Concepts Informatiques

2017–2018

Matthieu Picantin

Tests et examens

- CC : résultat des 3 tests (ou plus) effectués en TD
- E0 : partiel **mercredi 7 mars**
- E1 : examen deuxième quinzaine de mai
- E2 : examen deuxième quinzaine de 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 **mercredi 7 mars**
- E1 : examen deuxième quinzaine de mai
- E2 : examen deuxième quinzaine de 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 **mercredi 7 mars**
- E1 : examen deuxième quinzaine de mai
- E2 : examen deuxième quinzaine de 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 **mercredi 7 mars**
- E1 : examen deuxième quinzaine de mai
- E2 : examen deuxième quinzaine de juin

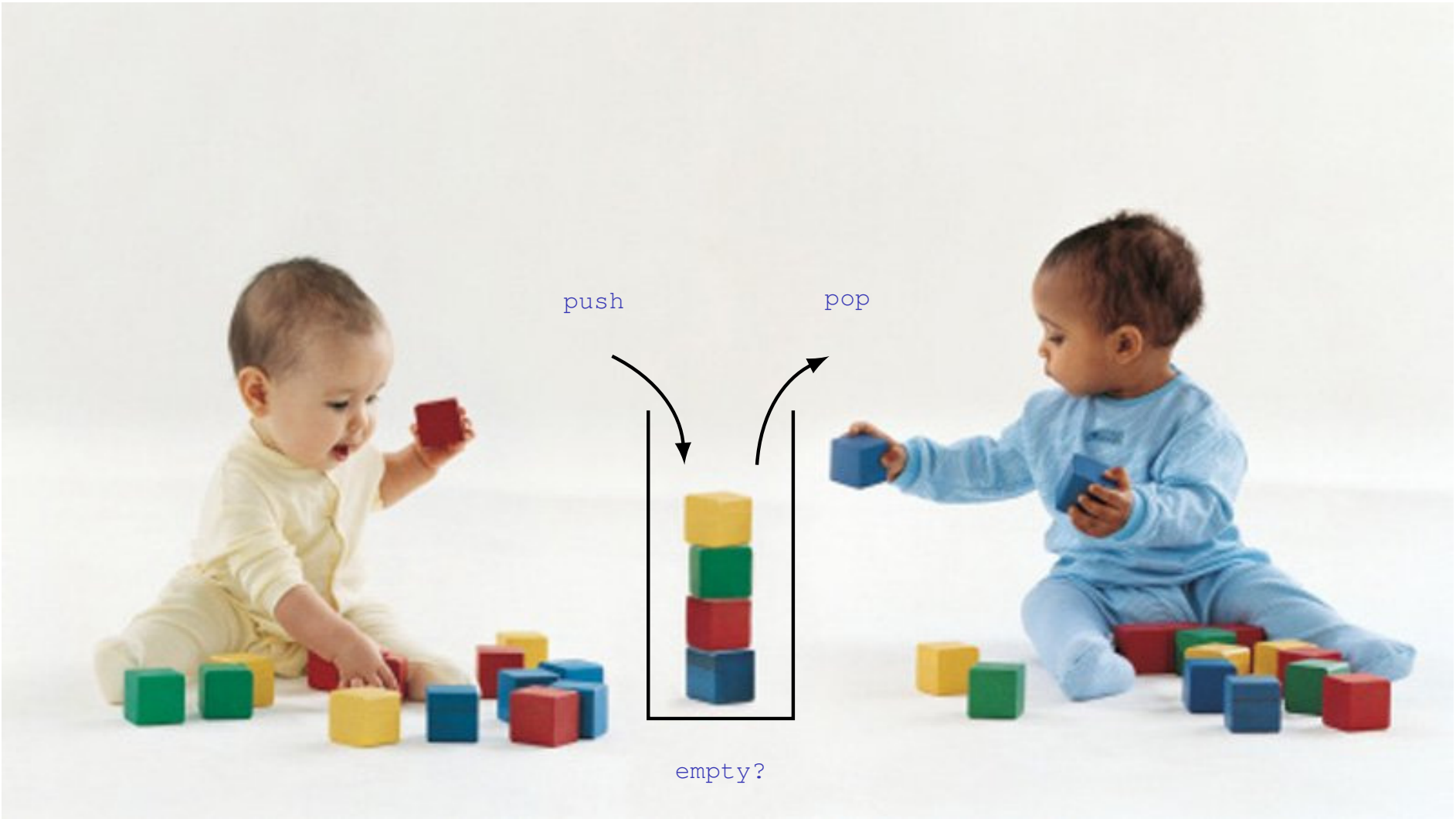
Notes finales

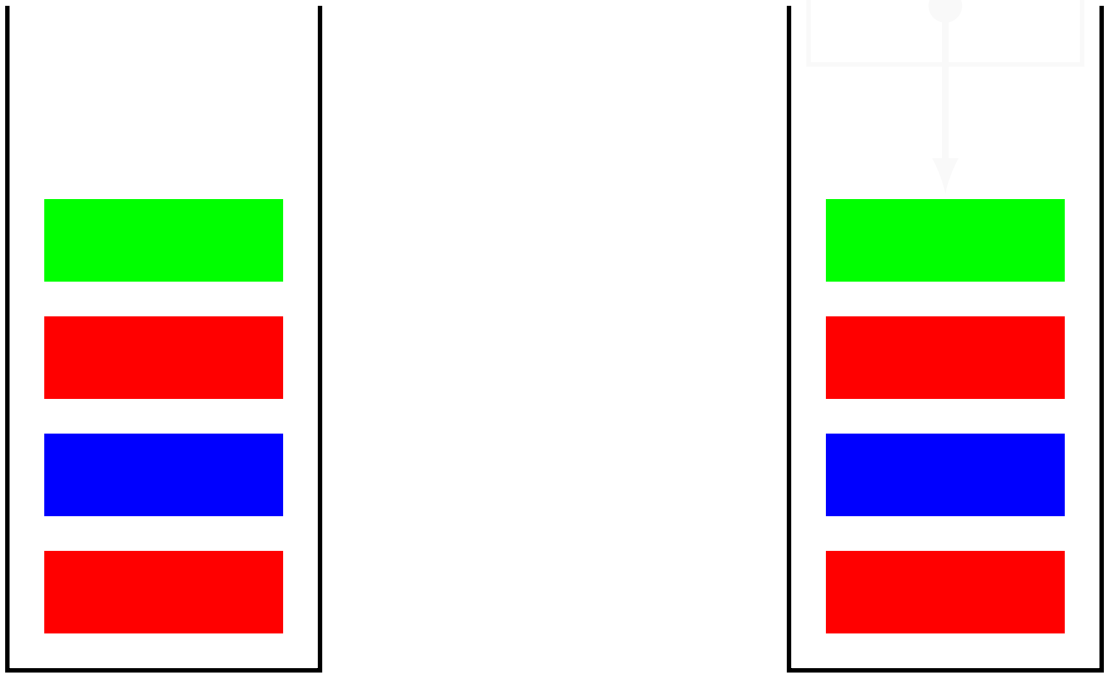
- Note session 1 : 20% CC + 20% E0 + 60% E1
- Note session 2 : max(E2, 20% CC + 80% E2)

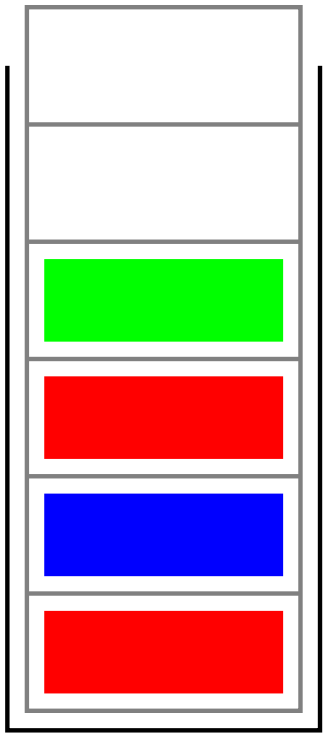
Rappel

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

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

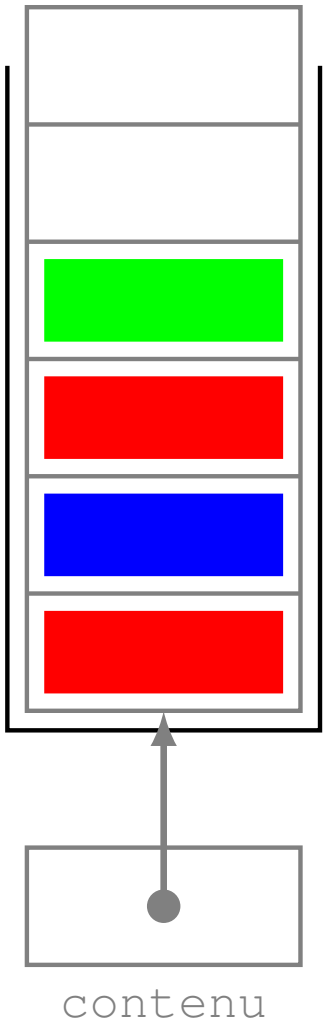

push
pop
empty?


