CM 8 de Compilation : Mini-JS machine Dynamic types and functions

Flavien BREUVART

March 12, 2026

Dynamic typing and implicit casts

In JS, which result are you expecting from the following expressions ?
(x not initialised)

```
true + 41                          ⟶
true + "!"                         ⟶
"the answer is " + 42              ⟶
"the answer is " + 4 + (1+true)    ⟶
x + " vs " + (1+x)                 ⟶
"result: " + ("test"?42:true)      ⟶
"result: " + (x?42:true)           ⟶
```

Dynamic typing and implicit casts

In JS, which result are you expecting from the following expressions ?
(x not initialised)

```
true + 41                          ⟶    42
true + "!"                         ⟶
"the answer is " + 42              ⟶
"the answer is " + 4 + (1+true)    ⟶
x + " vs " + (1+x)                 ⟶
"result: " + ("test"?42:true)      ⟶
"result: " + (x?42:true)           ⟶
```

Dynamic typing and implicit casts

In JS, which result are you expecting from the following expressions ?
(x not initialised)

`true + 41`	$\longrightarrow$	`42`
`true + "!"`	$\longrightarrow$	`"true!"`
`"the answer is " + 42`	$\longrightarrow$	
`"the answer is " + 4 + (1+true)`	$\longrightarrow$	
`x + " vs " + (1+x)`	$\longrightarrow$	
`"result: " + ("test"?42:true)`	$\longrightarrow$	
`"result: " + (x?42:true)`	$\longrightarrow$	

Dynamic typing and implicit casts

In JS, which result are you expecting from the following expressions ?
(x not initialised)

`true + 41`	$\longrightarrow$	`42`
`true + "!"`	$\longrightarrow$	`"true!"`
`"the answer is " + 42`	$\longrightarrow$	`"the answer is 42"`
`"the answer is " + 4 + (1+true)`	$\longrightarrow$	
`x + " vs " + (1+x)`	$\longrightarrow$	
`"result: " + ("test"?42:true)`	$\longrightarrow$	
`"result: " + (x?42:true)`	$\longrightarrow$	

Dynamic typing and implicit casts

In JS, which result are you expecting from the following expressions ?
(x not initialised)

`true + 41`	$\longrightarrow$	`42`
`true + "!"`	$\longrightarrow$	`"true!"`
`"the answer is " + 42`	$\longrightarrow$	`"the answer is 42"`
`"the answer is " + 4 + (1+true)`	$\longrightarrow$	`"the answer is 42"`
`x + " vs " + (1+x)`	$\longrightarrow$	
`"result: " + ("test"?42:true)`	$\longrightarrow$	
`"result: " + (x?42:true)`	$\longrightarrow$	

Dynamic typing and implicit casts

In JS, which result are you expecting from the following expressions ?
(x not initialised)

`true + 41`	$\longrightarrow$	`42`
`true + "!"`	$\longrightarrow$	`"true!"`
`"the answer is " + 42`	$\longrightarrow$	`"the answer is 42"`
`"the answer is " + 4 + (1+true)`	$\longrightarrow$	`"the answer is 42"`
`x + " vs " + (1+x)`	$\longrightarrow$	`"undefined vs NaN"`
`"result: " + ("test"?42:true)`	$\longrightarrow$	
`"result: " + (x?42:true)`	$\longrightarrow$	

Dynamic typing and implicit casts

In JS, which result are you expecting from the following expressions ?
(x not initialised)

```
true + 41                          ⟶   42
true + "!"                         ⟶   "true!"
"the answer is " + 42              ⟶   "the answer is 42"
"the answer is " + 4 + (1+true)    ⟶   "the answer is 42"
x + " vs " + (1+x)                 ⟶   "undefined vs NaN"
"result: " + ("test"?42:true)      ⟶   "result: 42"
"result: " + (x?42:true)           ⟶
```

Dynamic typing and implicit casts

In JS, which result are you expecting from the following expressions ?
(x not initialised)

`true + 41`	$\longrightarrow$	`42`
`true + "!"`	$\longrightarrow$	`"true!"`
`"the answer is " + 42`	$\longrightarrow$	`"the answer is 42"`
`"the answer is " + 4 + (1+true)`	$\longrightarrow$	`"the answer is 42"`
`x + " vs " + (1+x)`	$\longrightarrow$	`"undefined vs NaN"`
`"result: " + ("test"?42:true)`	$\longrightarrow$	`"result: 42"`
`"result: " + (x?42:true)`	$\longrightarrow$	`"result: true"`

Dynamic typing and implicit casts

In JS, which result are you expecting from the following expressions ?
(x not initialised)

```
true + 41                          ⟶   42
true + "!"                         ⟶   "true!"
"the answer is " + 42              ⟶   "the answer is 42"
"the answer is " + 4 + (1+true)    ⟶   "the answer is 42"
x + " vs " + (1+x)                 ⟶   "undefined vs NaN"
"result: " + ("test"?42:true)      ⟶   "result: 42"
"result: " + (x?42:true)           ⟶   "result: true"
```

Dynamic types are only known at execution time
and the operator "+" behave diferently depending on the type

Not the case in C that does the casts statically.

Casts are implicit

In JS, "`true+3`" trigers an implicit cast

The boolean `true` is casted in the number `1`.

Instruction	semantic	before	after
BoToNb	`if Pop then Push(1) else Push(0)`	`b:stk`	`n:stk`
NbToBo	`Push(Pop` $\neq_{\mathtt{f}}$ `0)`	`n:stk`	`b:stk`

Those instruction allows for explicit cast in the assembly

CsteBo True
BoToNb
CsteNb 3
AddiNb

How to cast from variable ?

A same variable can contain a number or a boolean, how to know which cast performing ? Two solutions :

Static type inference

Impossible in JS :

```
x = true;
if (b) x=0;
x+3;
```

Testing types dynamicaly

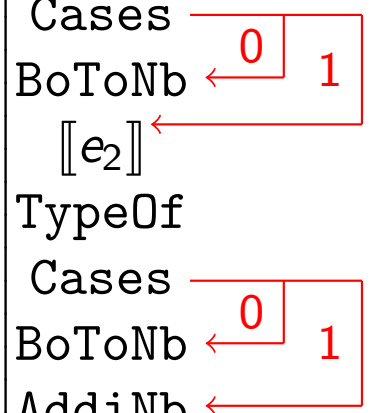
We will naively
test each type all the time
(partial inference and JIT help)

Instruction	semantics	before	after
TypeOf	Peak a value, returns 0 on a Boolean, 1 on a Number...	`v:stk`	`n:v:stk`
Cases	`PC := PC + Floor(Pop) + 1;`	`n:stk`	`stk`
Cases p	`PC := PC + p * Floor(Pop) + 1;`	`n:stk`	`stk`

A more reasonable code for sums

Instruction	semantique	before	after
TypeOf	Peak a value, return 0 son a Booleen, 1 on a Number...	`v:stk`	`n:v:stk`
Cases	$\texttt{PC} := \texttt{PC} + \texttt{Floor}(\texttt{Pop}) + 1;$	`n:stk`	`stk`

$$[\![e_1 + e_2]\!] = \left|\begin{array}{l} [\![e_1]\!] \\ \texttt{TypeOf} \\ \texttt{Cases} \\ \texttt{BoToNb} \\ [\![e_2]\!] \\ \texttt{TypeOf} \\ \texttt{Cases} \\ \texttt{BoToNb} \\ \texttt{AddiNb} \end{array}\right.$$



Cases instruction : a relativelly comon but dangerous instruction

The instruction `Cases` is a jump which offset is gotten dynamically, replacing many "if" by a unic instruction.

Cases	$\texttt{PC} := \texttt{PC} + \texttt{Pop} + 1;$	`n:stk`	`stk`
Cases p	$\texttt{PC} := \texttt{PC} + \texttt{p} * \texttt{Pop} + 1;$	`n:stk`	`stk`

It is used for

- dynamic types
- pattern-matching,
- switches in C,
- brutal optimisations

Default: in order to correctly align, one have to add `Noop` (or any other instruction) that are useless lines.

To understand how powerfull it can be: play TIS100

The Undefined type

Non initialised variable in JS

In JS if a variable is used before being initialised,
its value is `undefined` which type is `Undefined`.

In the mini-JS machine

Idem : the value `undefined` is pushed.
Additionally, `TypeOf` on an `Undefined` results in the number 3.

The type "`Undefined`" is in the project !

In the project, the undefined case has to be treated, so that their behavior mimic that of JS.

Type `Strings` are not mandatory in the project

Treated in TD.

Complexity of the assembly language vs complexity of the virtual machine

Dynamic types

- complex assembly code,
- simple machine,
- everything by hand.

Functions, ...

- simple assembly code,
- instructions/structures abstrait and complex.

Complexity of the assembly language vs complexity of the virtual machine

Dynamic types

- complex assembly code,
- simple machine,
- everything by hand.

Functions, ...

