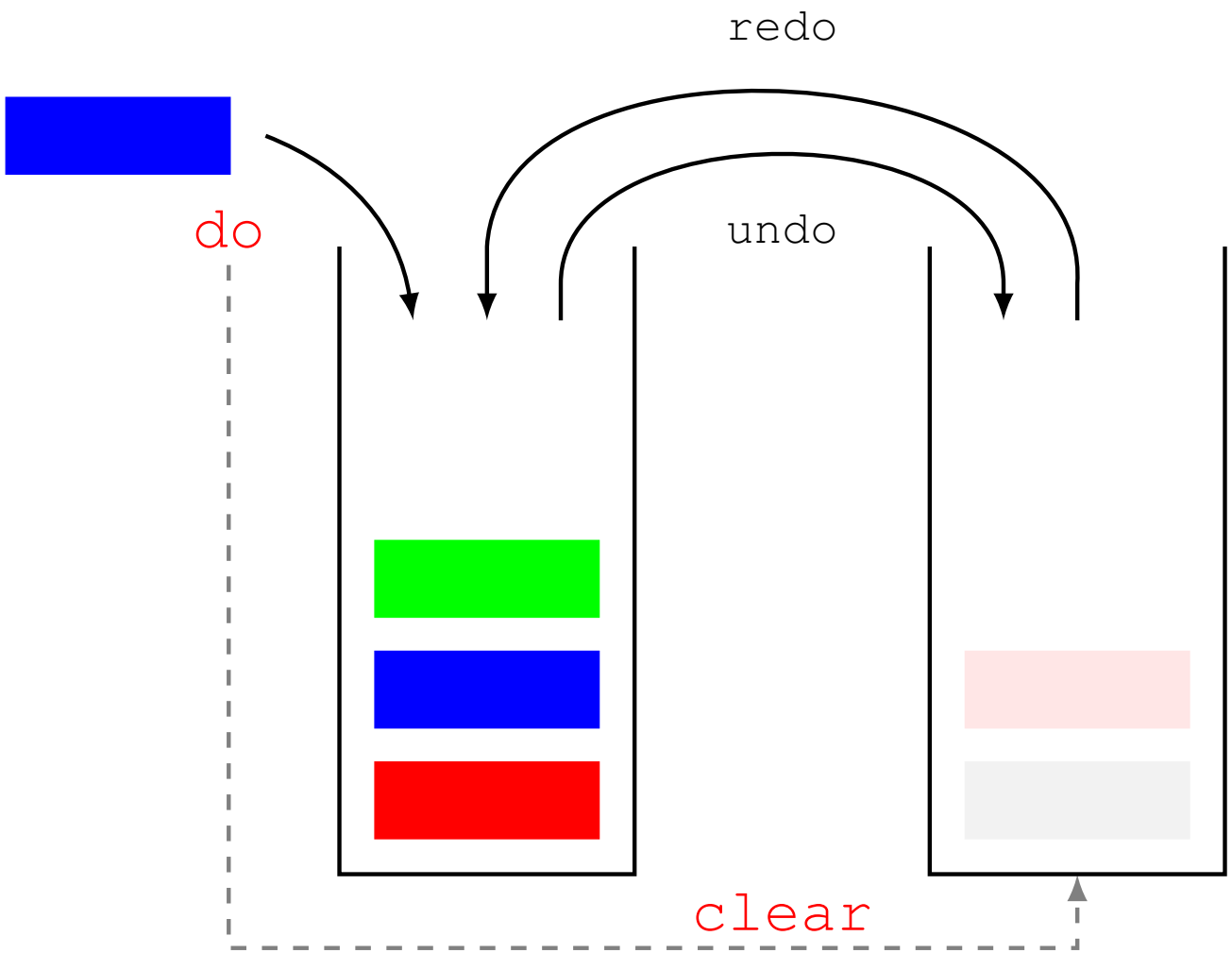

redo
do
undo
clear

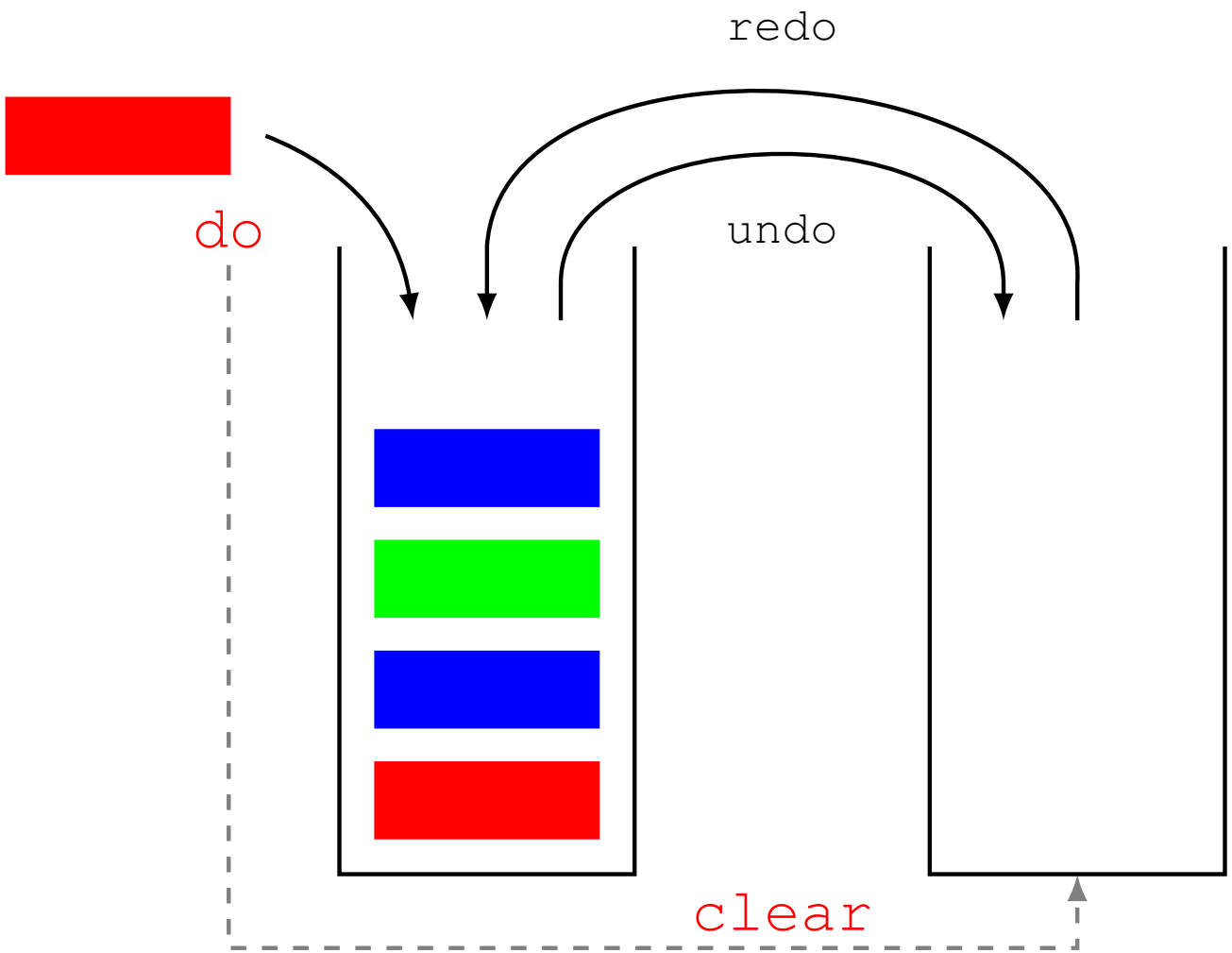

redo
do
undo
clear

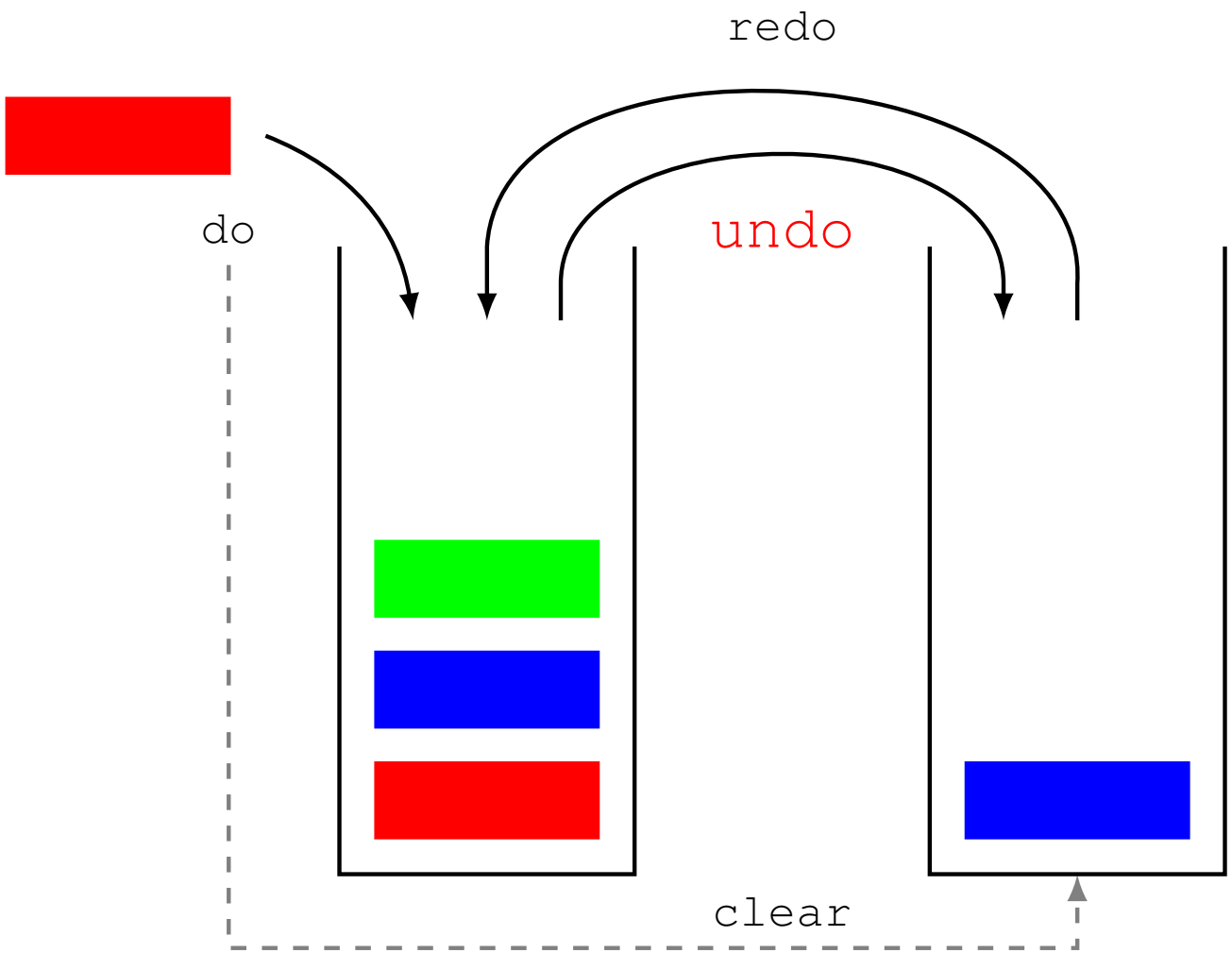

redo
do
undo
clear

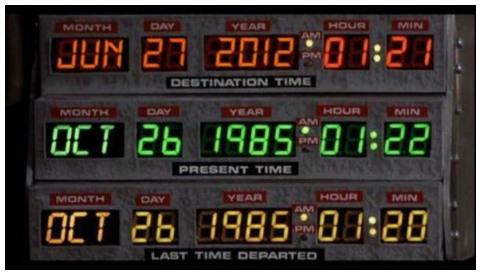

redo
do
undo
clear

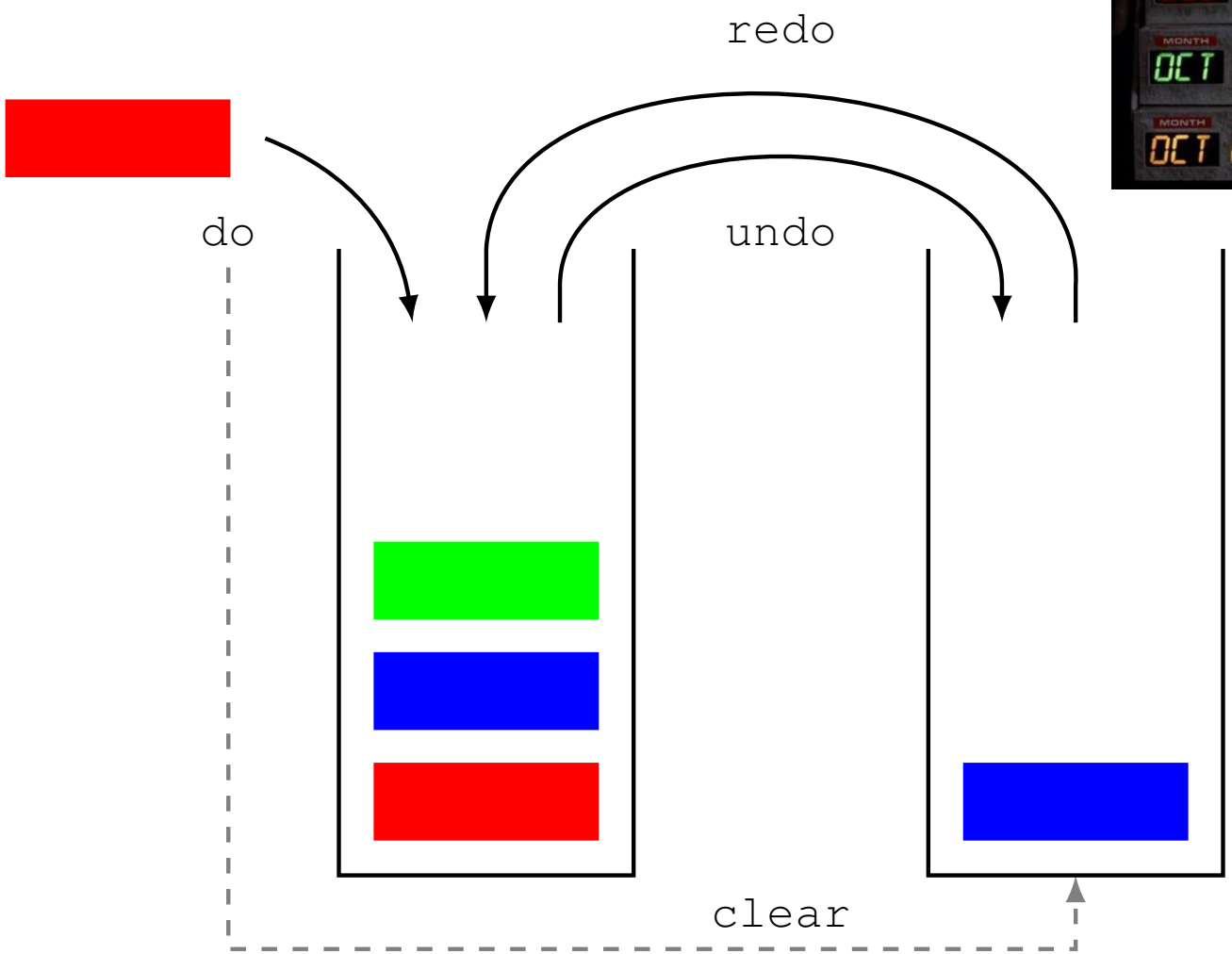

redo