liste
contenu

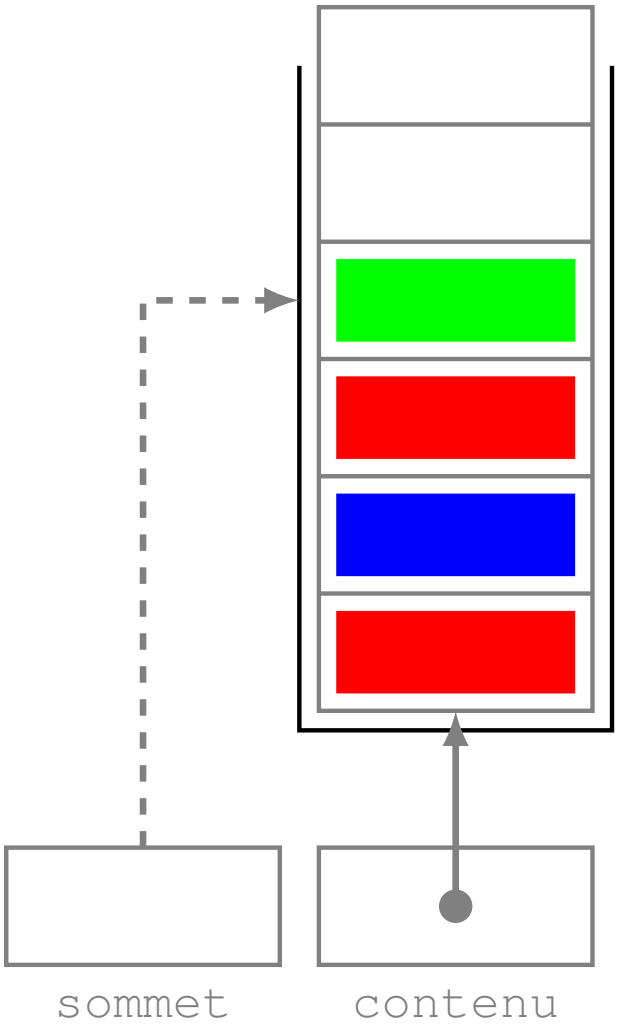


liste

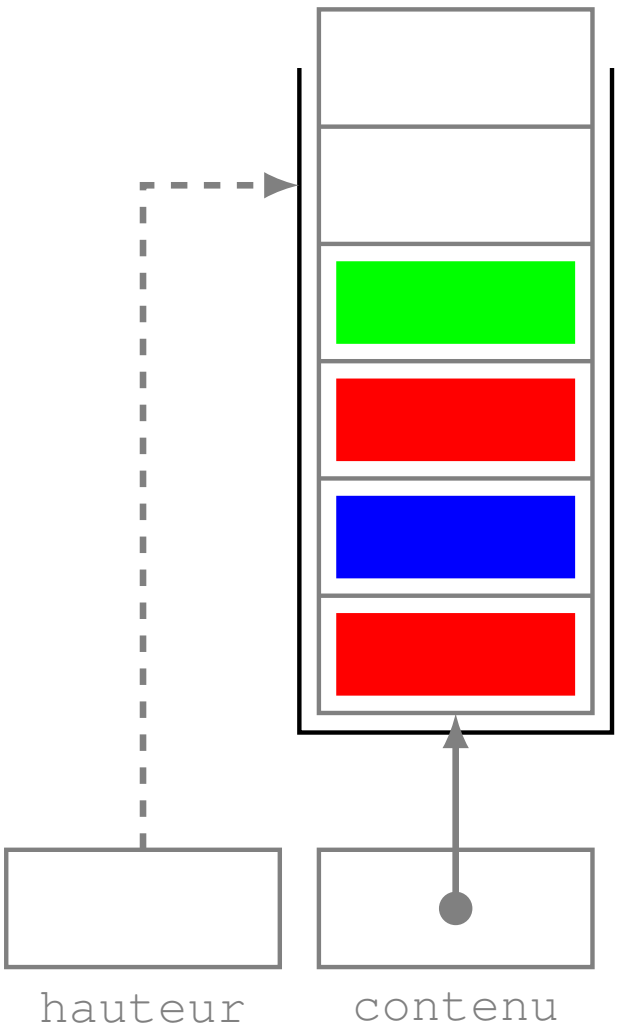
contenu


contenu

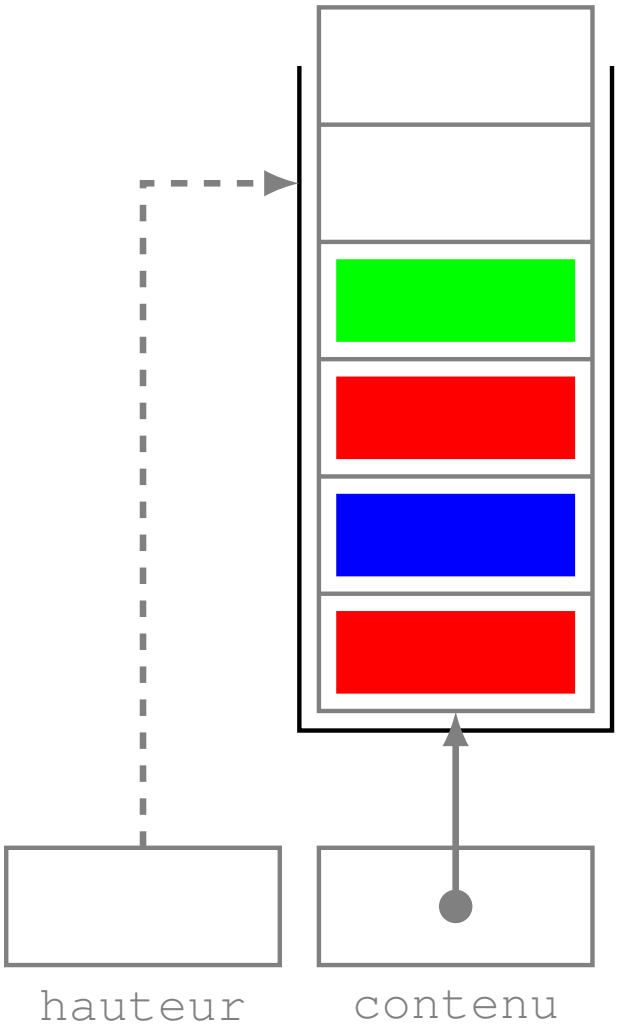
liste

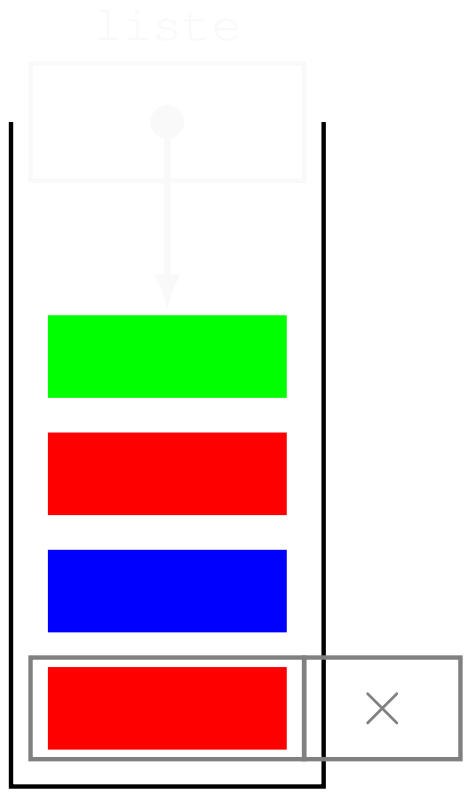

sommet
contenu

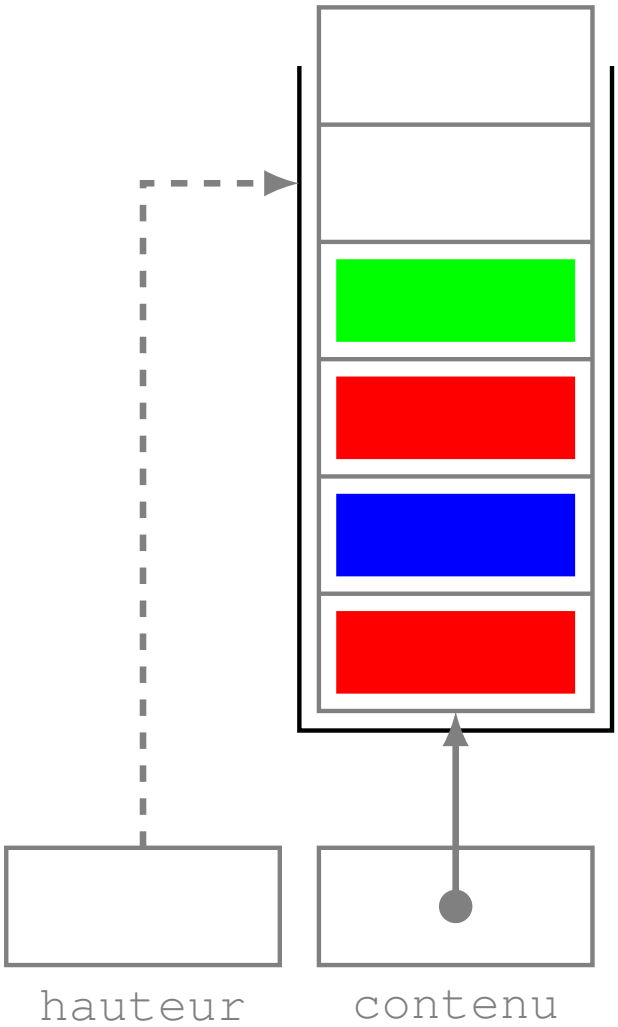
liste

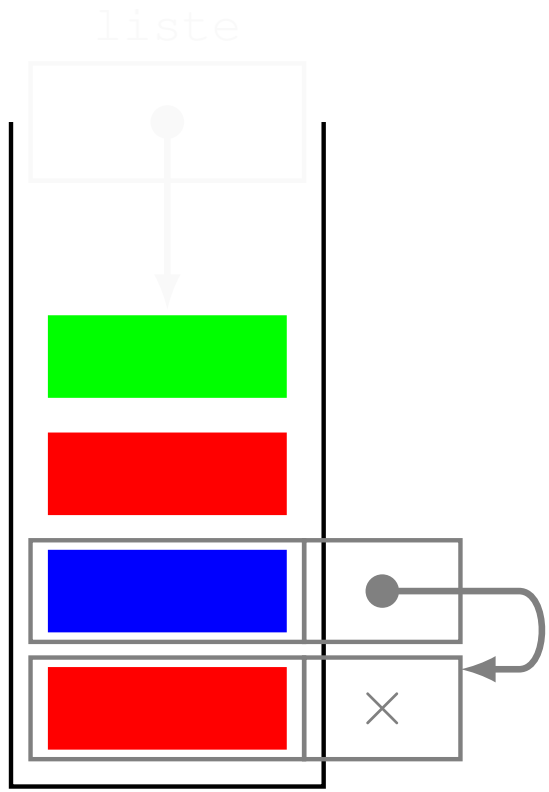

hauteur
contenu