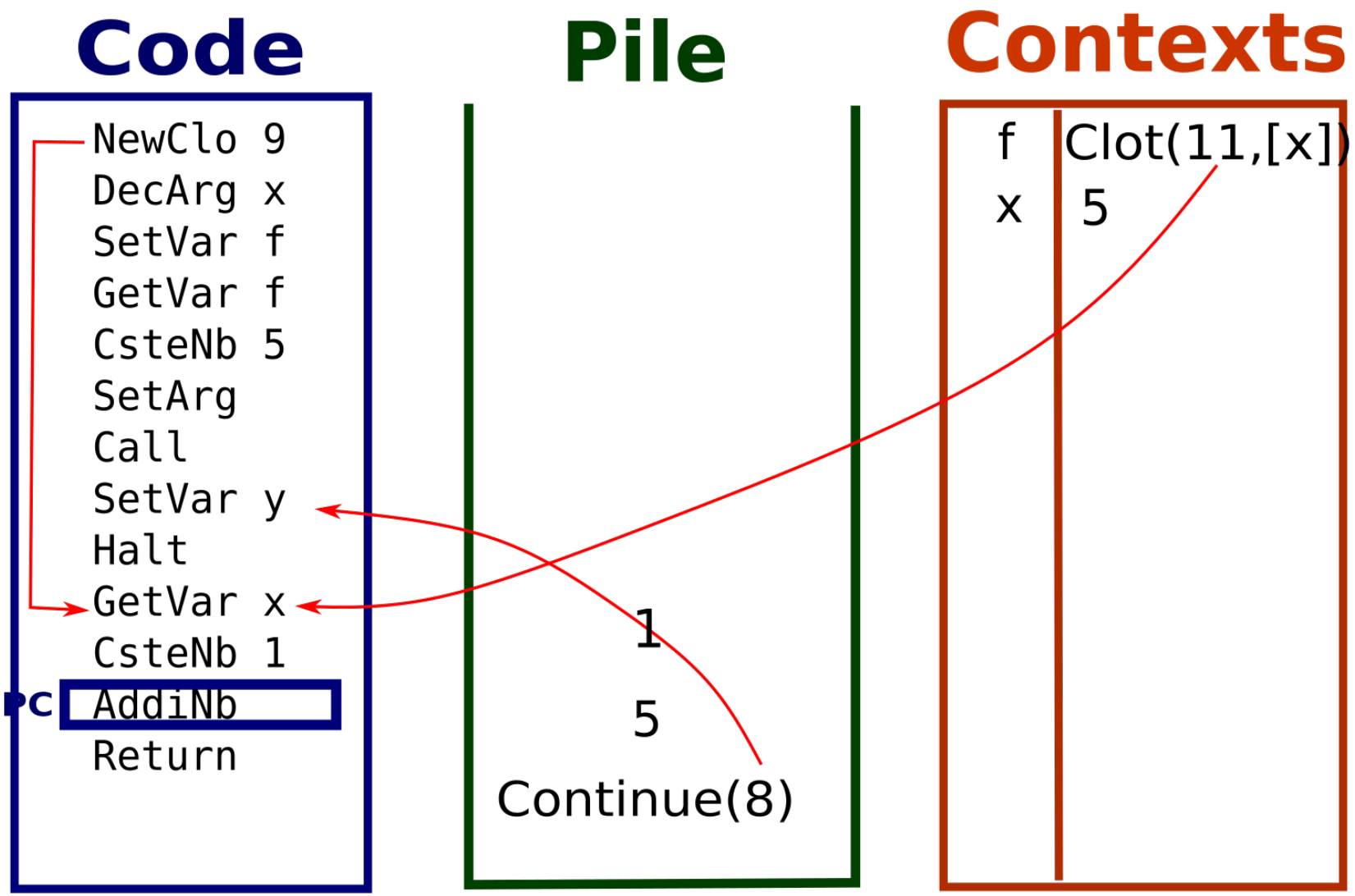
- simple assembly code,
- instructions/structures abstrait and complex.

Good practice in programming: not to mix codes that are

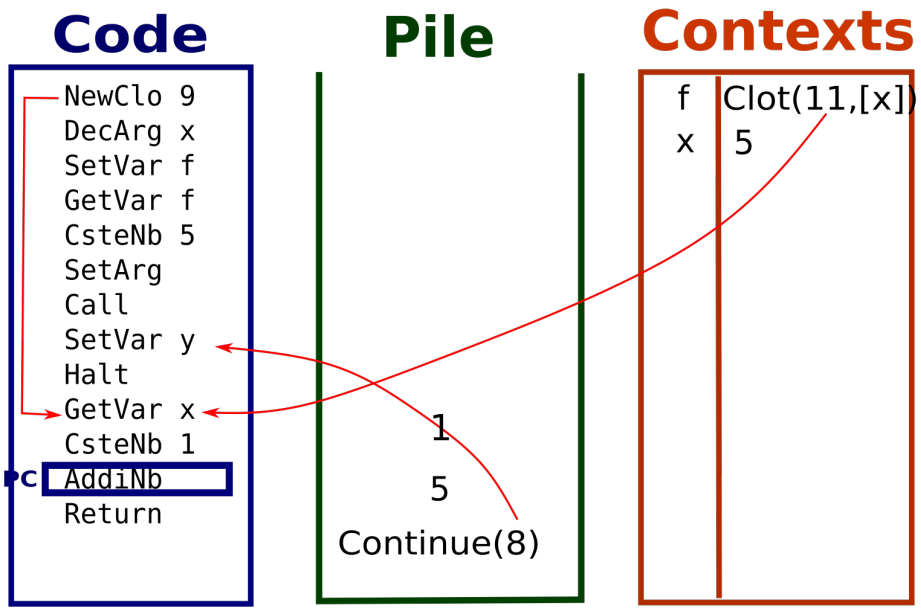
- low level / dirty / opimised and those that are
- high level / complexe / efficient.

$\rightsquigarrow$ our choice is pedagogical but do not respect good practices.

Naif encoding of functions



Closures and continuations : code pointers



Closure

Points to the starting instruction of the code of a function

Continuation

Point to the call-back instruction of a running function call

Closures also contain the name of their arguments

Closures "are" the functions themselves.

Naive translation of functionsT

```
/*main code*/
...
f(42,x)
...
function f(y,z) {
/*code of f*/
...
}
```

becomes

```
# header
NewClot funf
DecArg y
DecArg z
SetVar f
...
# main code
...
getVar f
CstNb 42
SetArg
GetVar x
SetArg
...
Halt
#code of f
funf  ...
```

Closure (`l`) : function seen as a value

A closure is a value

- can be pushed on the stack,
- can be associated with a variable name,
- has a type in the machine (`l`).

It contains

- a pointer toward the code of the funtion,
- a stack of names for variables,
- ... (seen later)

NewClo off	`Push(NewCloture{code = PC+off+1})`	`stk`	`l:stk`
DclArg name	`Pull.args.Push(name)`	`l:stk`	`l:stk`
SetArg	`Set(Pull.args.Pop, Pop)`	`v:l:stk`	`l:stk`

Continuation (t) : return information (invisible)

A continuation is not a value

- can bin pushed on the stack,
- but not in the manipulated in the context,
- it has a type t but crashed on **TypeOf**.

It contains

- a pointer toward the line that called the function,
- ... (seen later)

(Continuation could be made values but would require to copy the stack.)

Call	`clot = Pop;`
	`Push(NewContinuation{code = PC})`
	`PC := clot.code`
		`l:stk`	`t:stk`

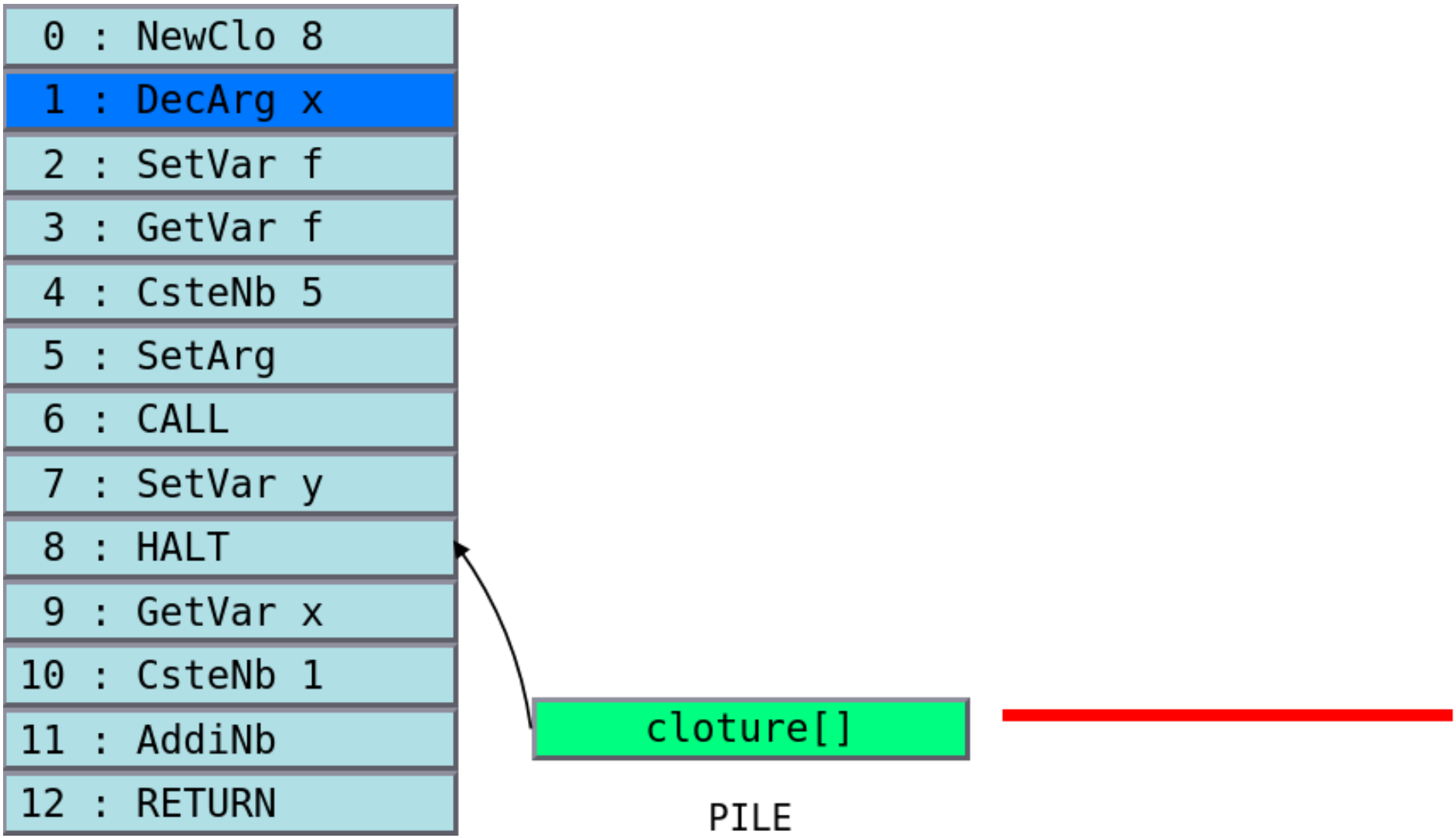
Return	`res = Pop; PC := Pop.code; Push(res);`	`v:t:stk`	`v:stk`