do
undo
clear


redo
do
undo
clear


redo
do
undo
clear


redo
do
undo
clear


MONTH DAY YEAR HOUR MIN
JUN 27 2012 AM PM 01:21
DESTINATION TIME
MONTH DAY YEAR HOUR MIN
OCT 26 1985 AM PM 01:22
PRESENT TIME
MONTH DAY YEAR HOUR MIN
OCT 26 1985 AM PM 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 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

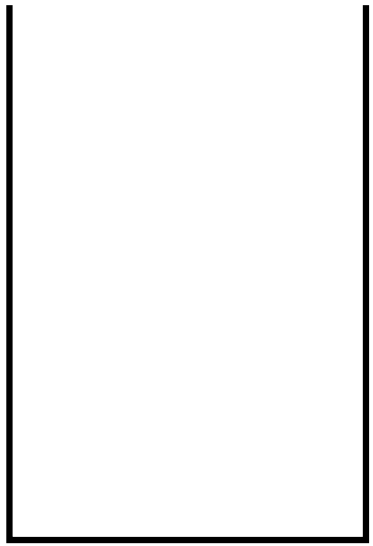
Définition itérative de factorielle (en C)

```
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)

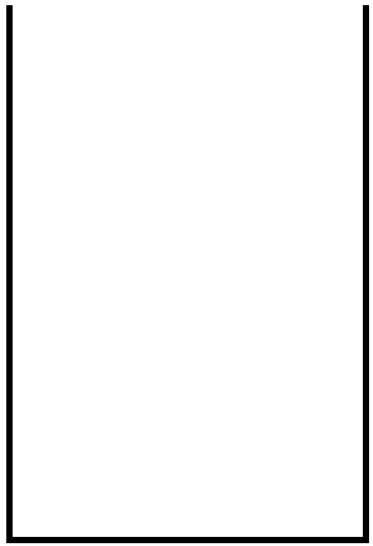


Définition récursive de factorielle (en Caml)