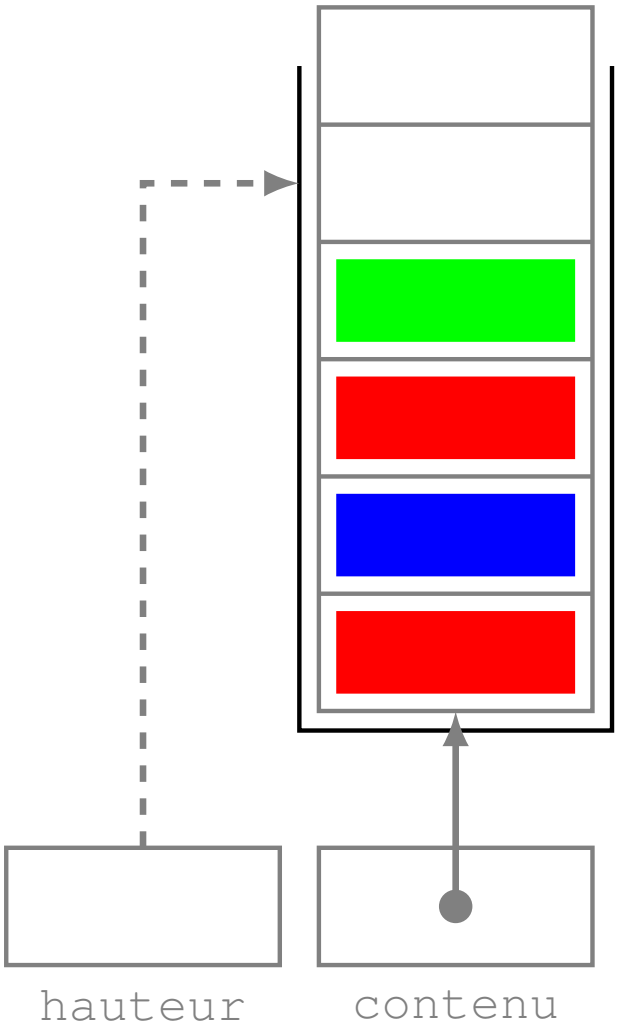
liste

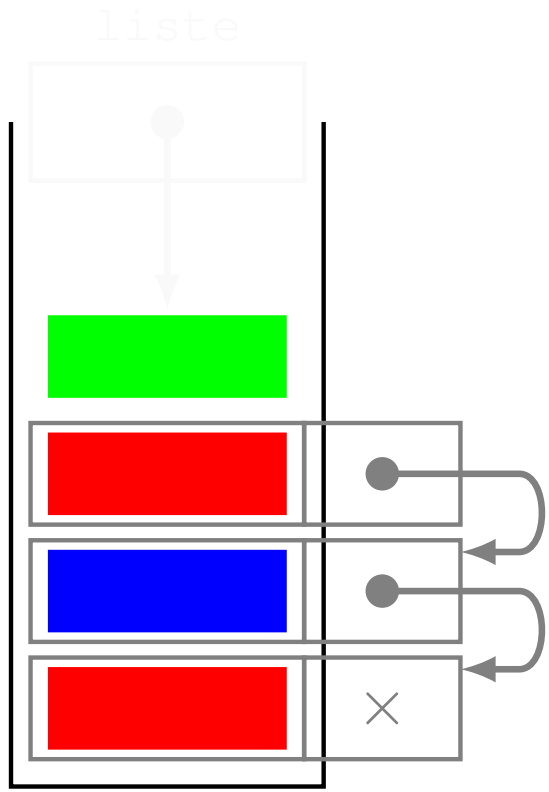

hauteur
contenu

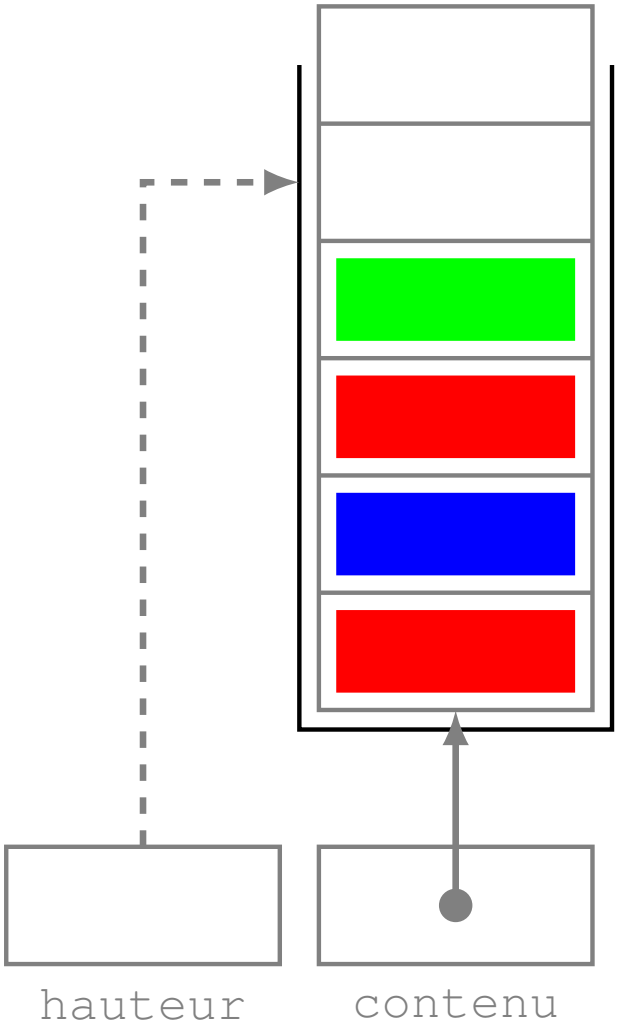

liste

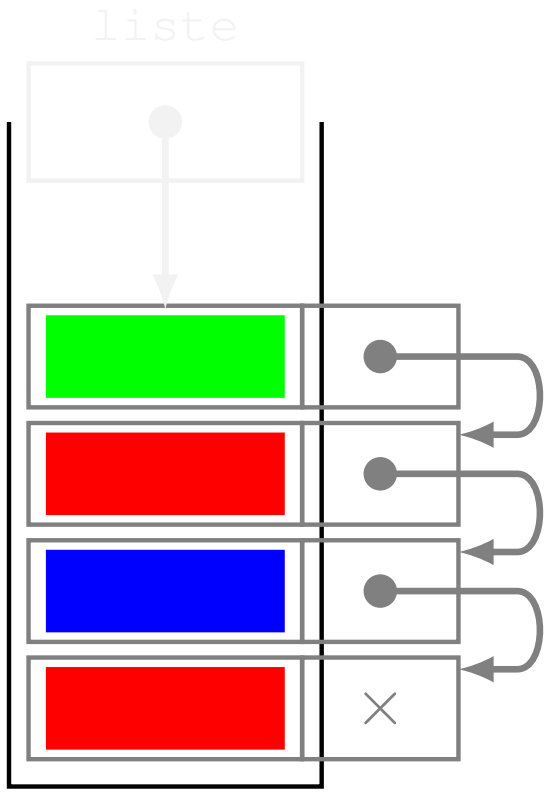

hauteur
contenu

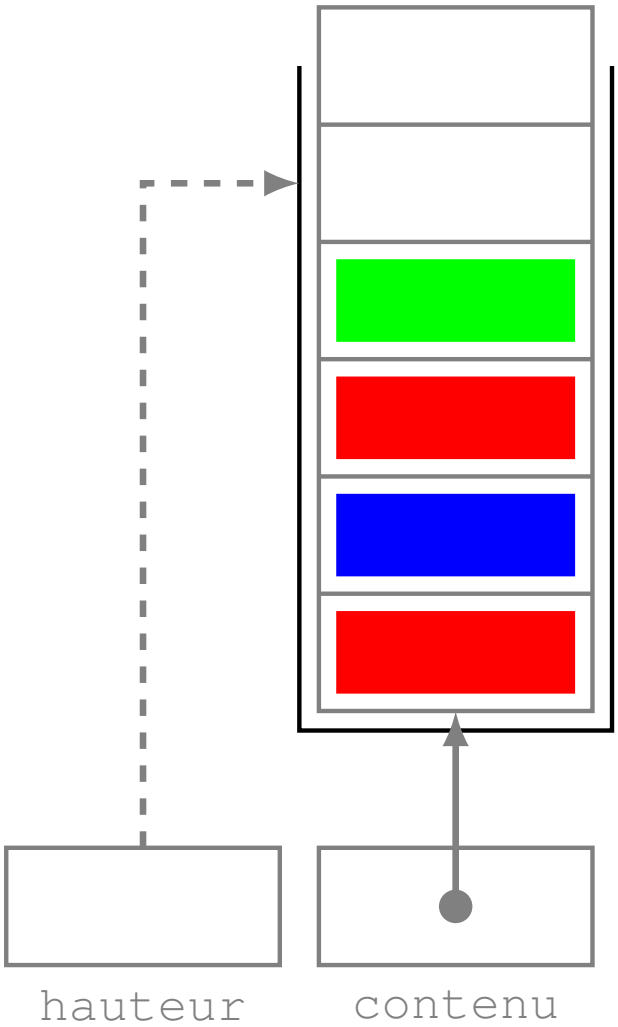

liste

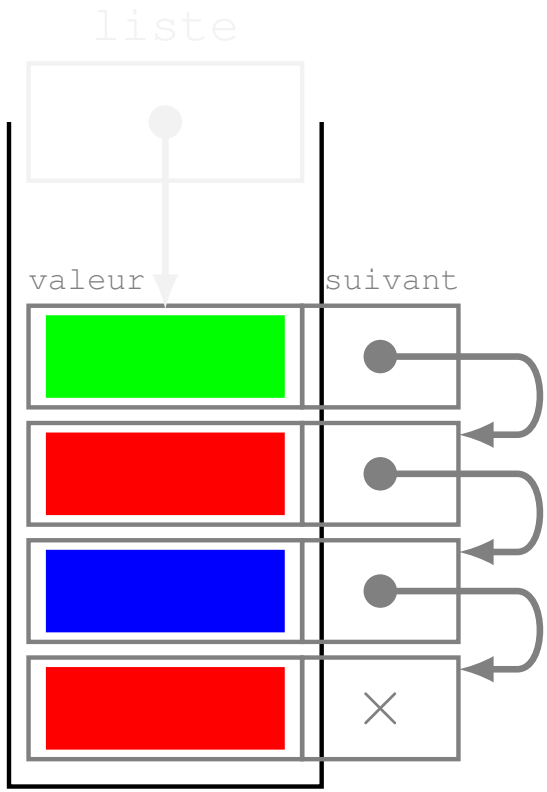

hauteur
contenu