Example : y=f(5); function f(x) {return x+1;}

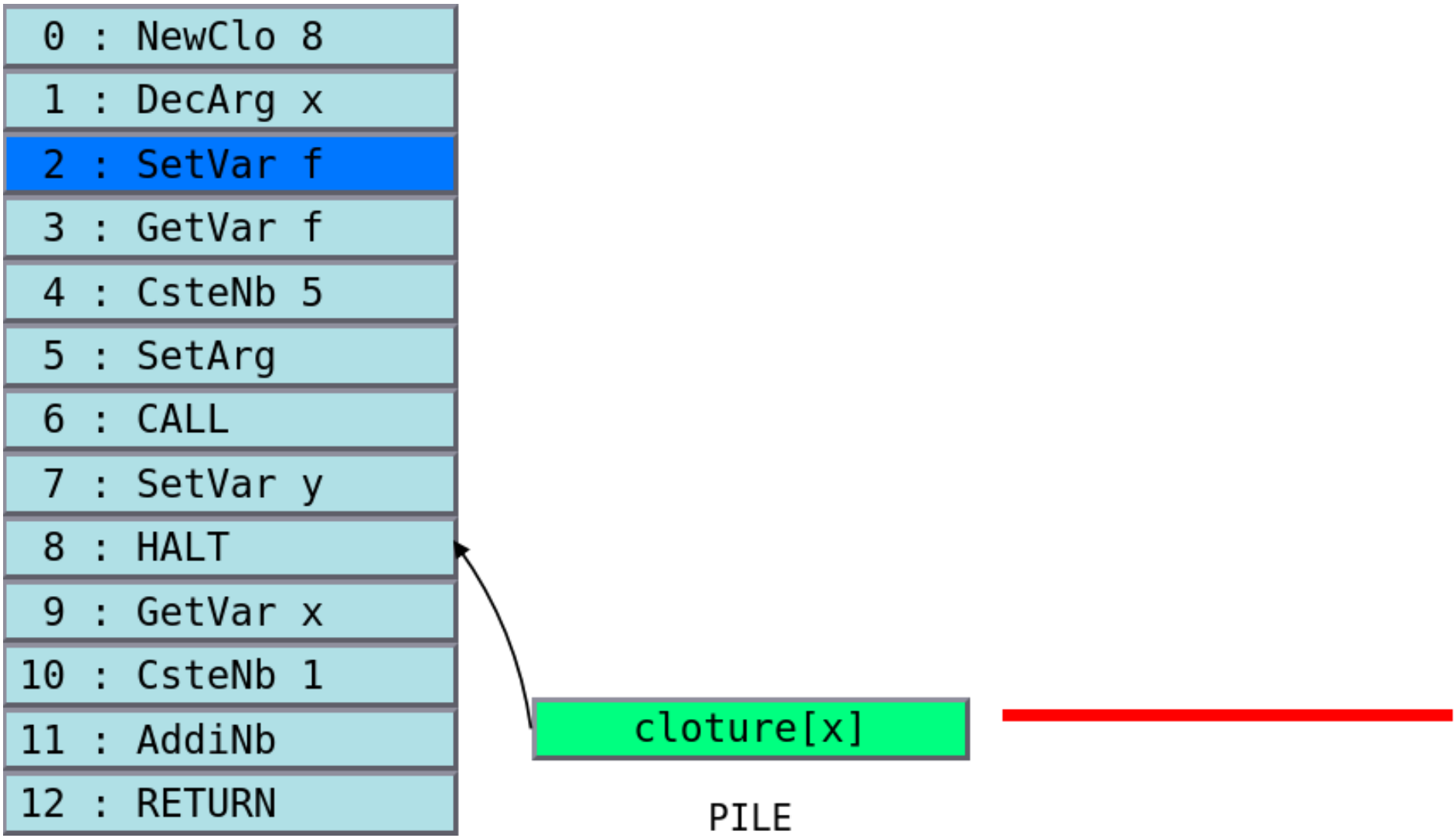
```
 0 : NewClo 8
 1 : DecArg x
 2 : SetVar f
 3 : GetVar f
 4 : CsteNb 5
 5 : SetArg
 6 : CALL
 7 : SetVar y
 8 : HALT
 9 : GetVar x
10 : CsteNb 1
11 : AddiNb
12 : RETURN
```

PILE

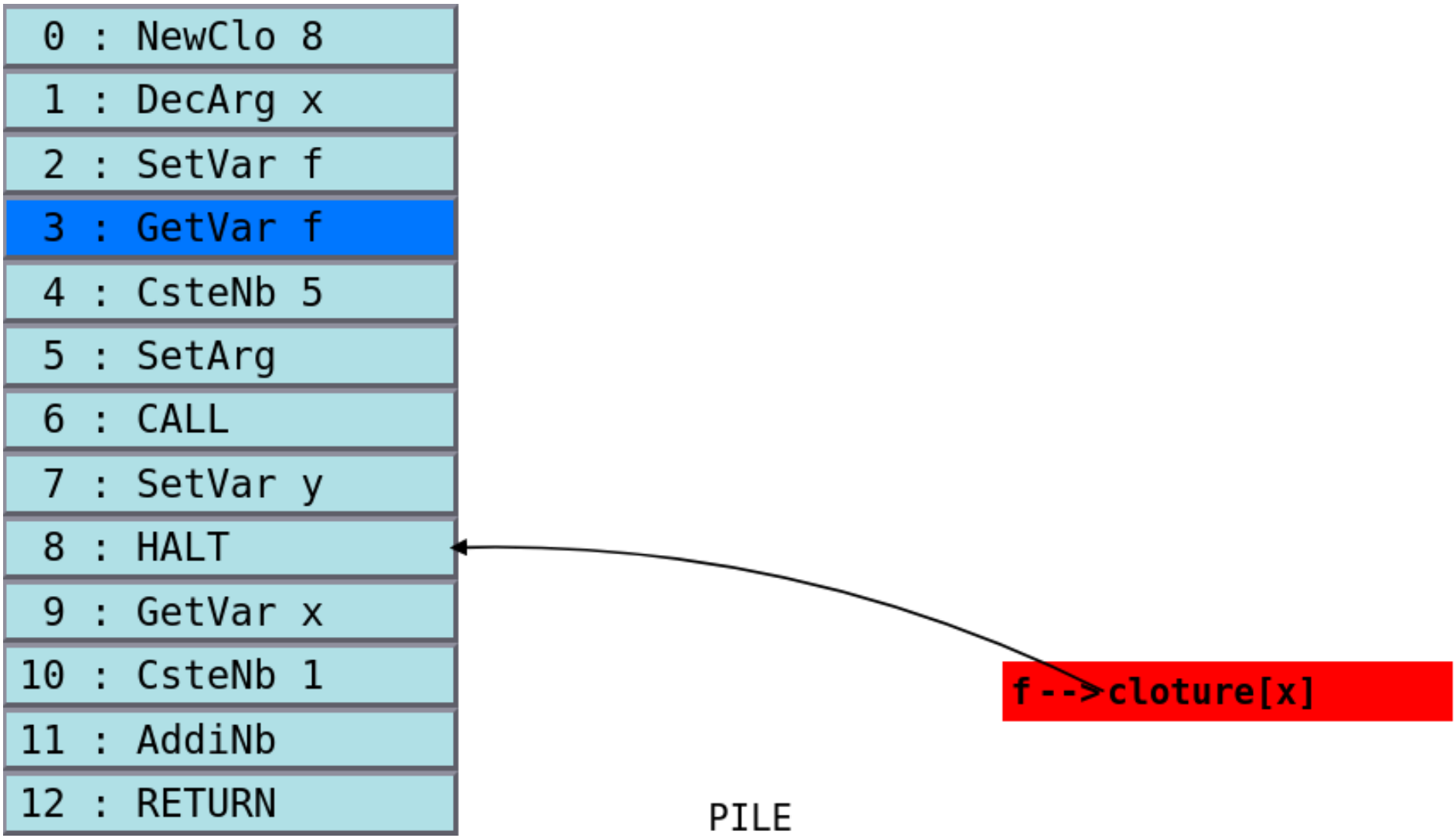
Example : y=f(5); function f(x) {return x+1;}