```
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`



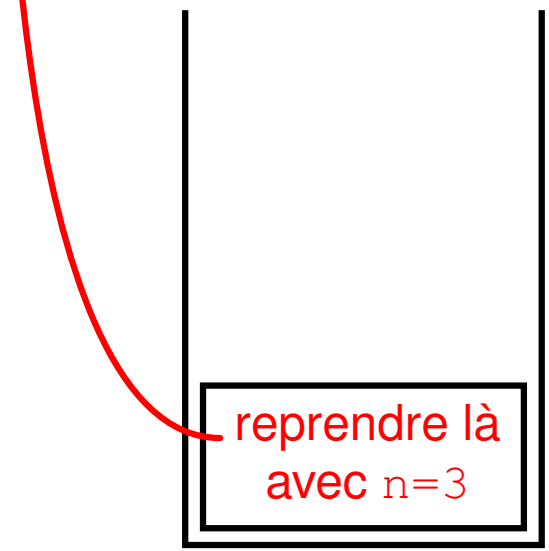
Définition récursive de factorielle (en Caml)

```
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)

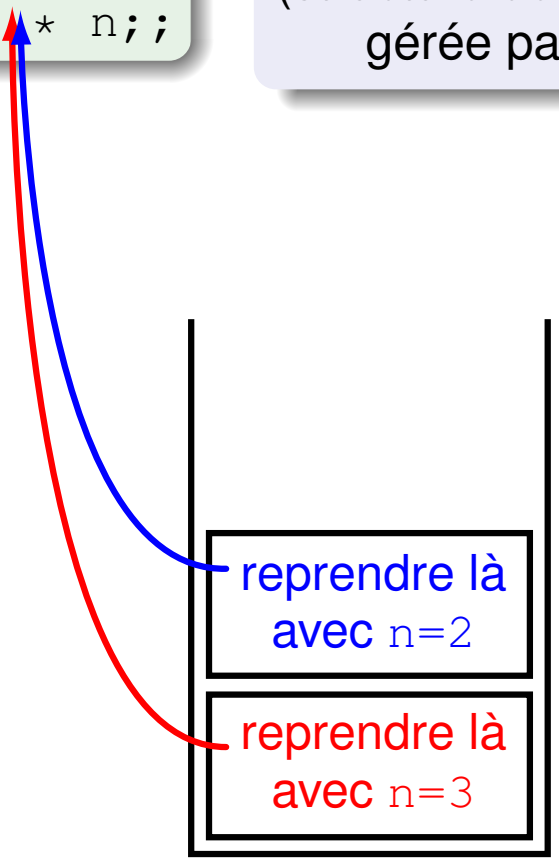
```
let rec fact n =
 if n < 2 then 1 else fact(n - 1) * n;;
```

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;;
```

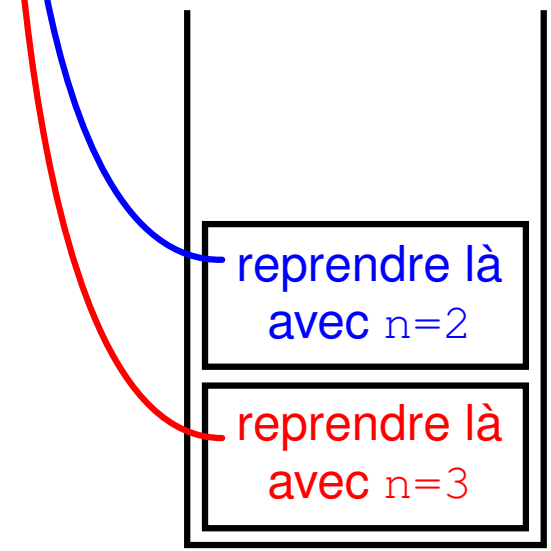
Compilation plus délicate : utilise une pile d'appels (structure de données gérée par le compilateur)

appel de `fact` avec `n=3`

appel de `fact` avec `n=2`

appel de `fact` avec `n=1`

retour avec `résultat=1`

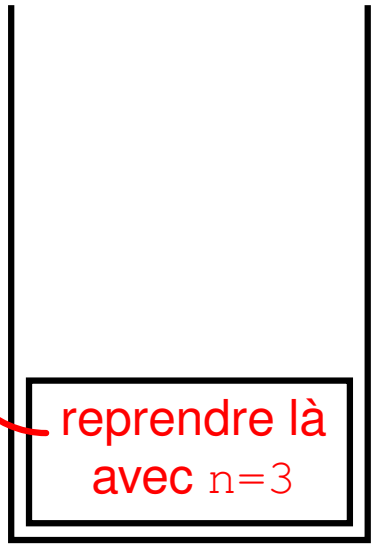


Définition récursive de factorielle (en Caml)

```
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`

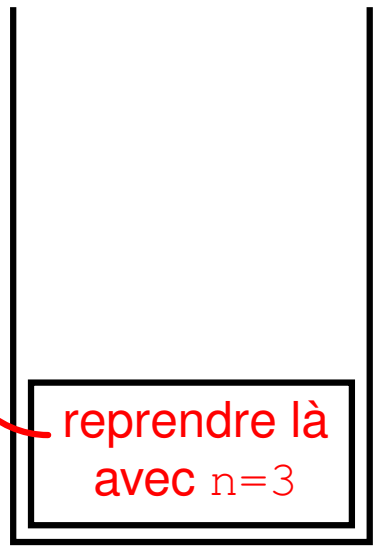


Définition récursive de factorielle (en Caml)