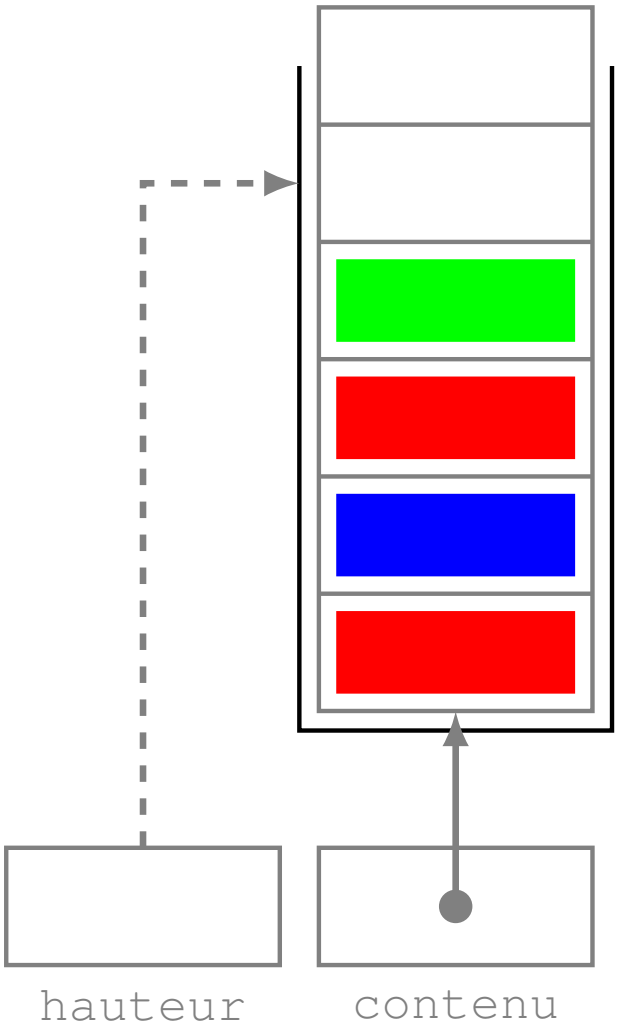

liste

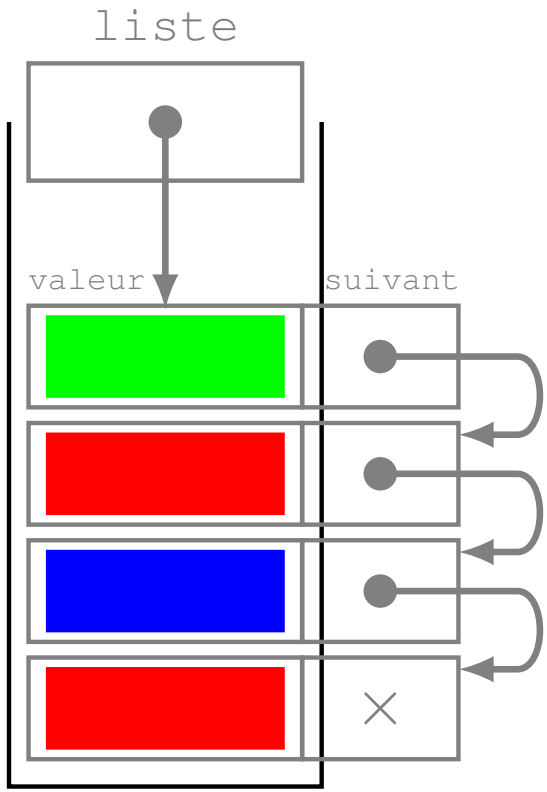

hauteur
contenu

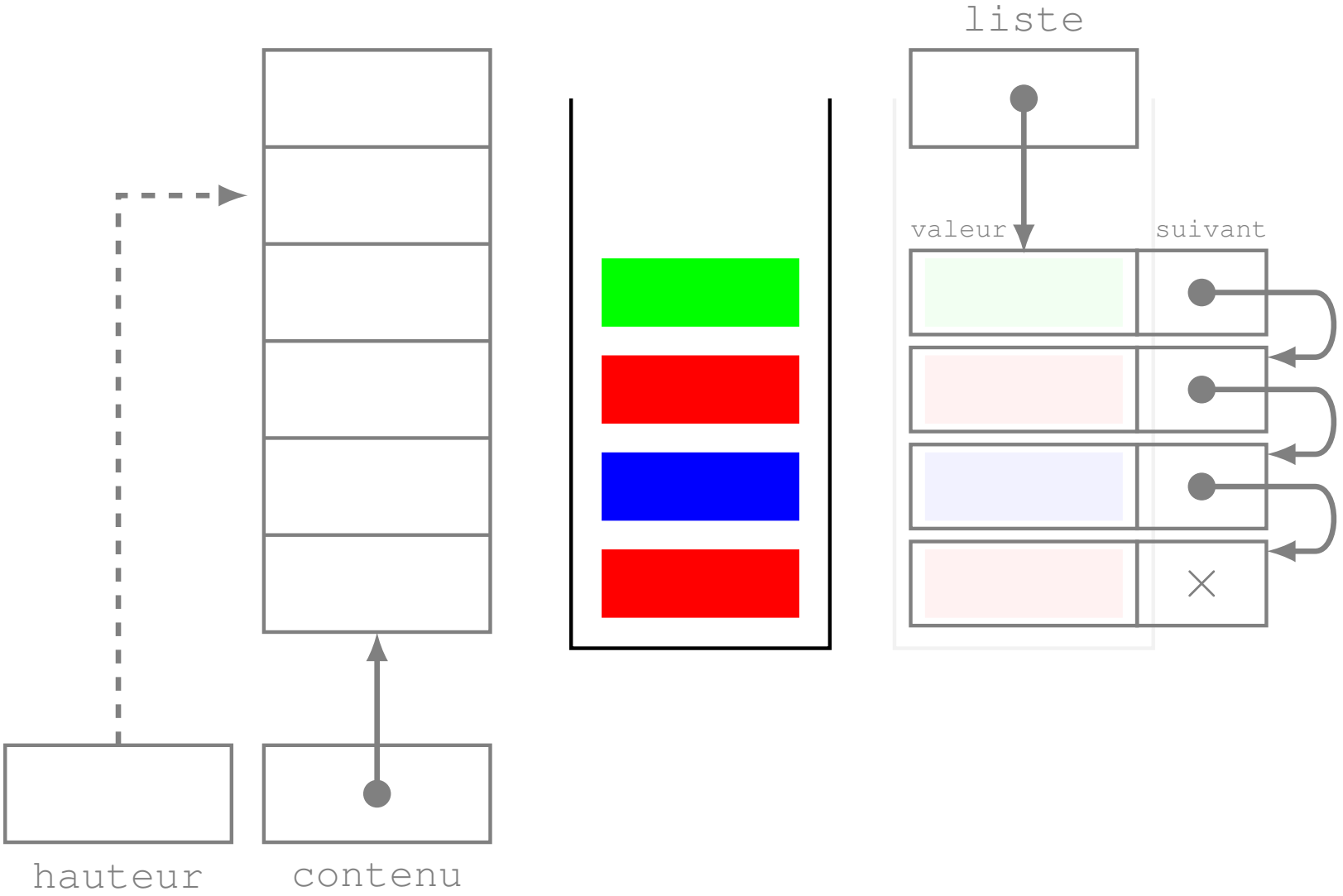

liste

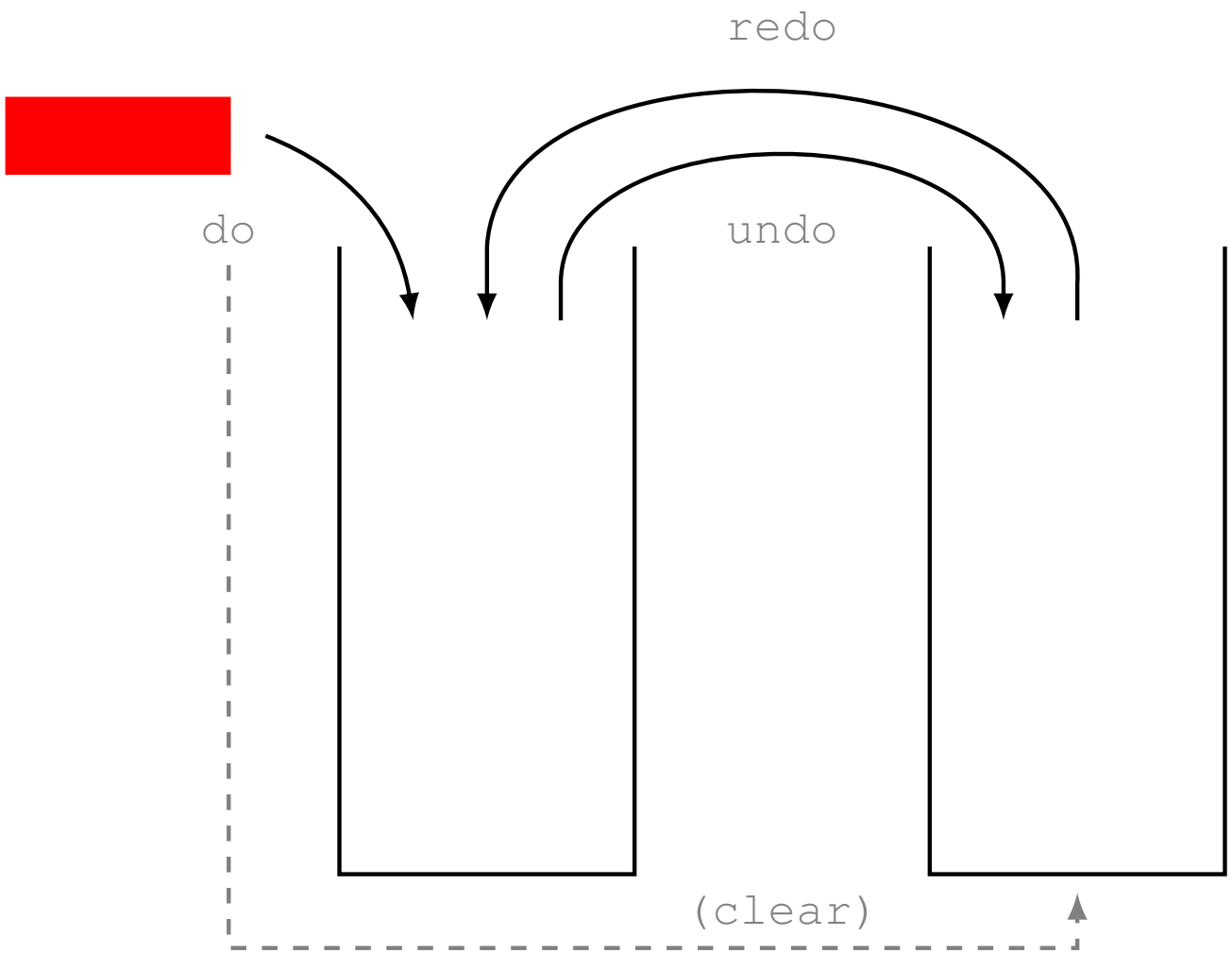

hauteur
contenu

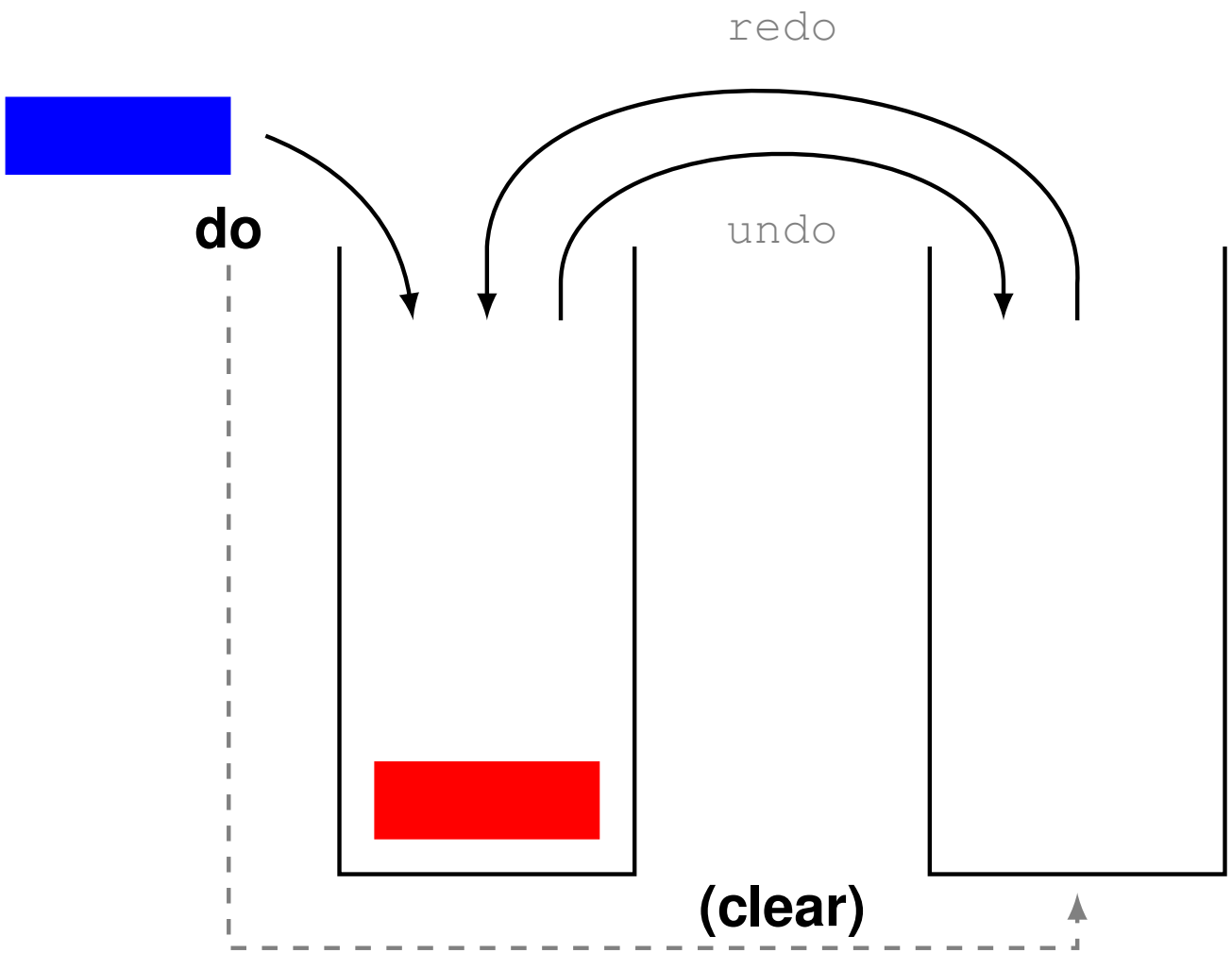

liste
valeur
suivant

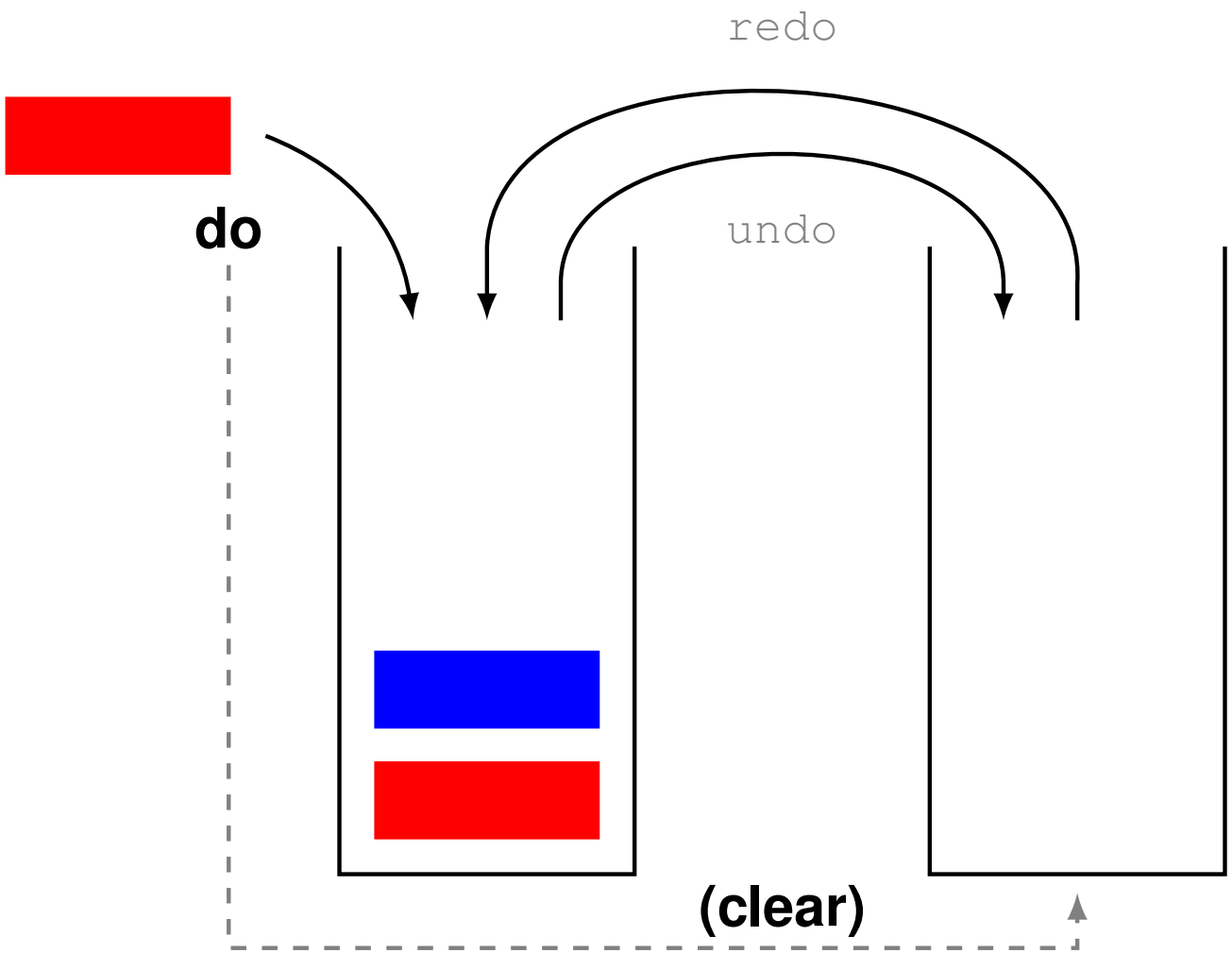

hauteur
contenu