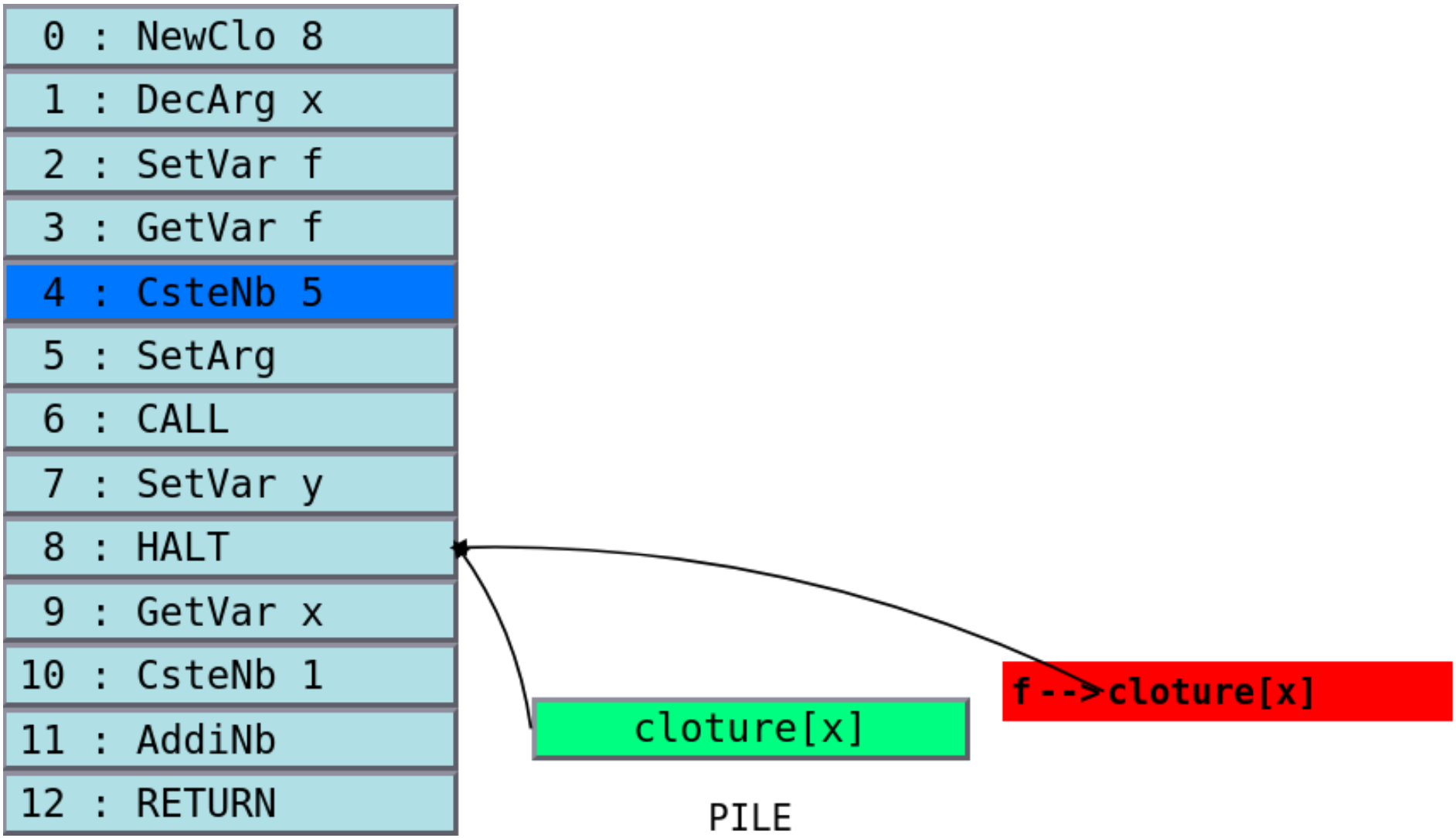
Example : y=f(5); function f(x) {return x+1;}



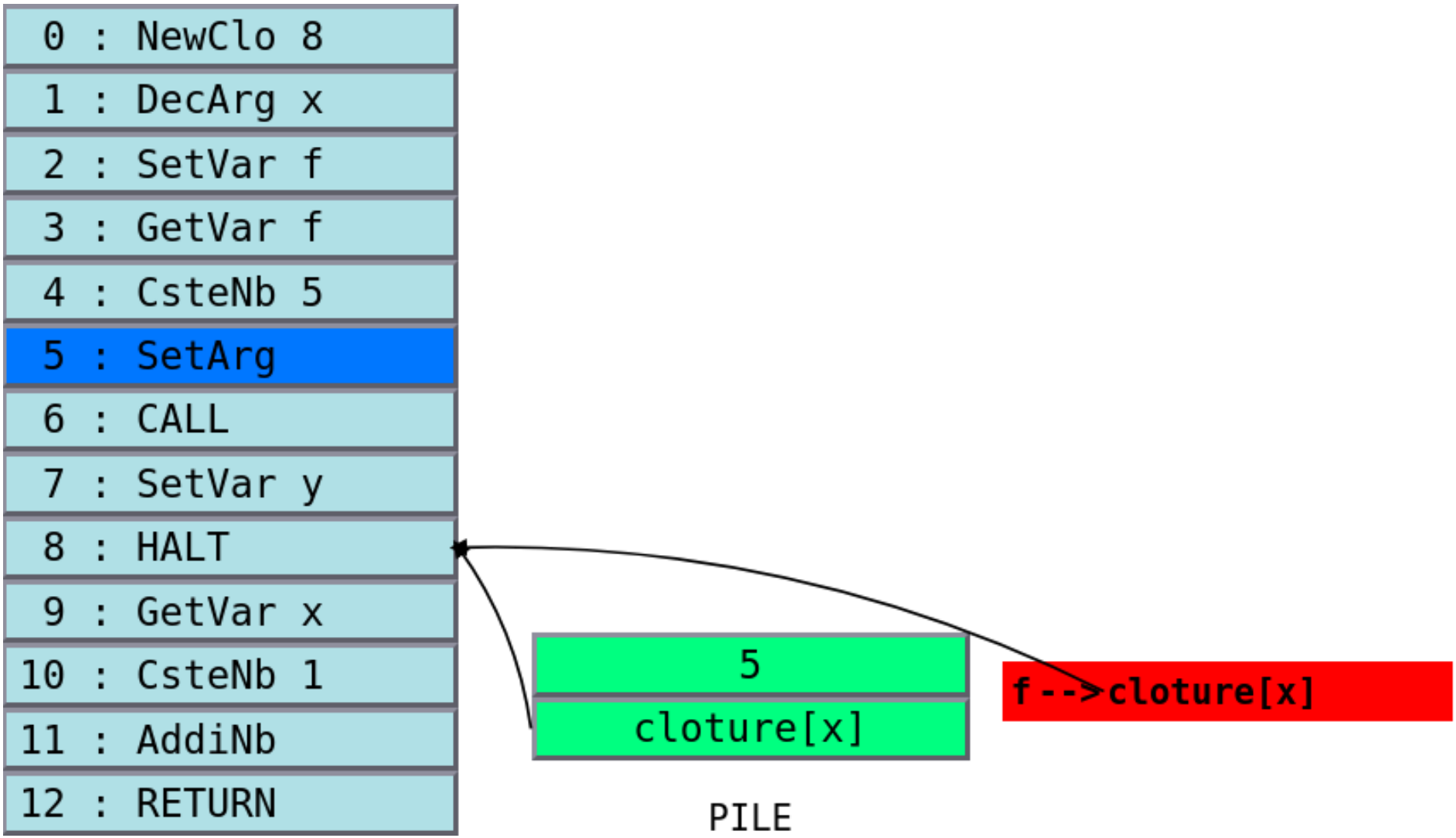
Example : y=f(5); function f(x) {return x+1;}



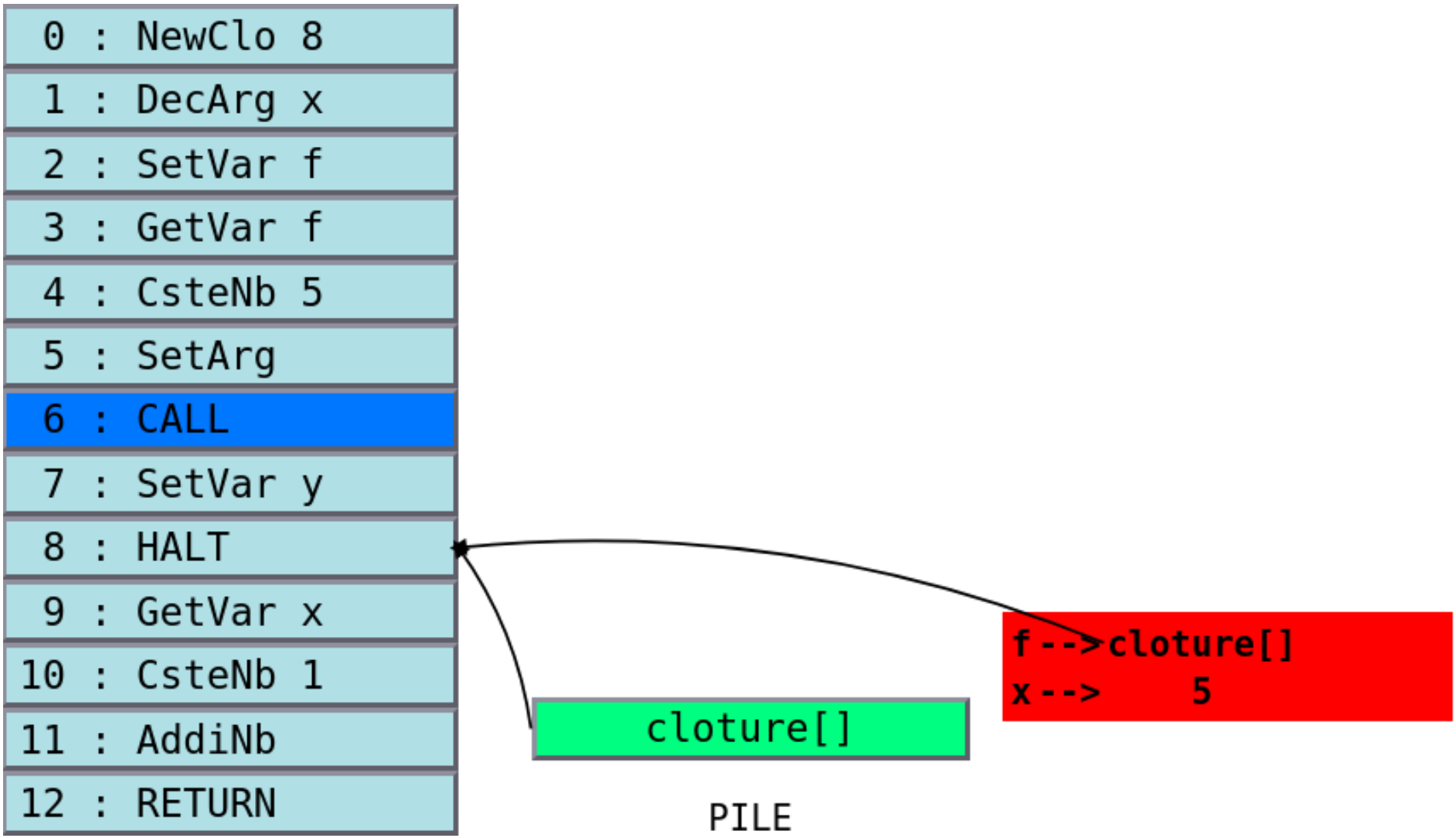
Example : y=f(5); function f(x) {return x+1;}