```
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`



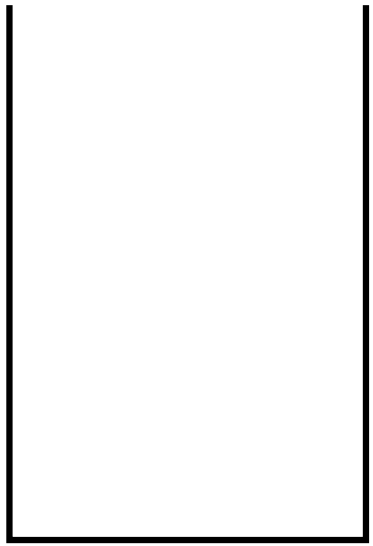
Définition récursive de factorielle (en Caml)

```
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`



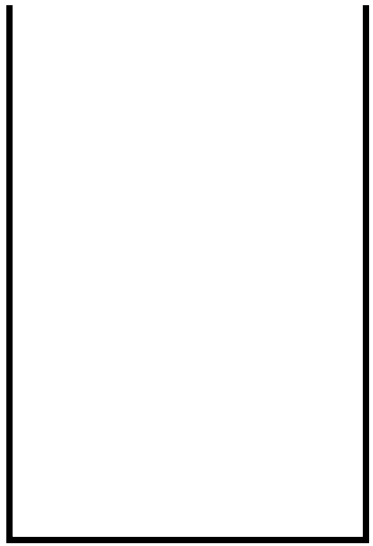
Définition récursive de factorielle (en Caml)

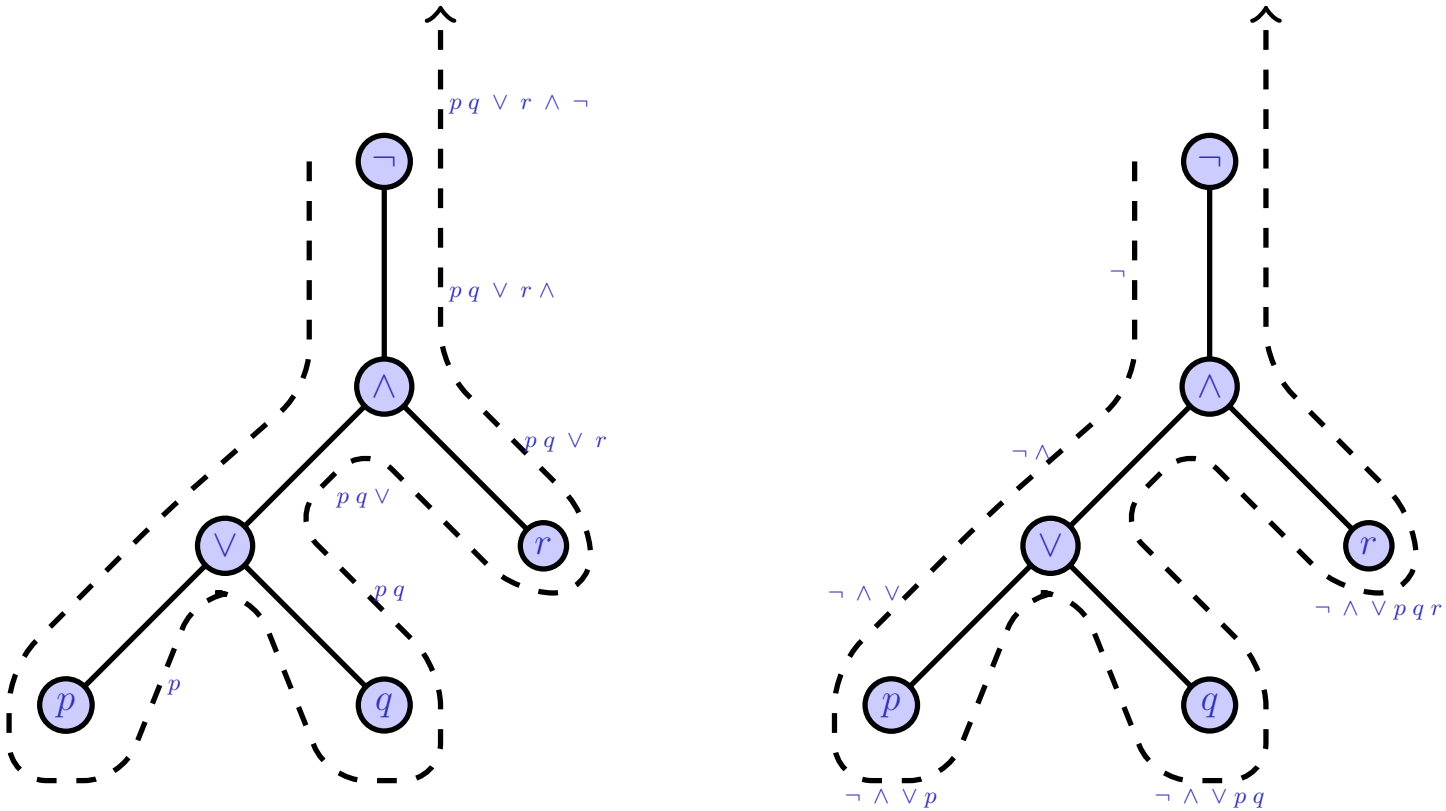
```