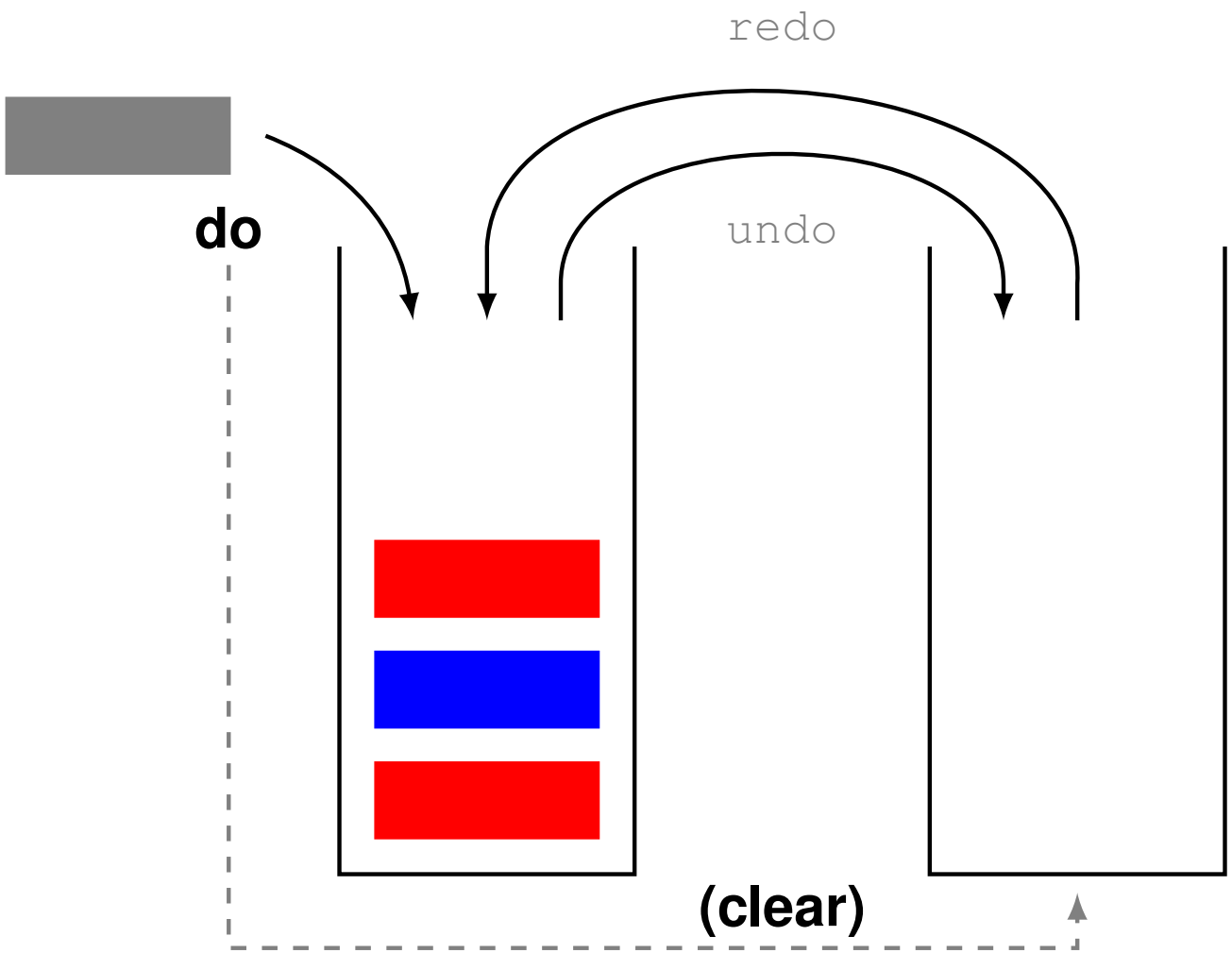

liste
valeur
suivant

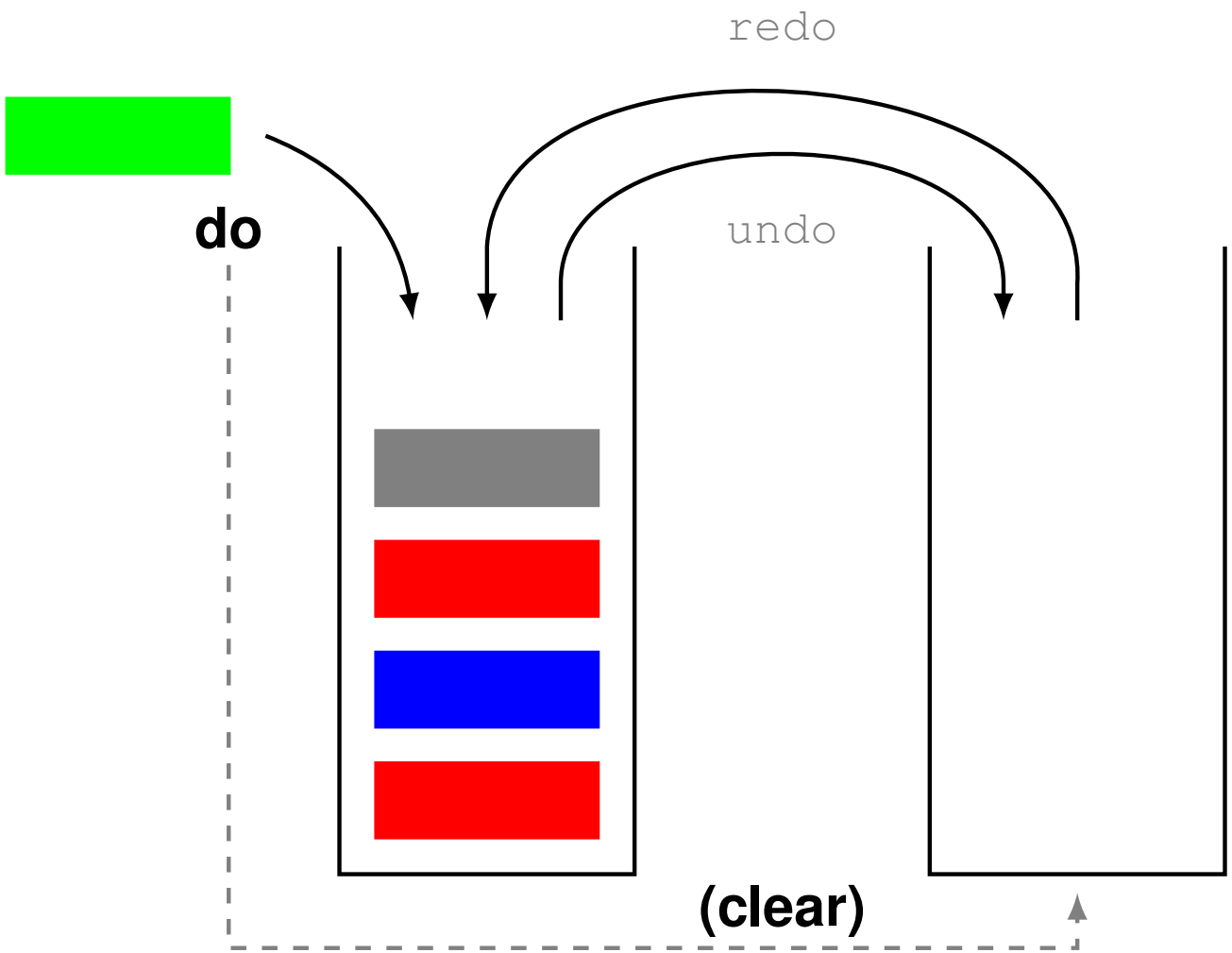

liste
valeur
suivant
hauteur
contenu

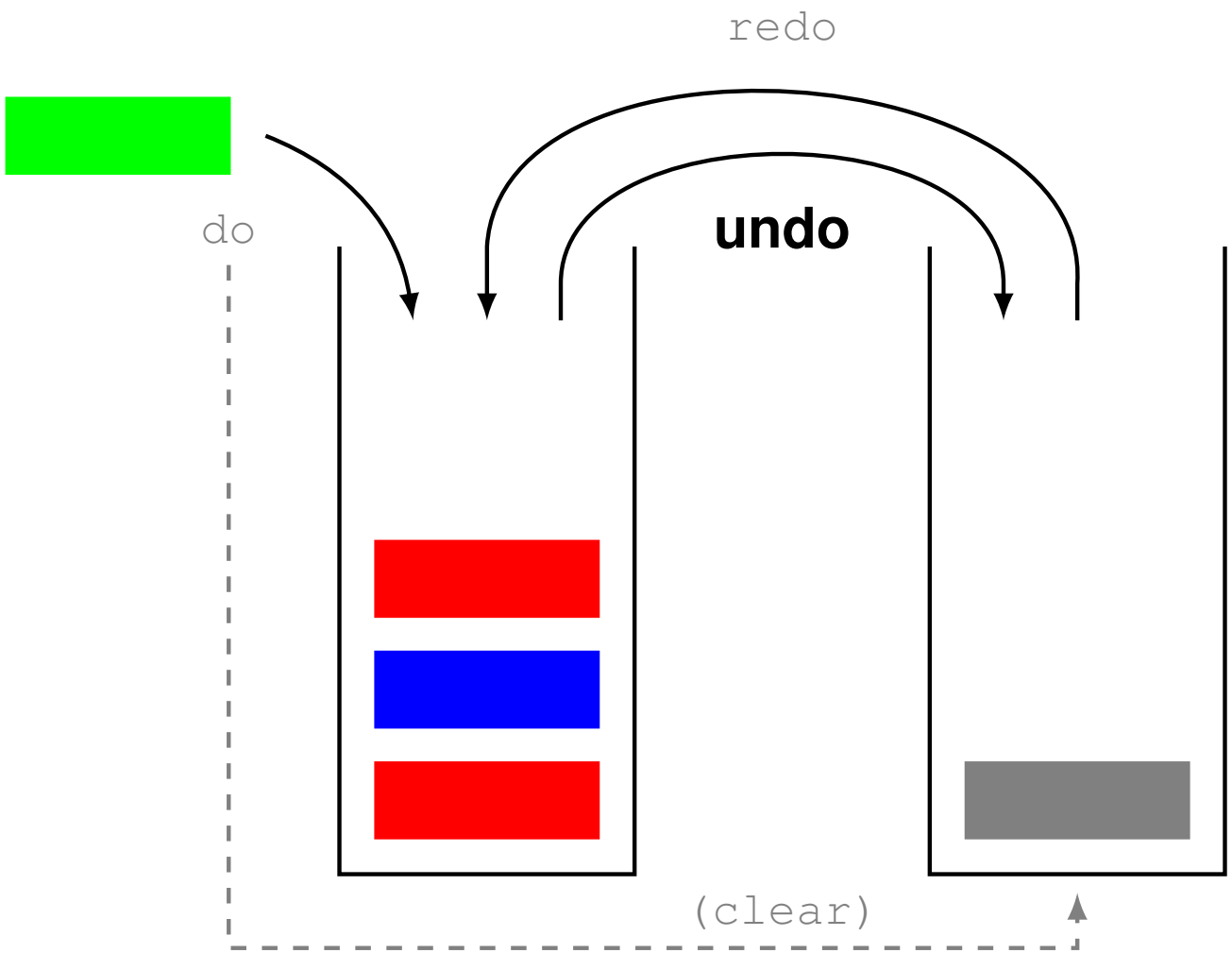

redo
do
undo
(clear)

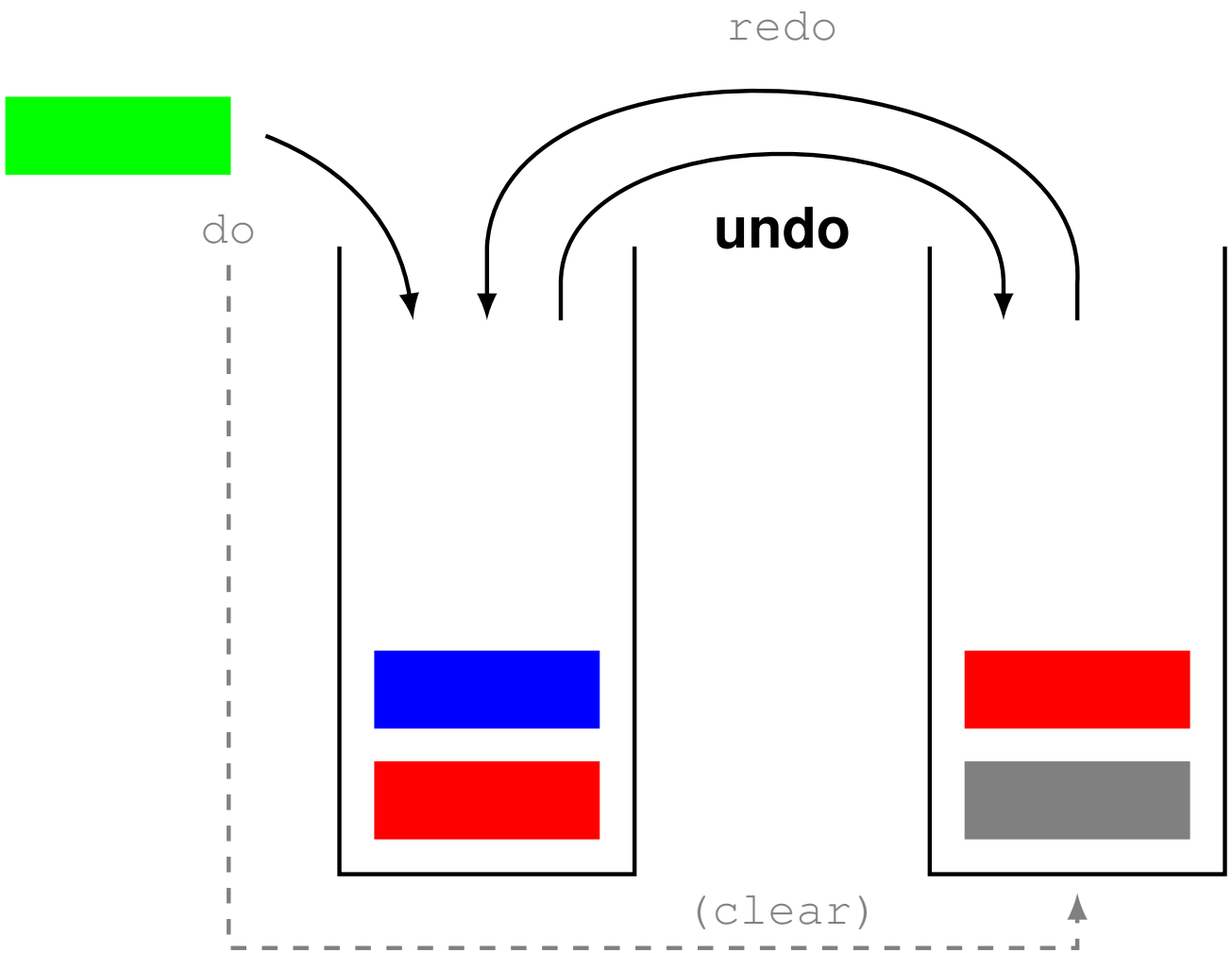

redo
do
undo
(clear)

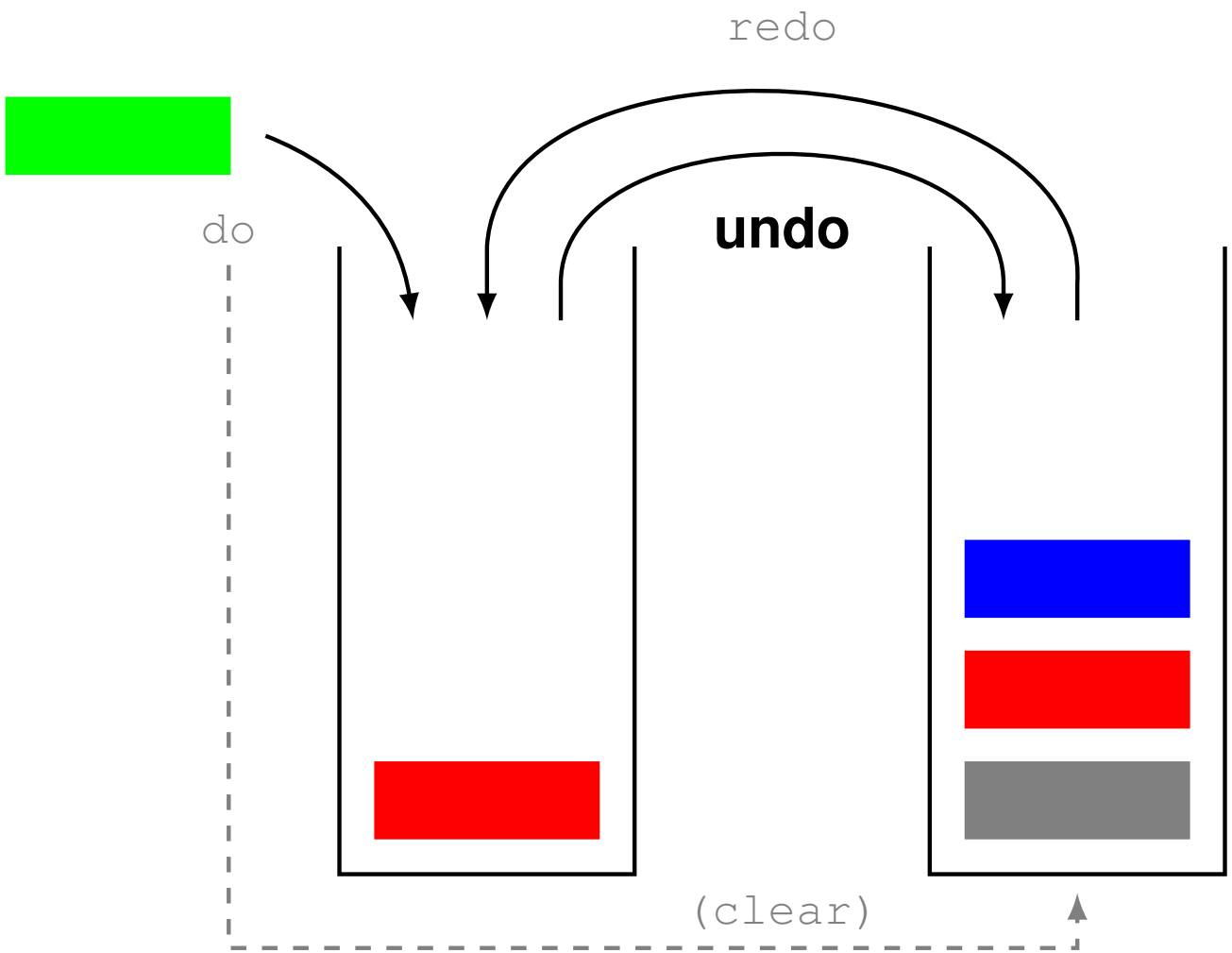

redo
do
undo
(clear)