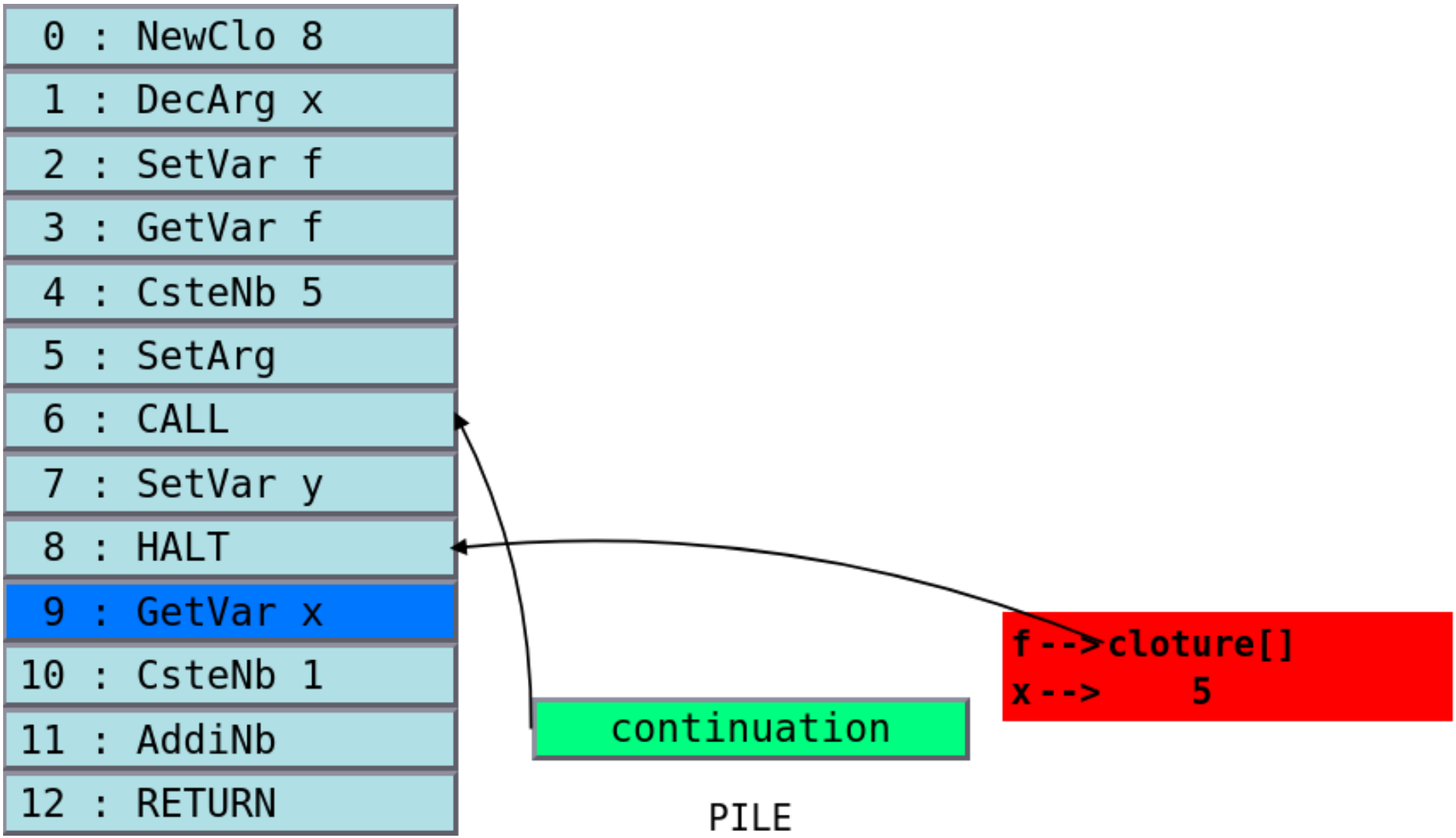
Example : y=f(5); function f(x) {return x+1;}



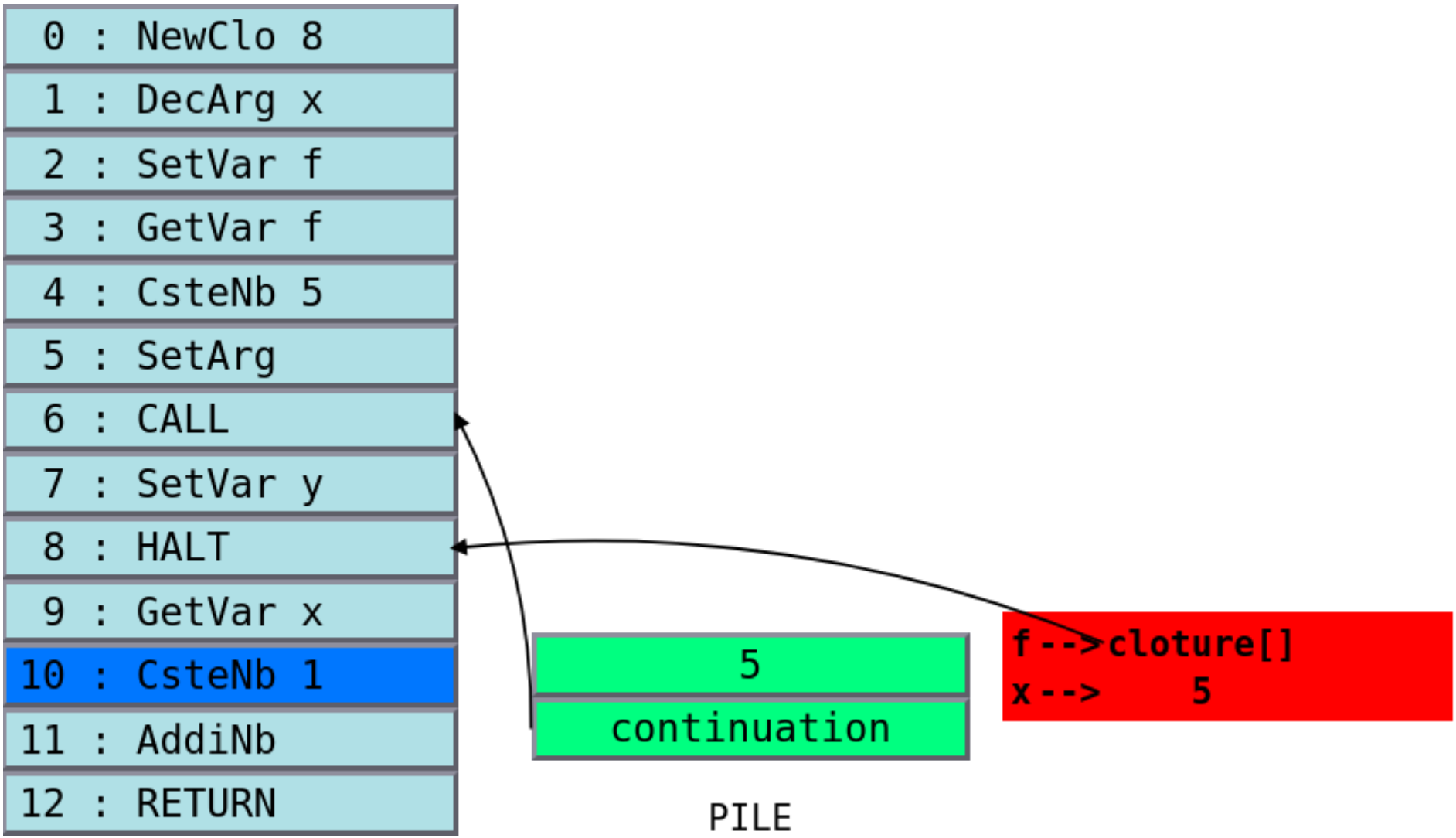
Example : y=f(5); function f(x) {return x+1;}