let rec fact n =
 if n < 2 then 1 else fact(n - 1) * n;;
```

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

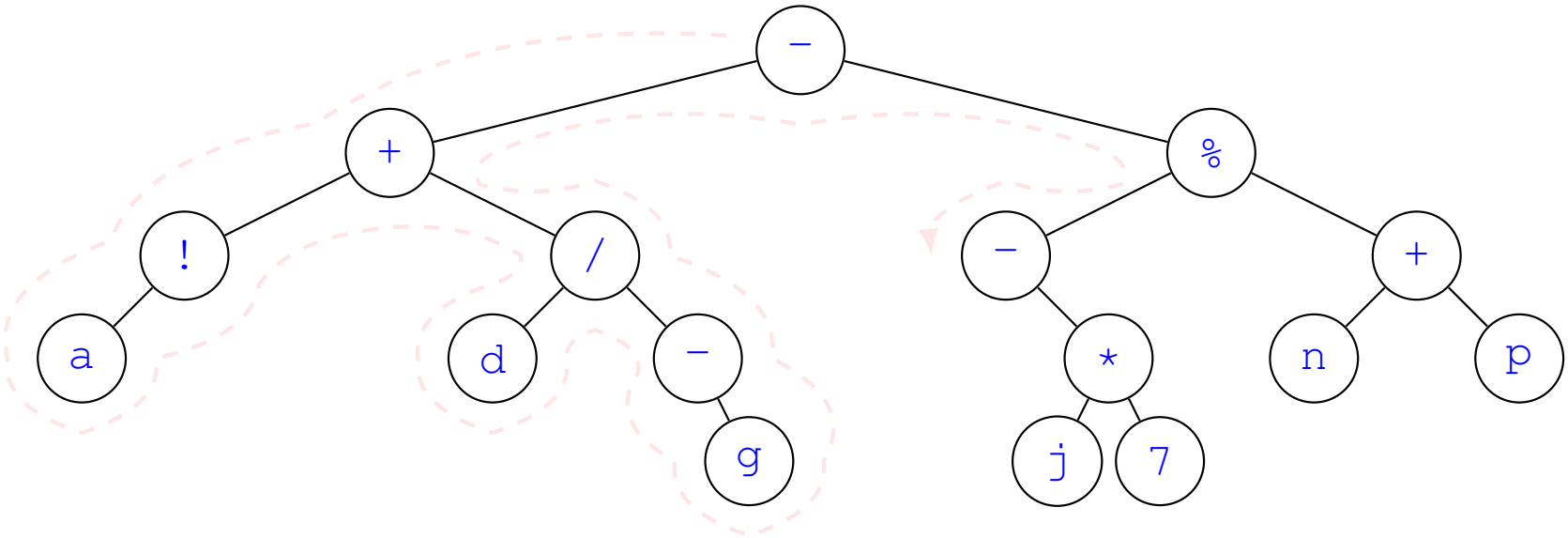
appel de `fact` avec `n=3`
- appel de `fact` avec `n=2`
  - appel de `fact` avec `n=1`
  - retour avec `résultat=1`
- calcul de `1*n` avec `n=2`
- retour avec `résultat=2`