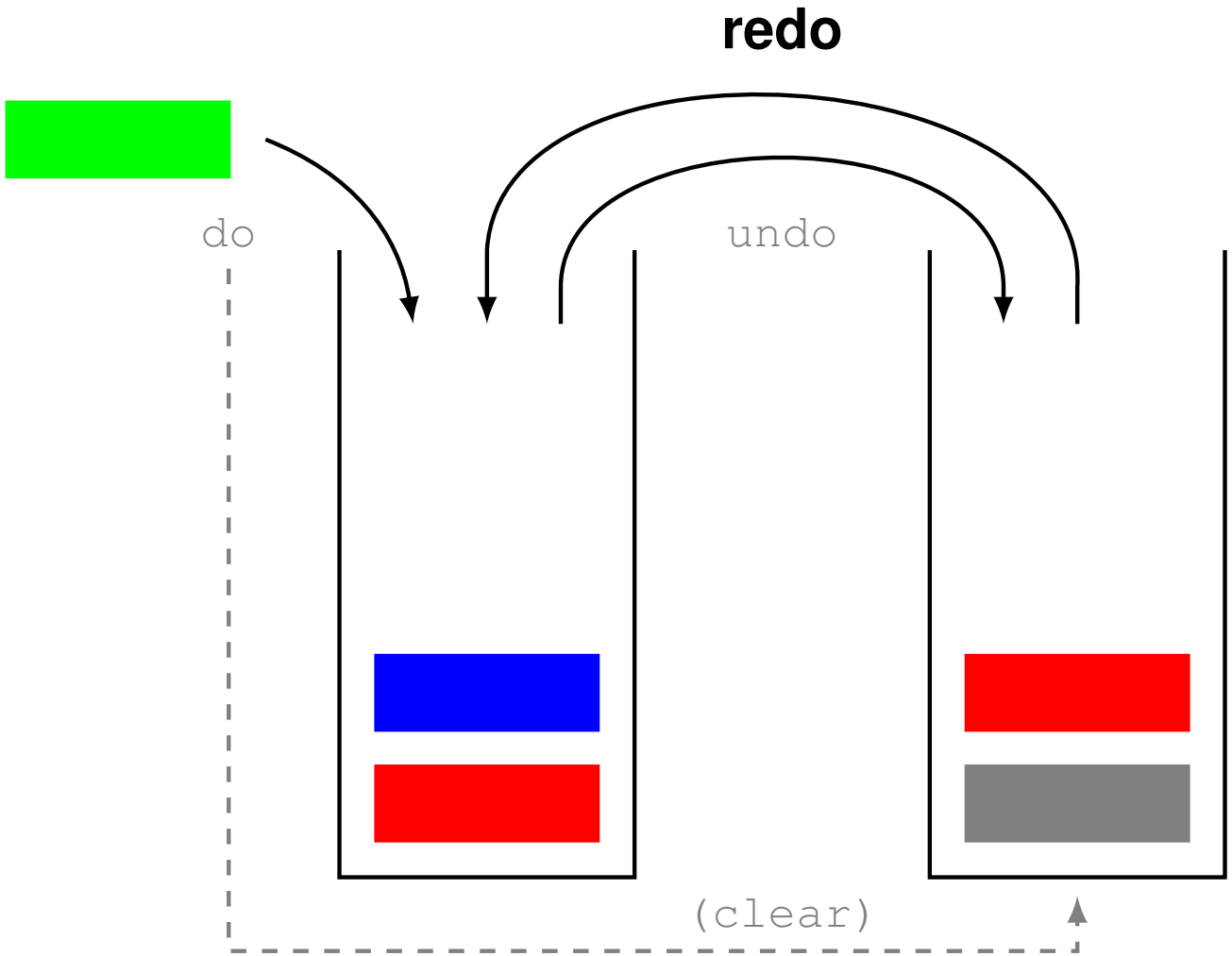

redo
undo
do
(clear)

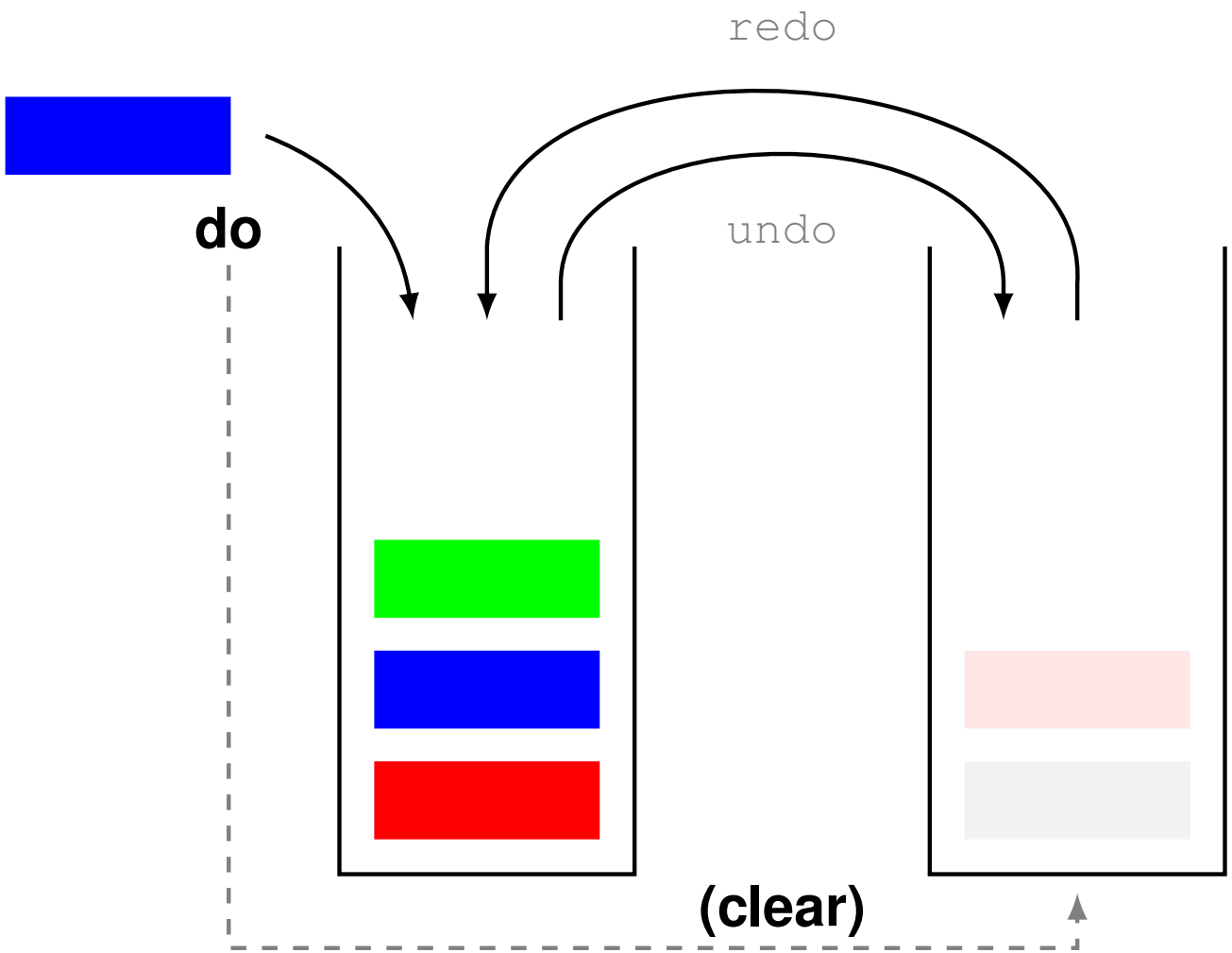

redo
do
undo
(clear)

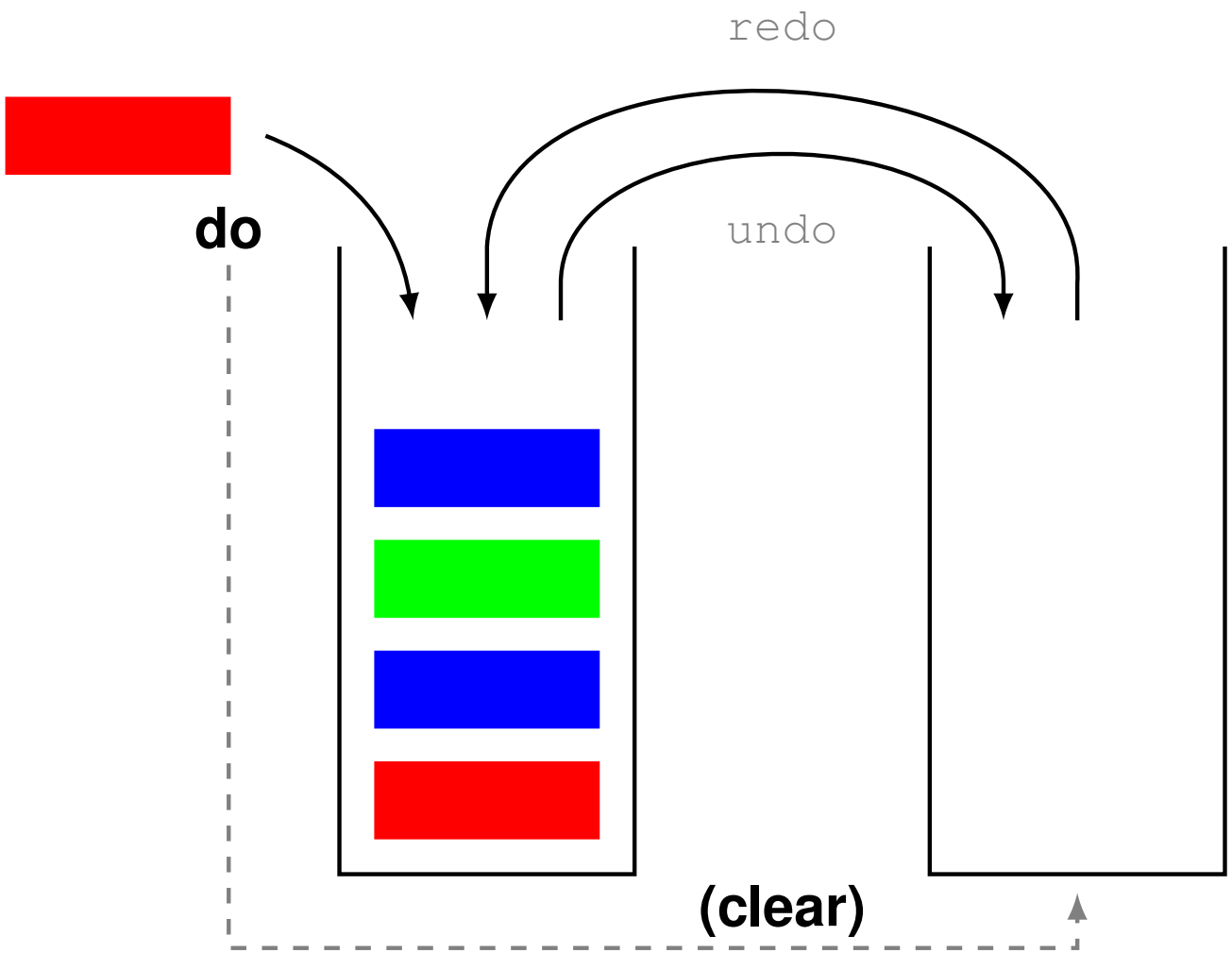

redo
do
undo
(clear)