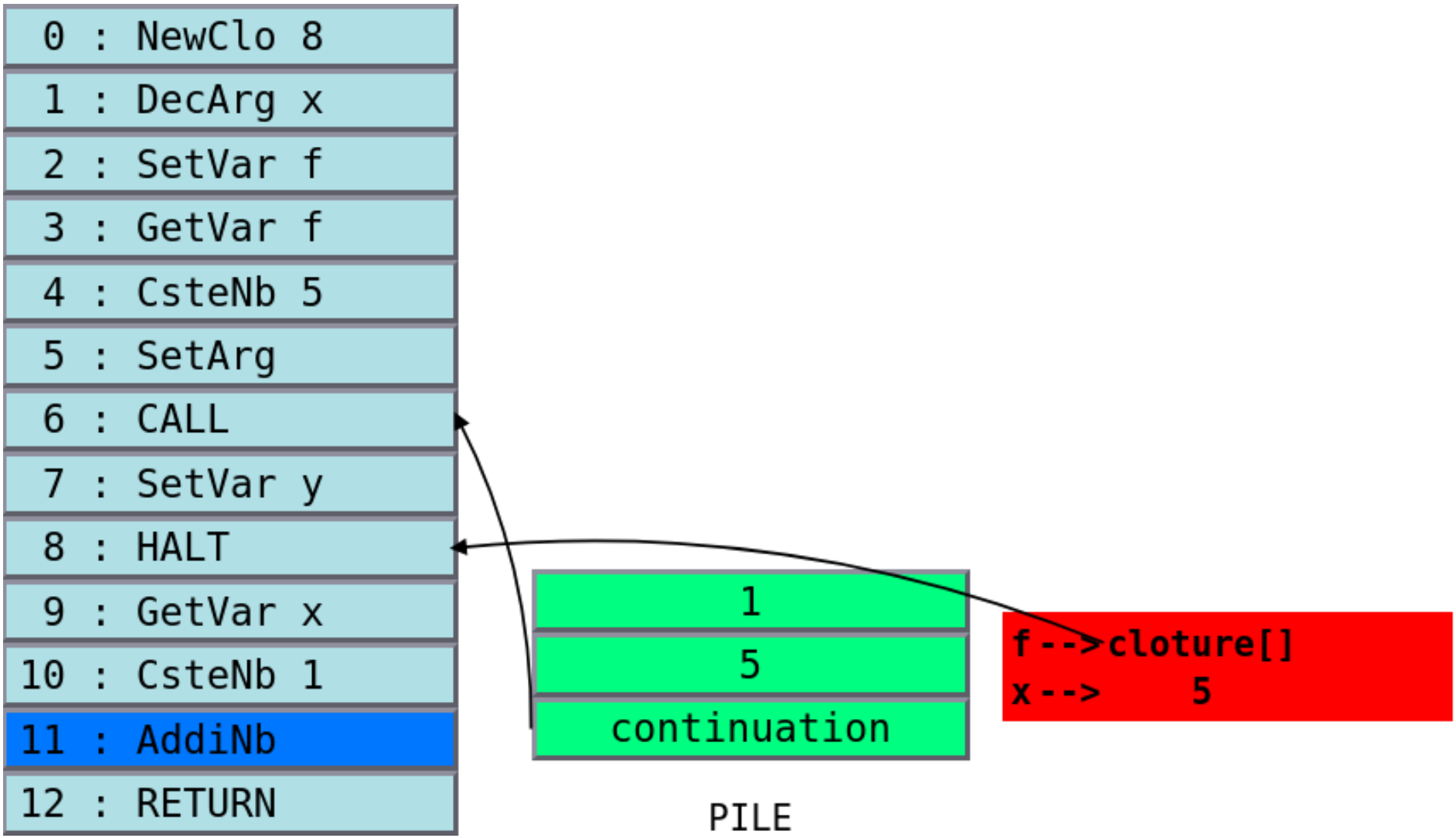
Example : y=f(5); function f(x) {return x+1;}



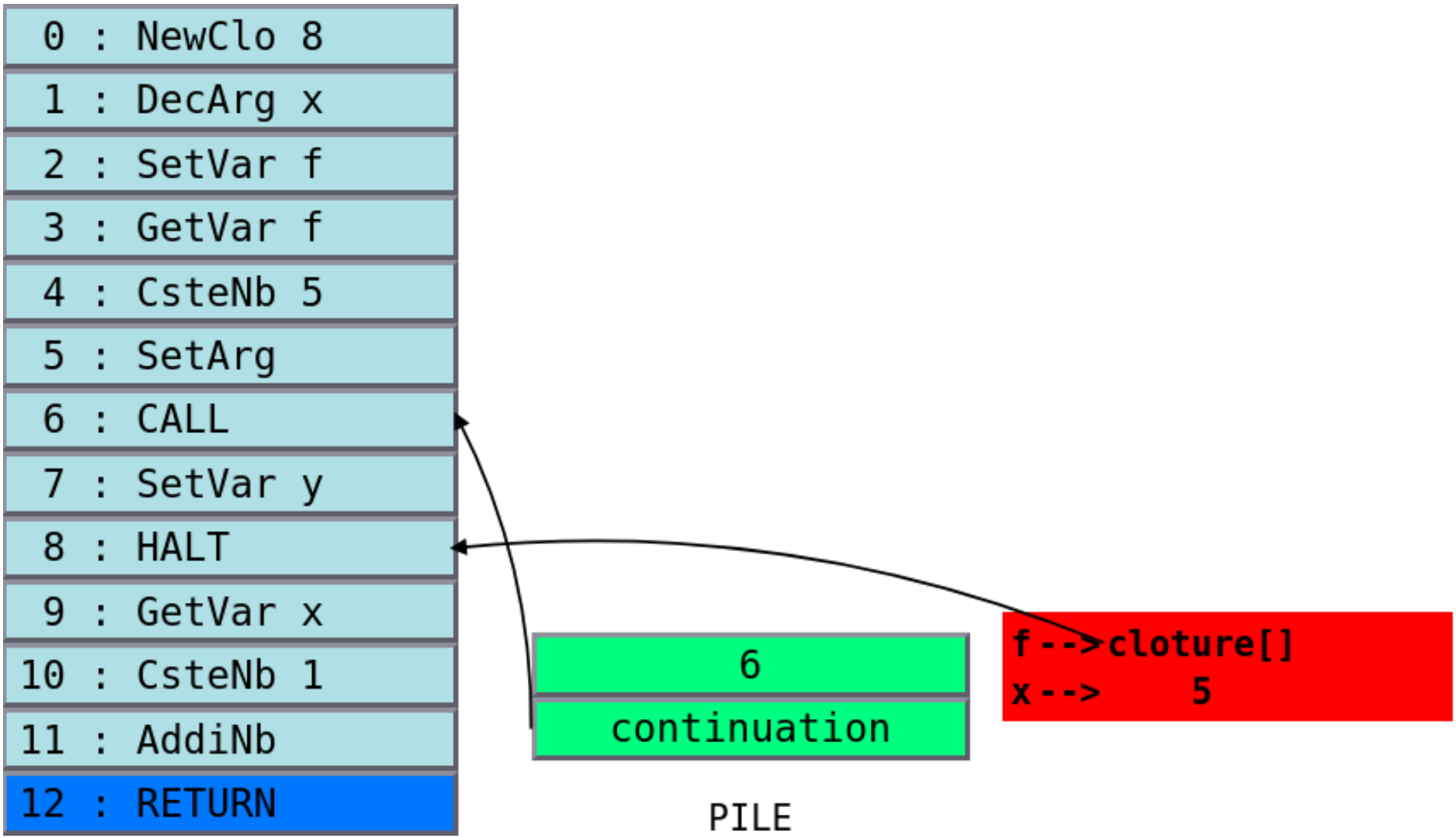
Example : y=f(5); function f(x) {return x+1;}



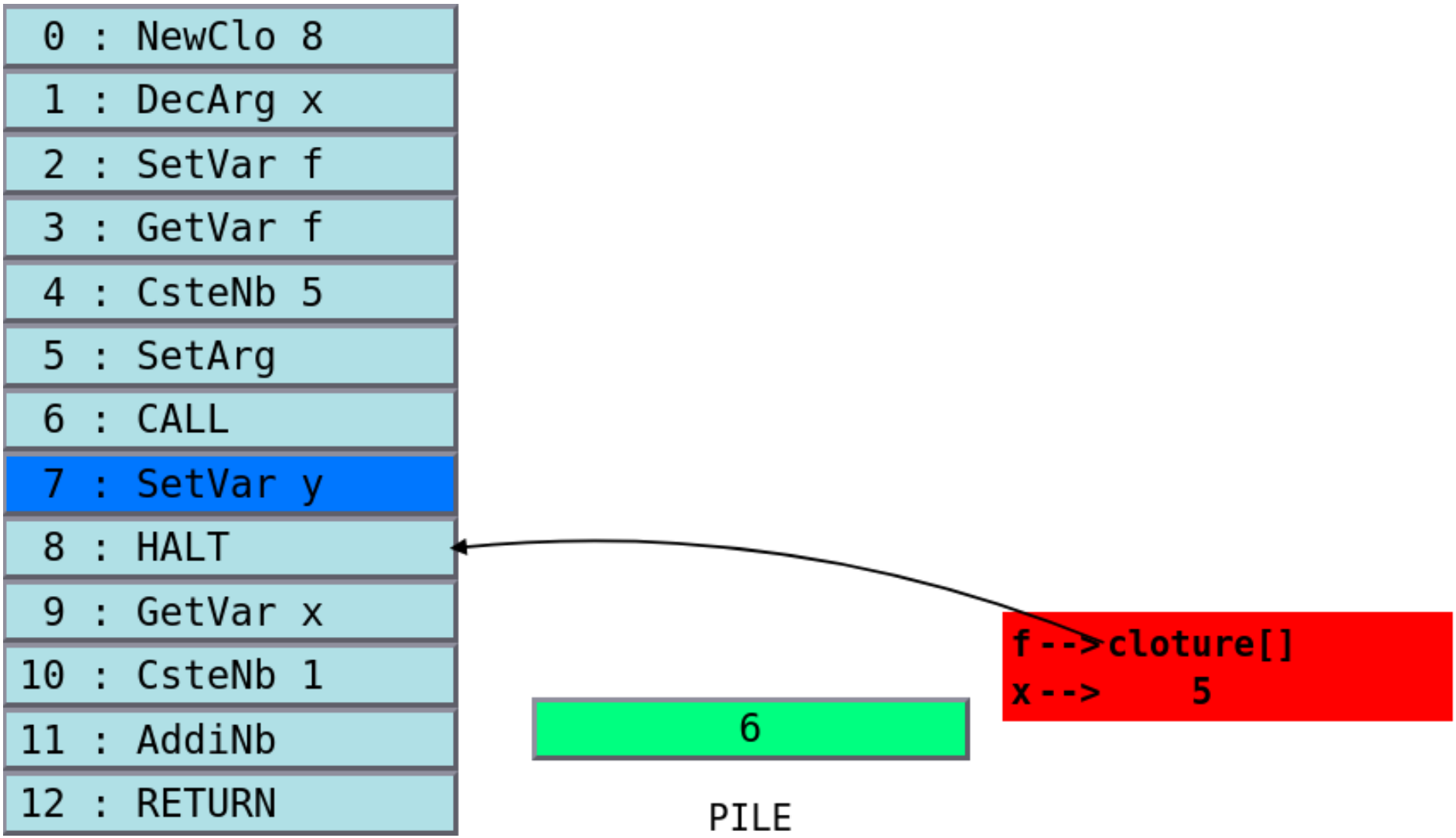
Example : y=f(5); function f(x) {return x+1;}