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





(vu en PF1 — amphi 6)

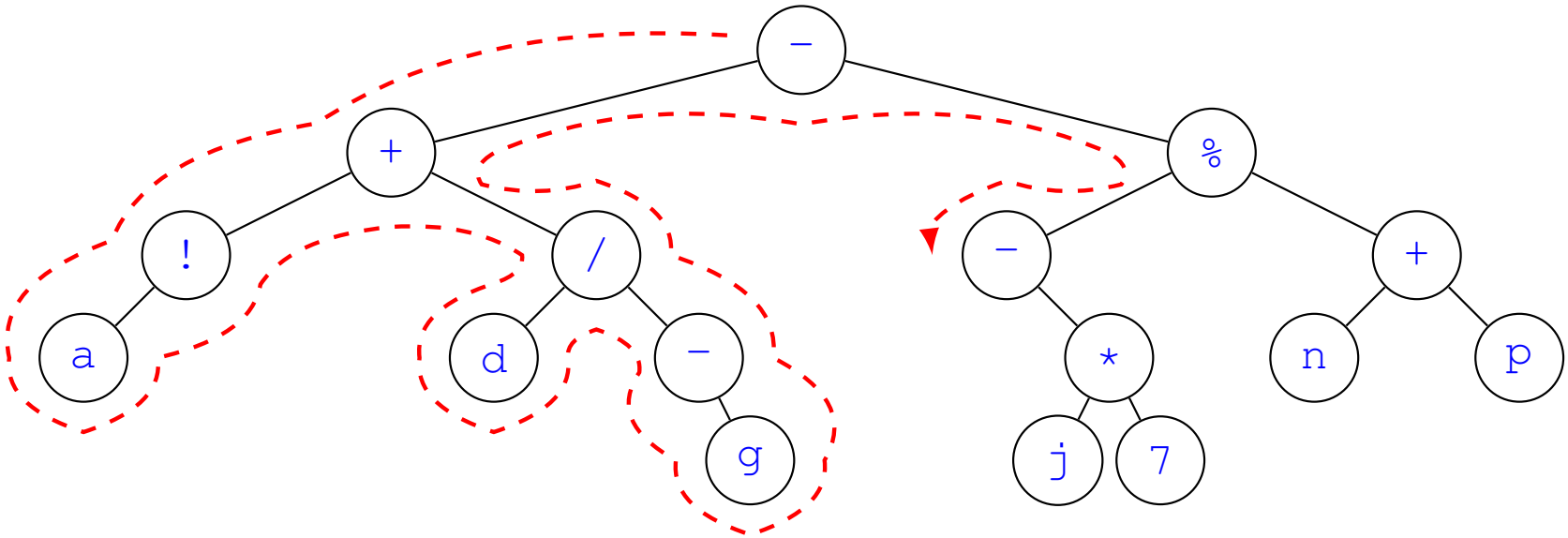


infixe

préfixe

postfixe

évaluation

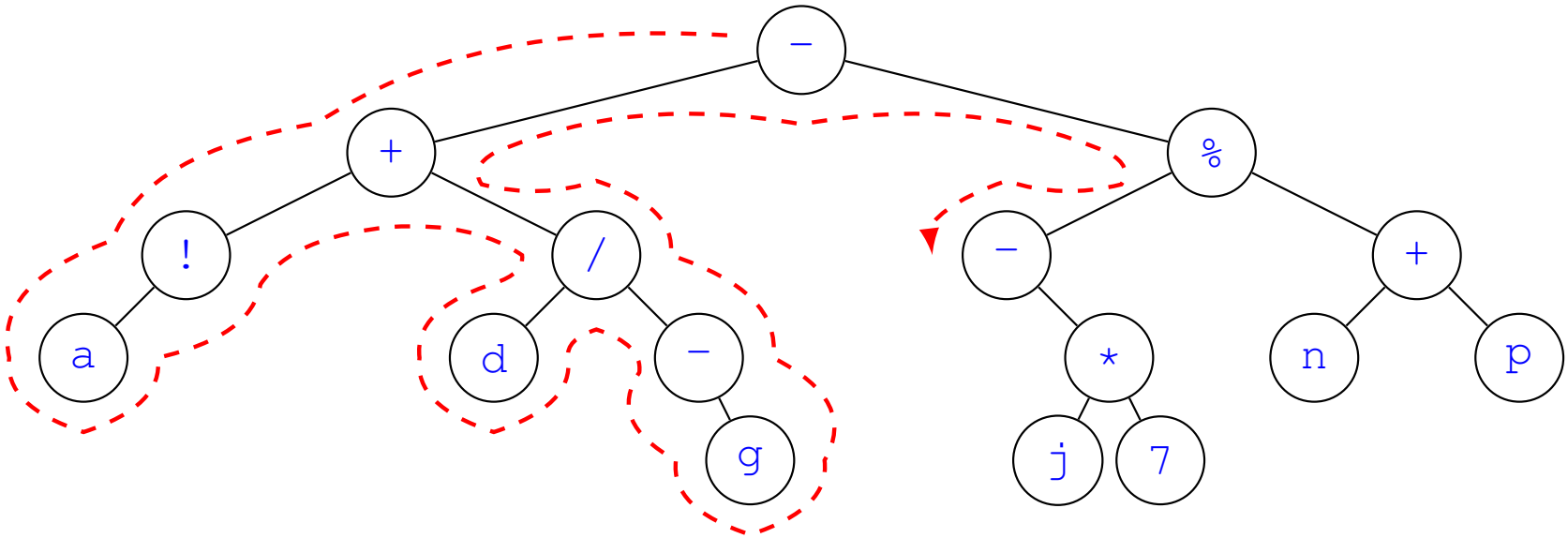


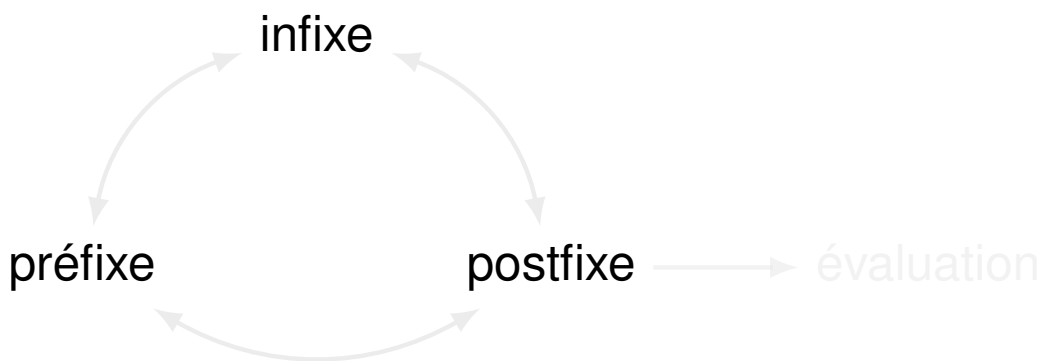
infixe

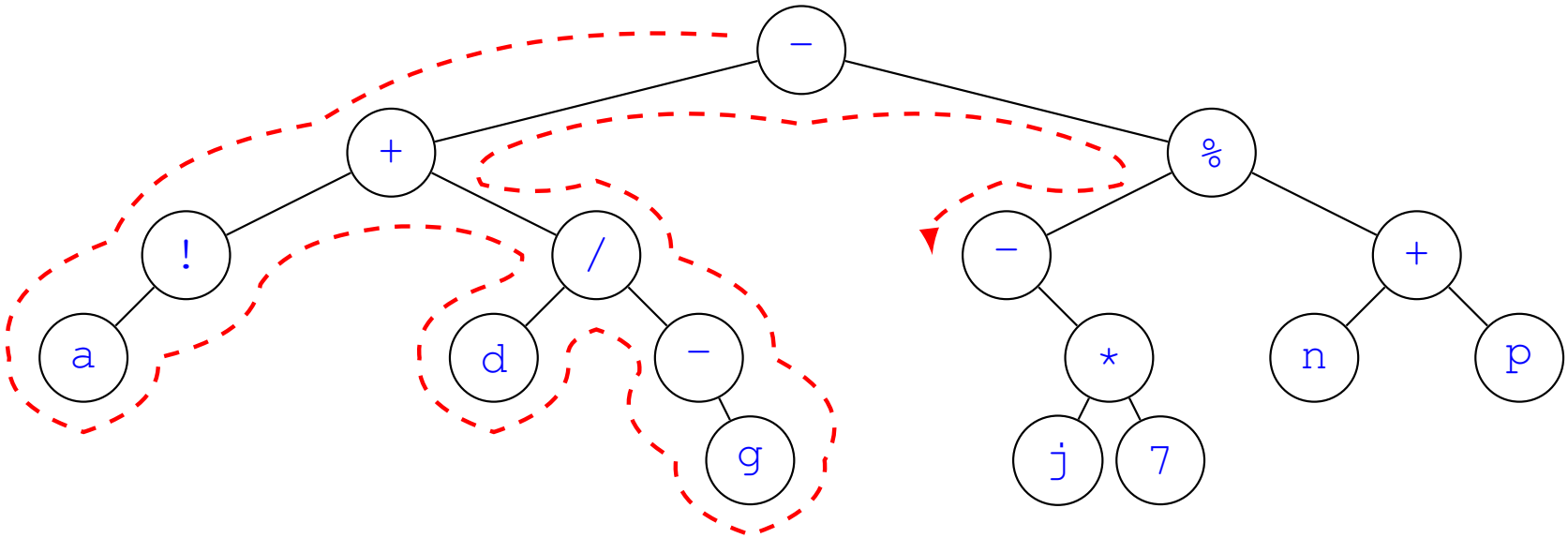
préfixe

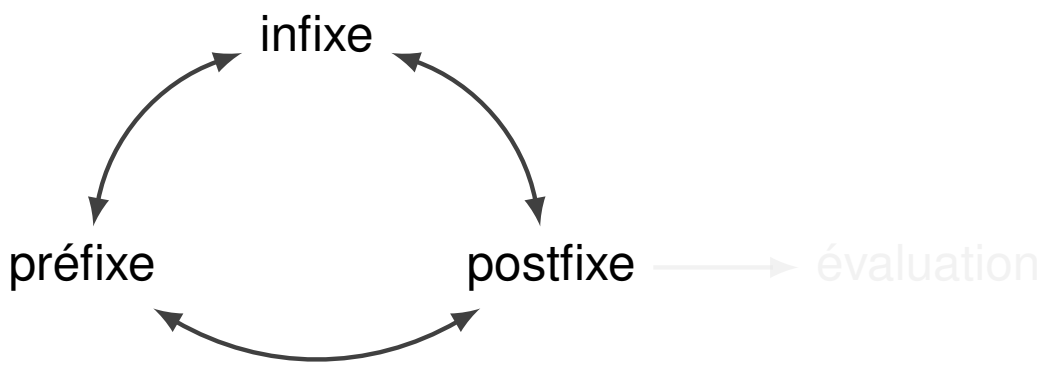
postfixe