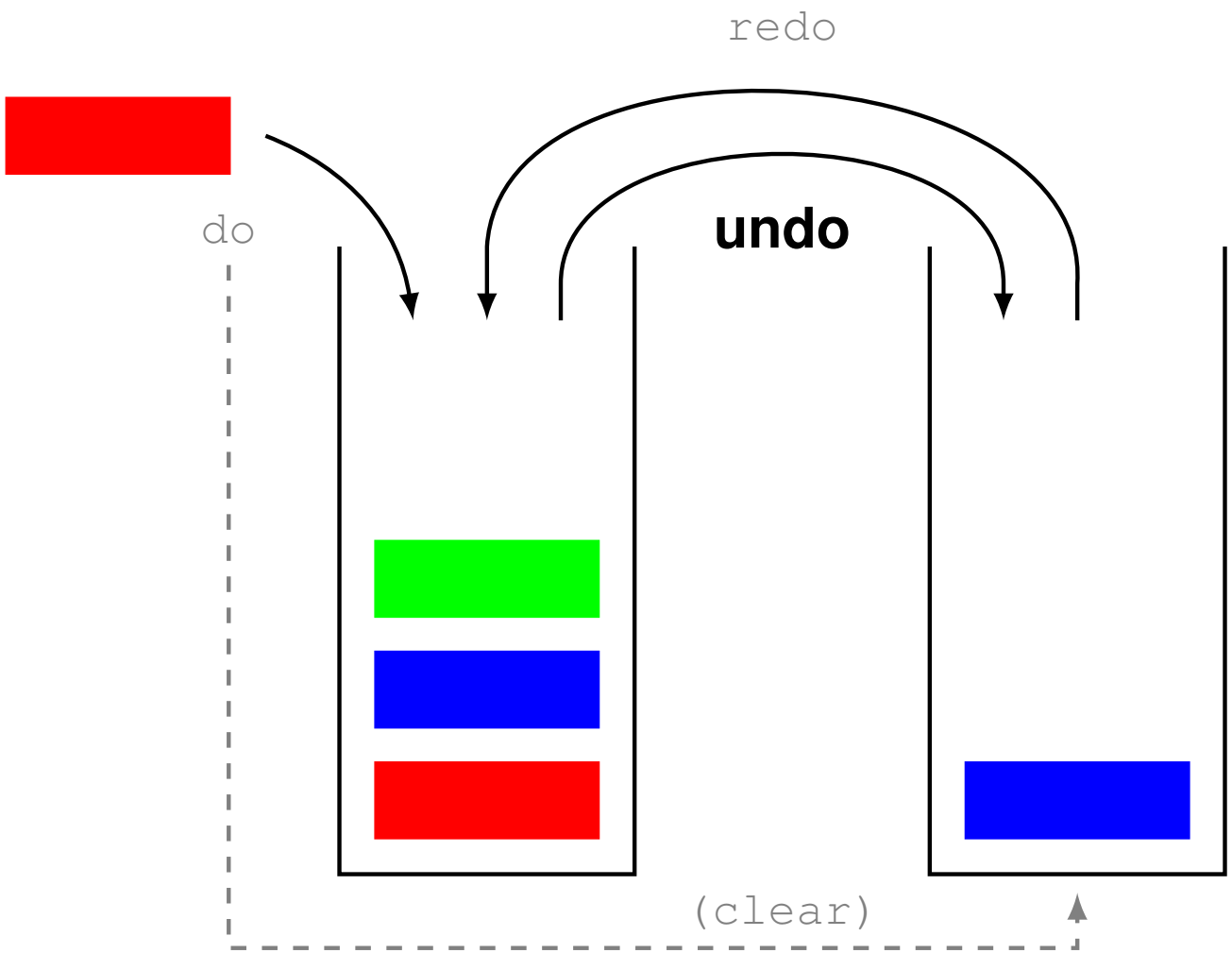

redo
do
undo
(clear)

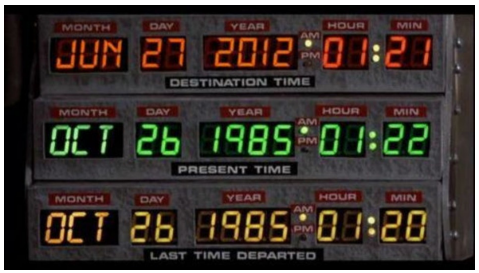

redo
do
undo
(clear)

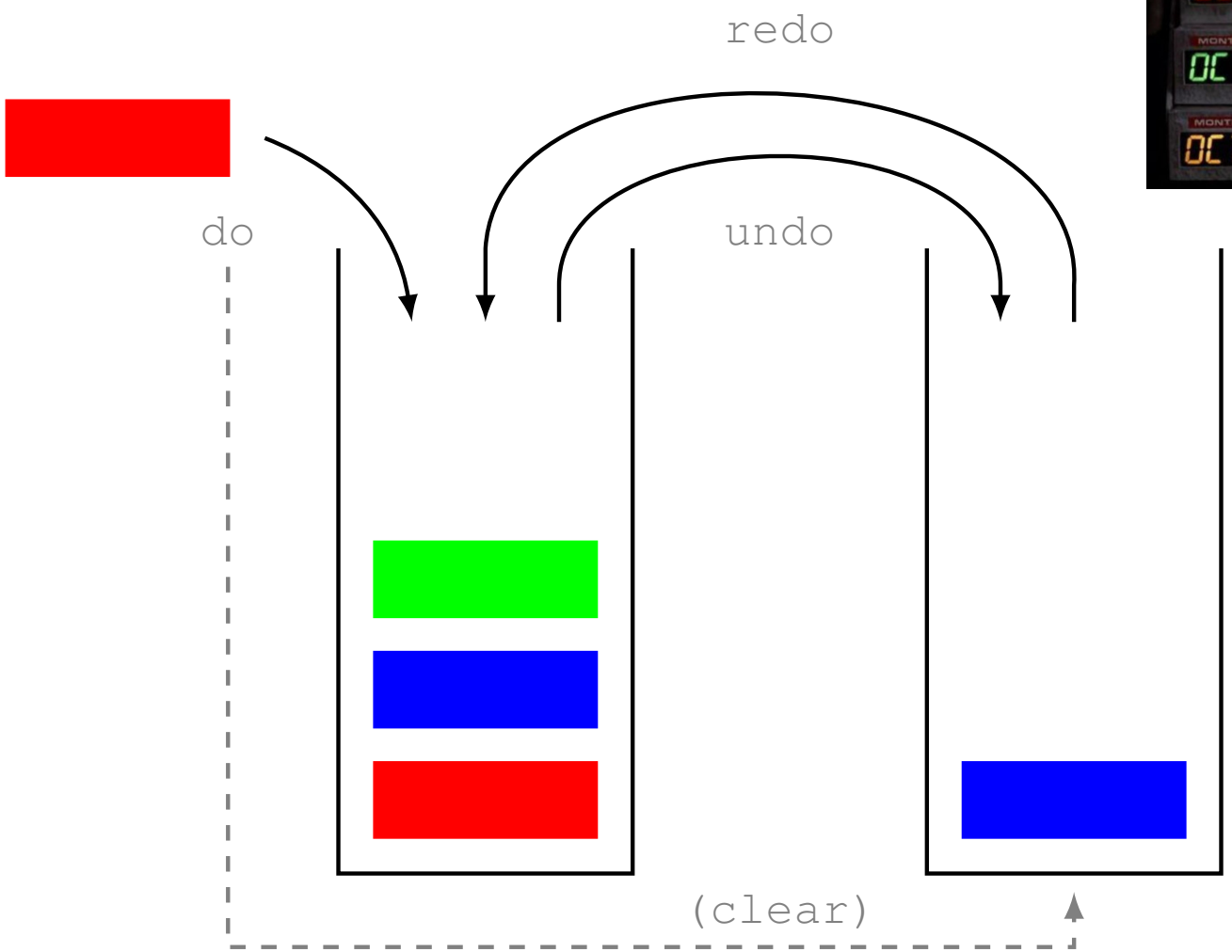

redo
do
undo
(clear)


redo
do
undo
(clear)


redo
do
undo
(clear)


redo
do
undo
(clear)


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

Compilation facile :
utilise des instructions
de branchement conditionnels

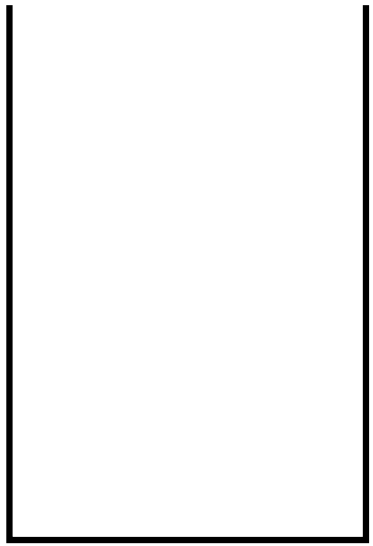
Définition itérative de factorielle (en C)

```
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)

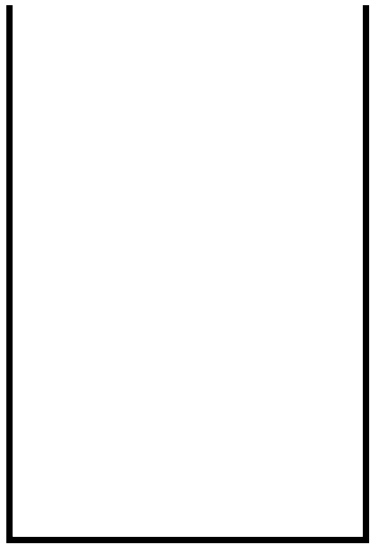


Définition récursive de factorielle (en Caml)

```
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`



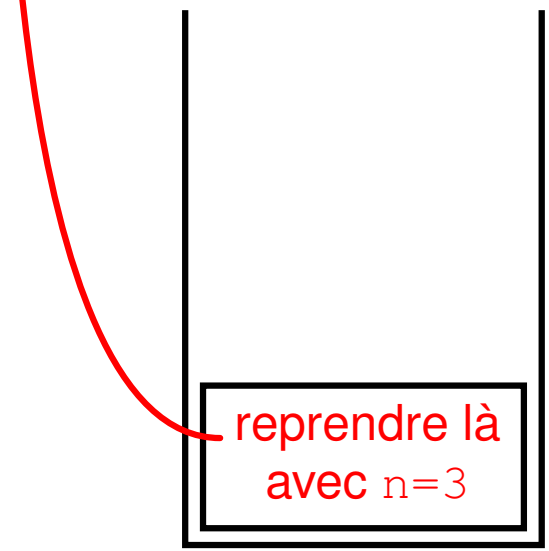
Définition récursive de factorielle (en Caml)

```
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)

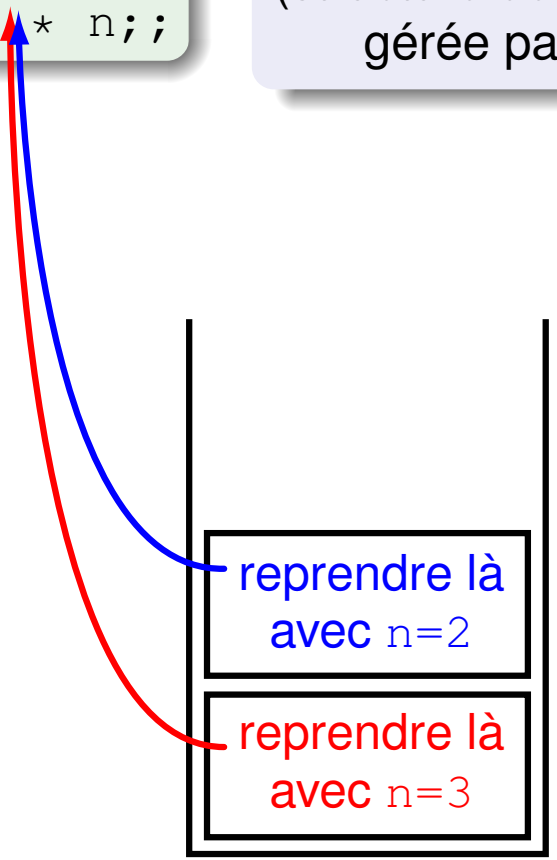
```
let rec fact n =
 if n < 2 then 1 else fact(n - 1) * n;;
```

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`

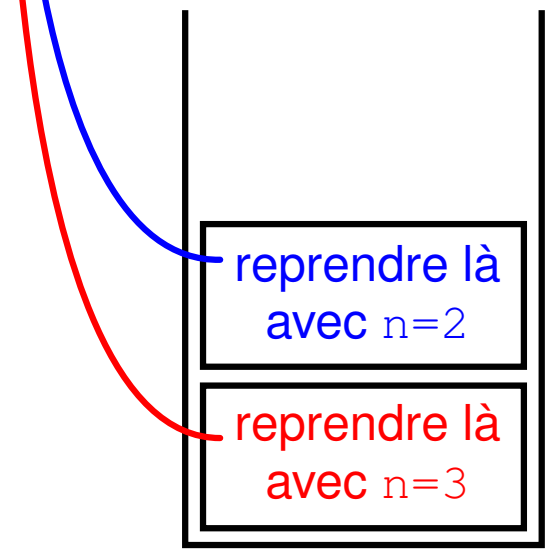


Définition récursive de factorielle (en Caml)

```
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`

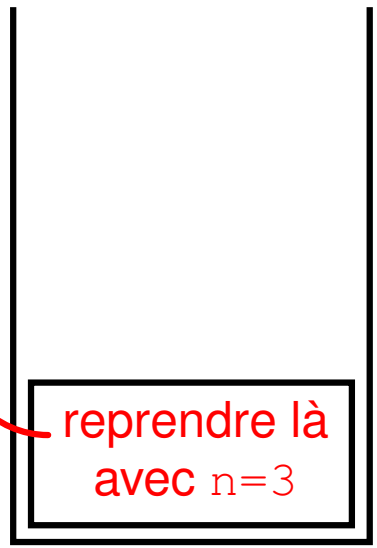


Définition récursive de factorielle (en Caml)

```
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`

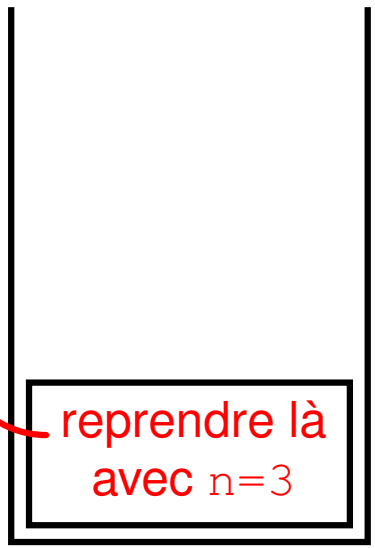


Définition récursive de factorielle (en Caml)