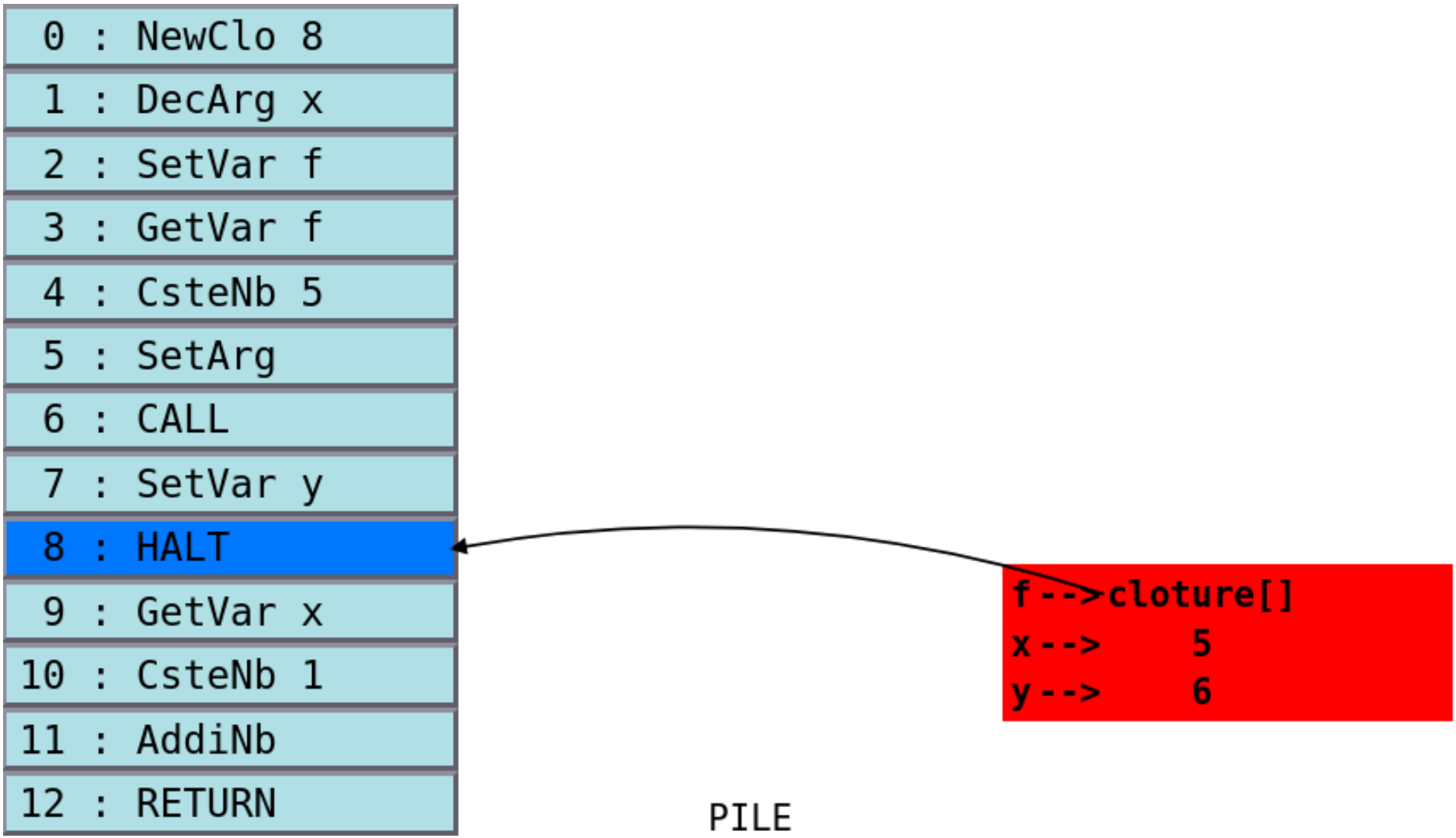
Example : y=f(5); function f(x) {return x+1;}



Example : y=f(5); function f(x) {return x+1;}



Example : y=f(5); function f(x) {return x+1;}

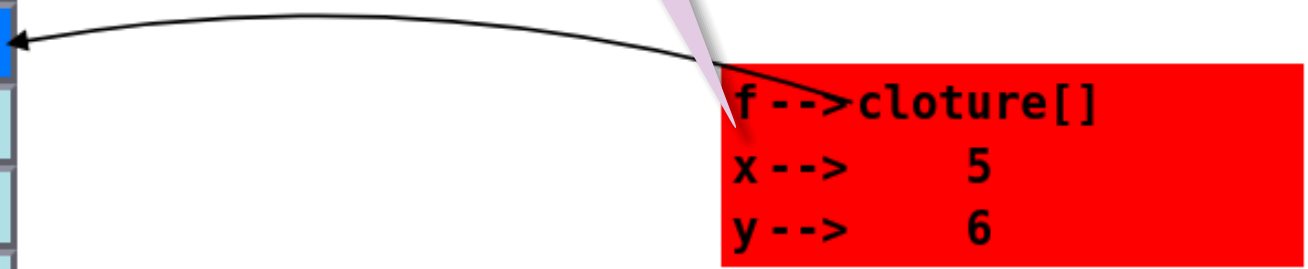


Example : y=f(5); function f(x) {return x+1;}

```
 0 : NewClo 8
 1 : DecArg x
 2 : SetVar f
 3 : GetVar f
 4 : CsteNb 5
 5 : SetArg
 6 : CALL
 7 : SetVar y
 8 : HALT
 9 : GetVar x
10 : CsteNb 1
11 : AddiNb
12 : RETURN
```

Why is x still in the contexte ??

we are not in f anymore, but x is local...



PILE

What makes a function call correct ?

Local variables are ephemeral

Local variables (and function arguments) should be local to the function.

Independant calls

Another call to the same function (sequential or recursive) should not interfer with local variable.

Muting global variables

A function can still see and modify global variables.

We definitly have to work with more contexts

Closures must refer to contexts

Closures and continuations contain pointers toward new contexts !

Managing local and global contexts

Solution 1: Push variables on the stack

More efficient but heavy and do not go well with higher order.

Solution 2: New context for each function call

Lighter assembly code, but one has to manage context inclusion.

Our solution: Chained list of hash tables

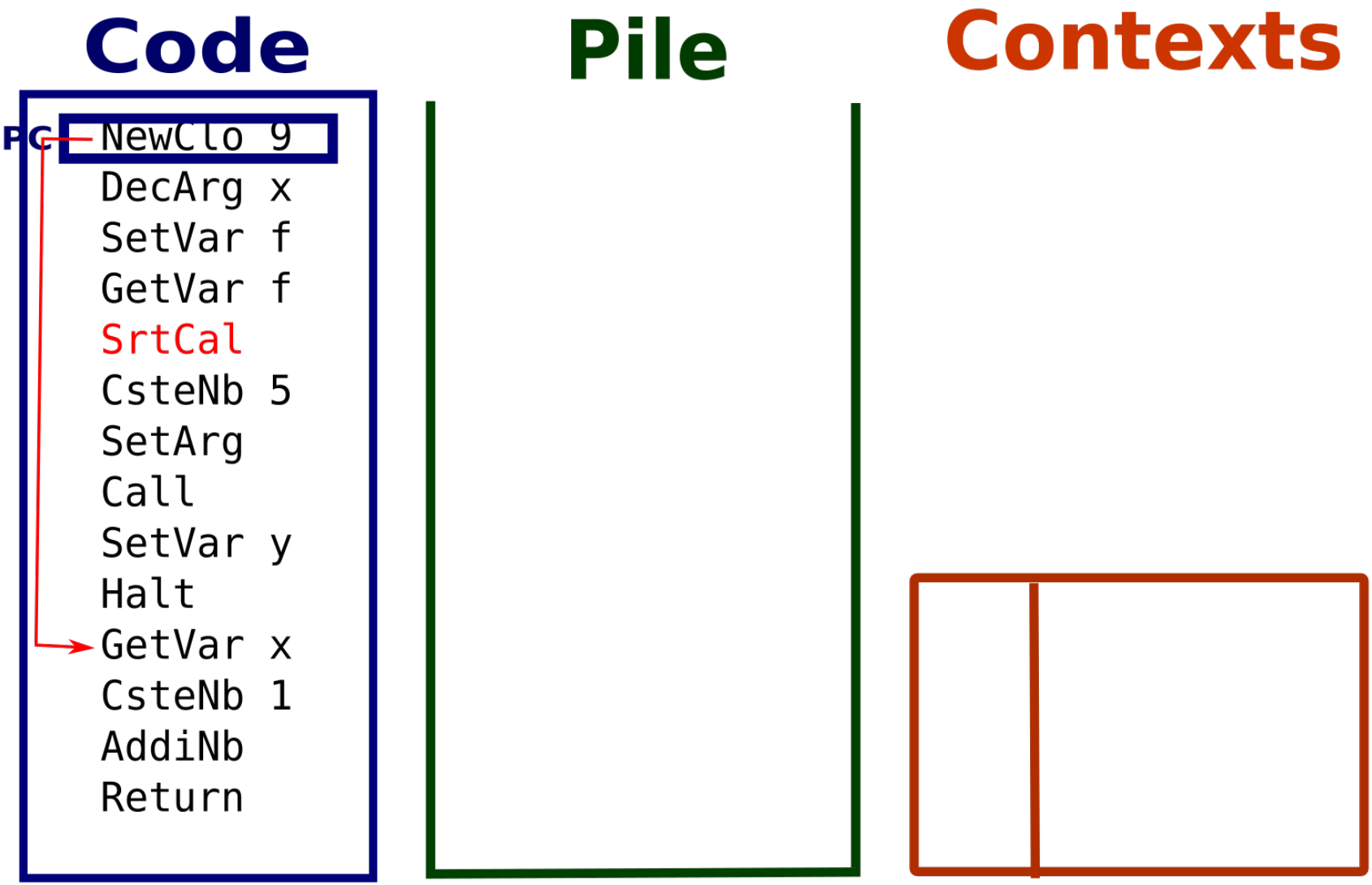
- Still a dictionary,
- easy sharing sub-contexts (copy a pointeur),
- as many locality level as we need !