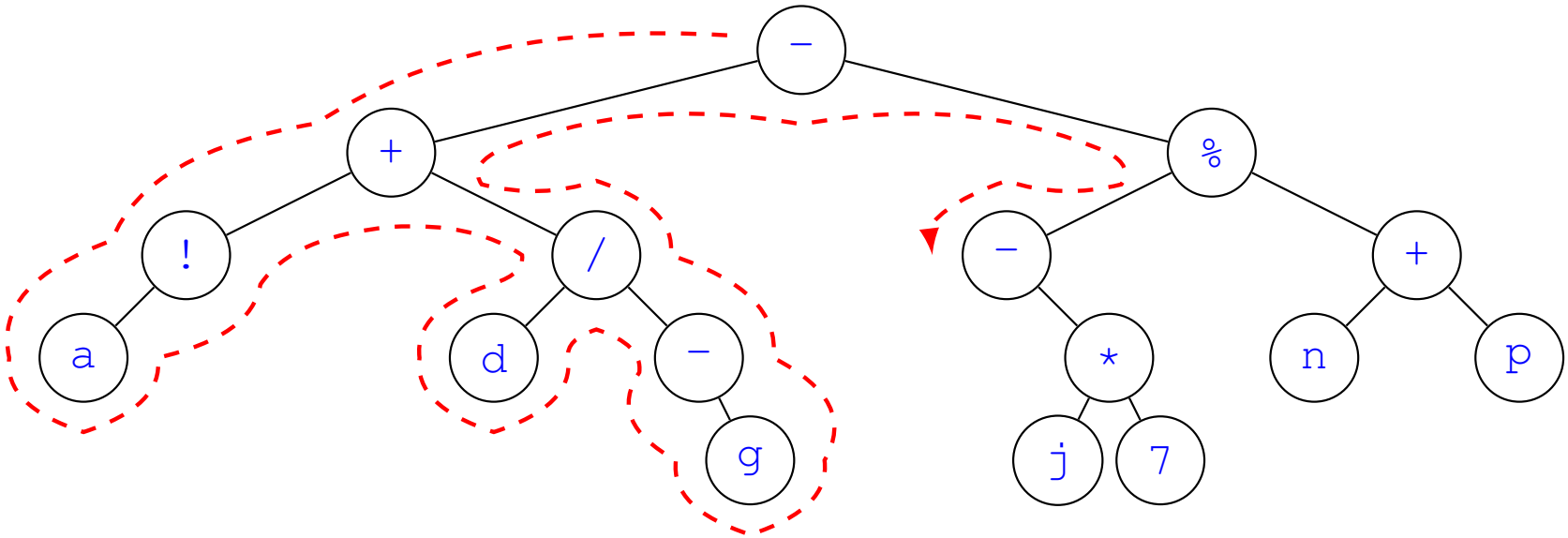
évaluation

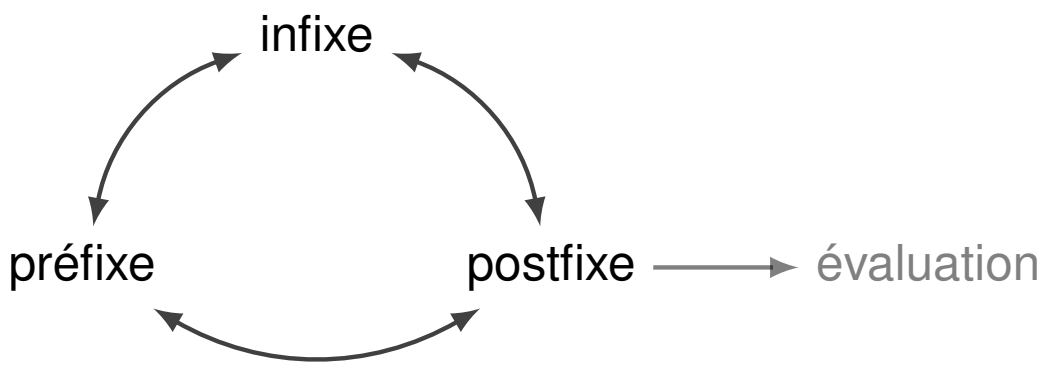

-
+
%
!
/
-
+
a
d
-
*
n
p
g
j
7


infixe
préfixe
postfixe
évaluation


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


infixe
préfixe
postfixe
évaluation


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


infixe
préfixe
postfixe
évaluation