```
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`



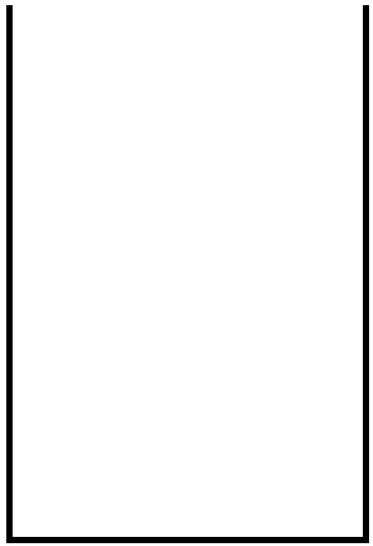
Définition récursive de factorielle (en Caml)

```
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`



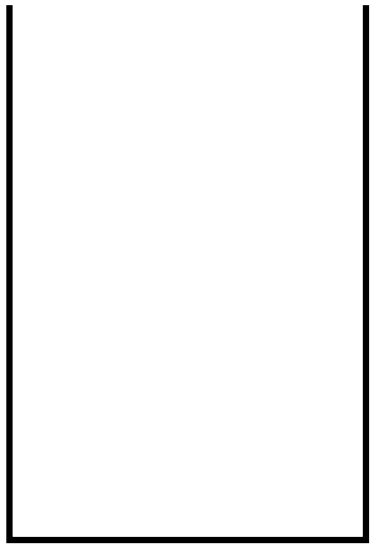
Définition récursive de factorielle (en Caml)

```
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`

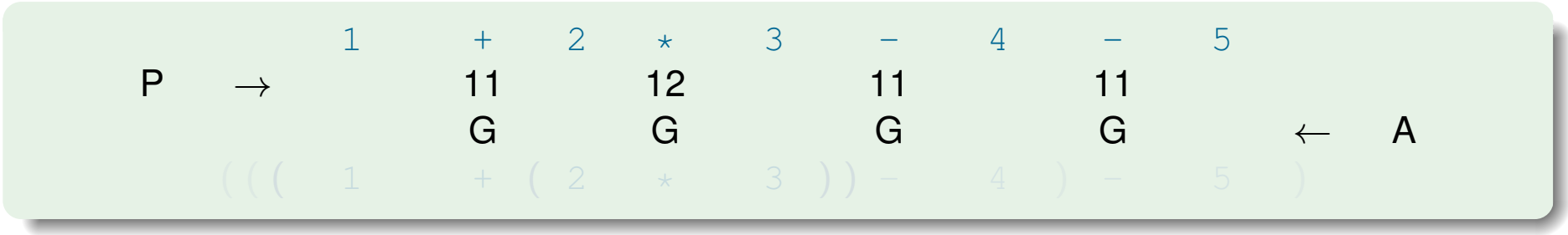
calcul de `2*n` avec `n=3`
arrêt avec `résultat=6`



	1	+	2	*	3	−	4	−	5		
P	→		11		12		11		11		
			G		G		G		G	←	A
	(	(	(	1	+	(	2	*	3	)	)

						−	4	)	−	5	)

		1	+	2	*	3	−	4	−	5		
P	→		11		12		11		11			
			G		G		G		G		←	A
		(((	1	+ (	2	*	3))	−	4	) −	5	)

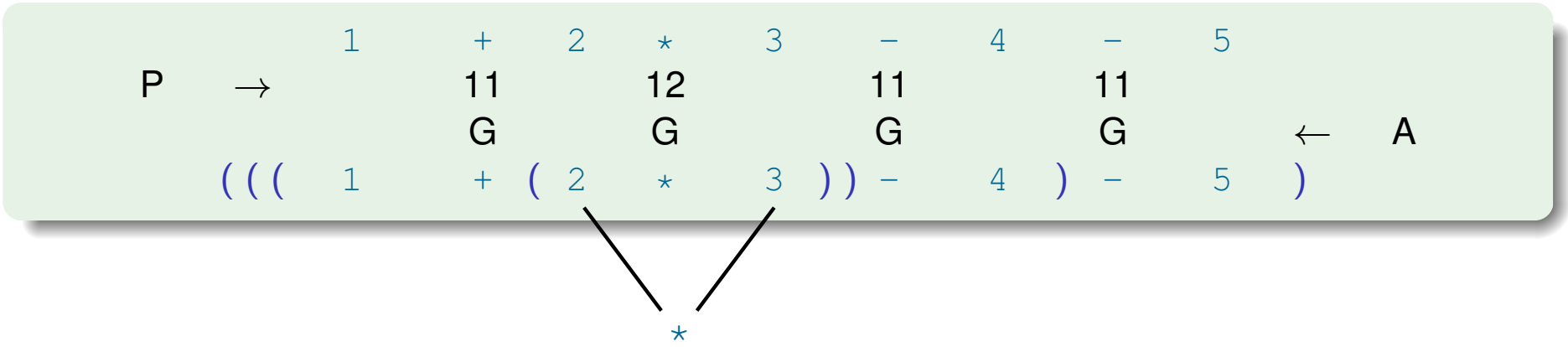

1 + 2 * 3 − 4 − 5
P → 11 12 11 11 ← A
G G G G
(((1 + (2 * 3)) − 4) − 5)

		1	+	2	*	3	−	4	−	5		
P	→		11		12		11		11			
			G		G		G		G		←	A
		(((	1	+	(2 | * | 3 |)) −	4	)	−	5	)	

		1	+	2	*	3	−	4	−	5		
P	→		11		12		11		11			
			G		G		G		G		←	A
	(((	1	+	(2 | * | 3 |))	−	4	)	−	5	)	

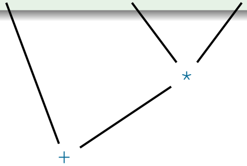
		1	+	2	*	3	−	4	−	5		
P	→		11		12		11		11			
			G		G		G		G		←	A
	(((	1	+	(2	*	3	)) −	4	) −	5	)	

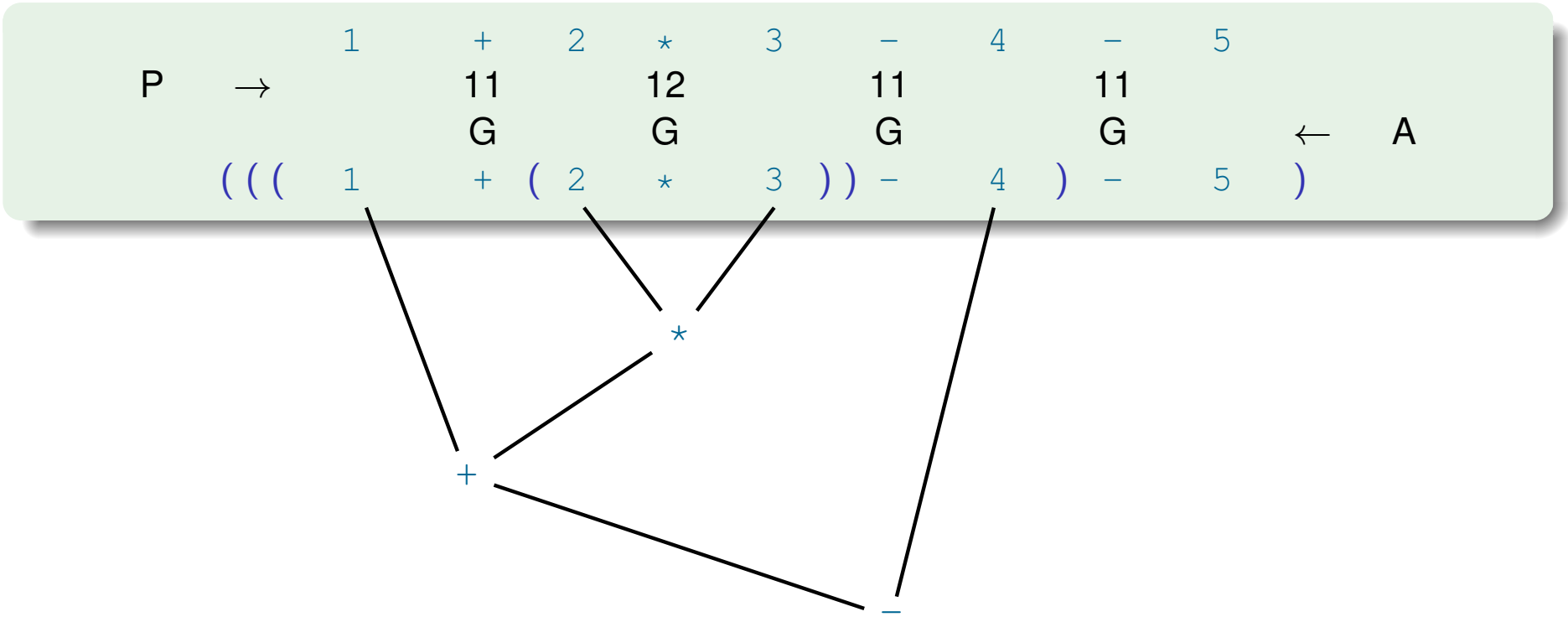
		1	+	2	*	3	−	4	−	5		
P	→		11		12		11		11			
			G		G		G		G		←	A
	(((	1	+	(2	*	3	)) −	4	) −	5	)	

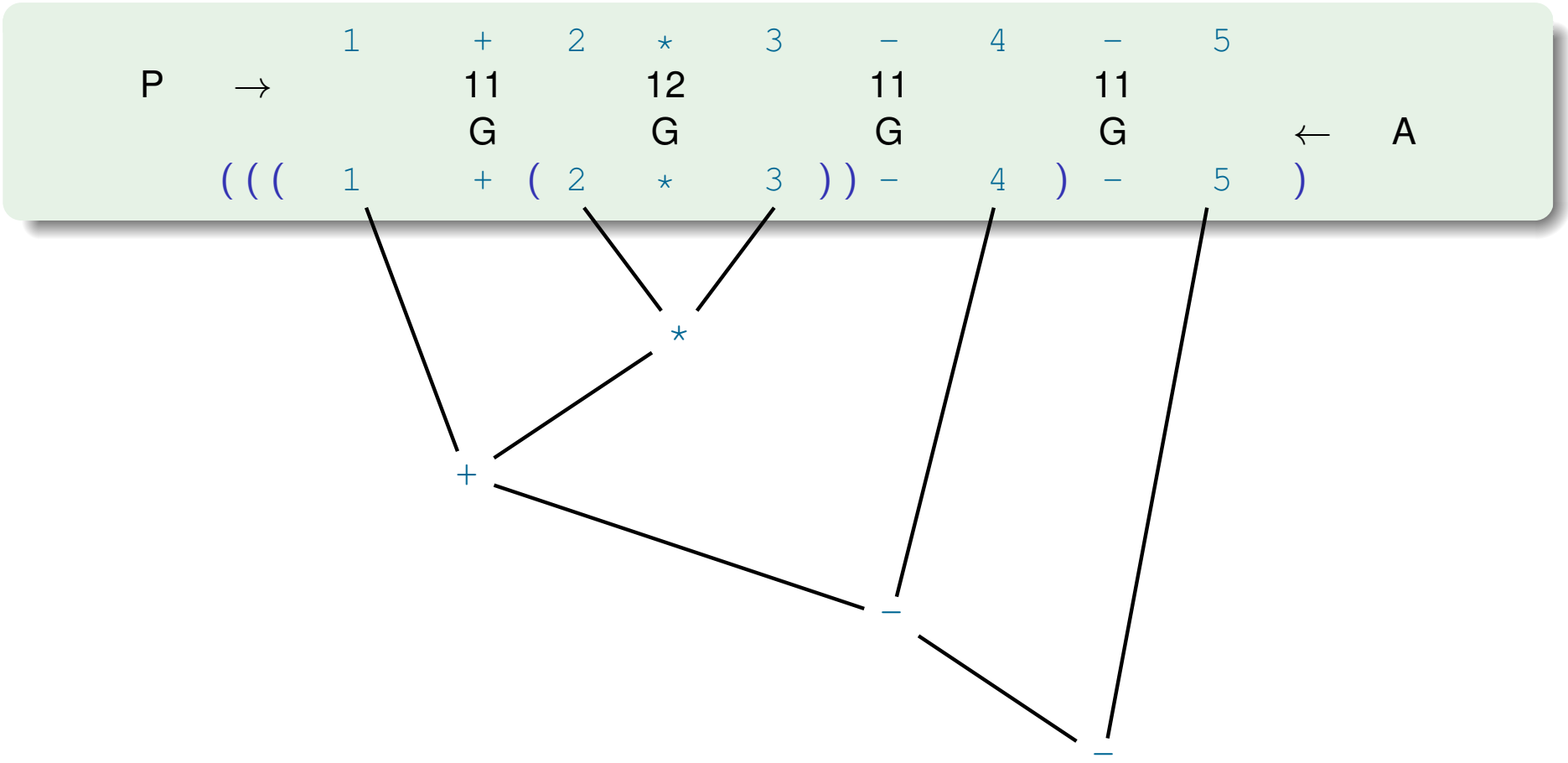

1 + 2 * 3 − 4 − 5
P → 11 12 11 11 ← A
G G G G
(((1 + (2 * 3)) − 4) − 5)
*

		1	+	2	*	3	−	4	−	5	
P	→		11		12		11		11		
			G		G		G		G	←	A

(((1 + (2 * 3)) − 4) − 5)




1 + 2 * 3 − 4 − 5
P → 11 12 11 11 ← A
G G G G
(((1 + (2 * 3)) − 4) − 5)
*
+
−


1 + 2 * 3 − 4 − 5
P → 11 12 11 11 ← A
G G G G
(((1 + (2 * 3)) − 4) − 5)
*
+
−
−

```
                5.1   +   d   *   't'   -   4.0   -   c5
P   →            11       12       11       11
                 G        G        G        G          ←   A
      (((   5.1   + ( d   *   't' )) -   4.0 ) -   c5   )
```