A new instructions do manage locality :

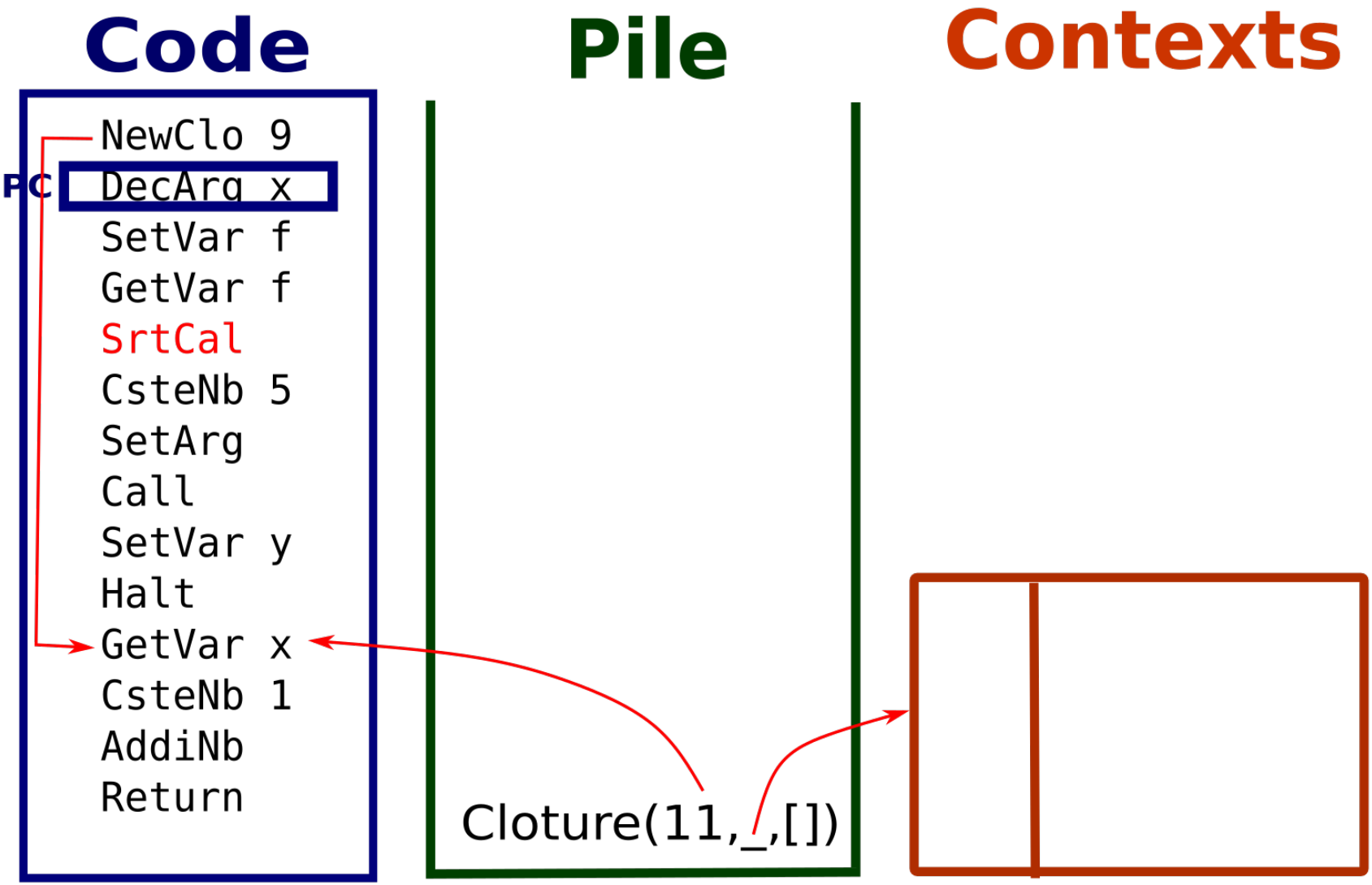
StrCall	`Pull.setContext(NewContext(CC))`	`l:stk`	`l:stk`

This extends the closure context with a new, “more local”, context.

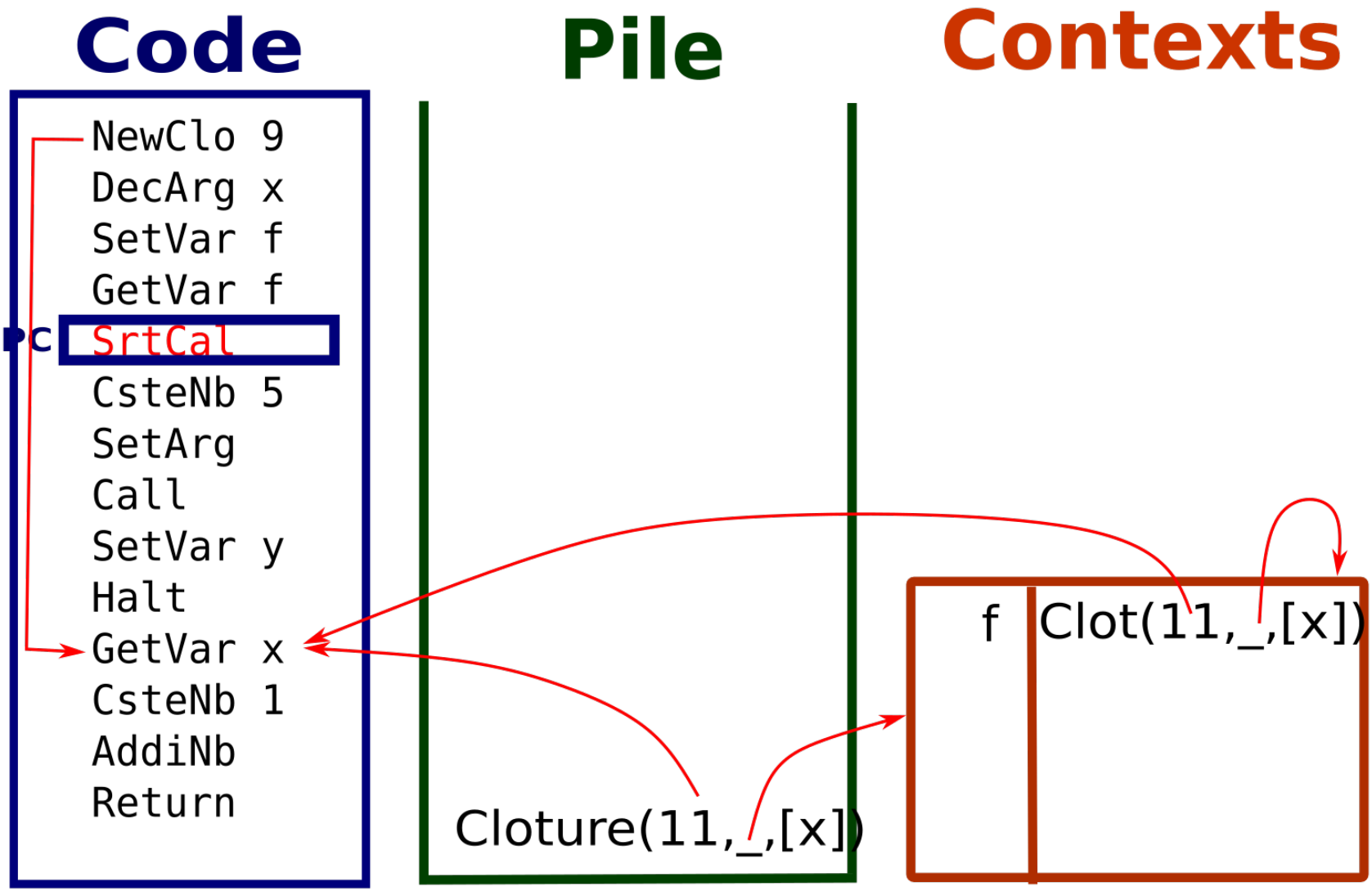
Correction : y=f(5); function f(x) {return x+1;}



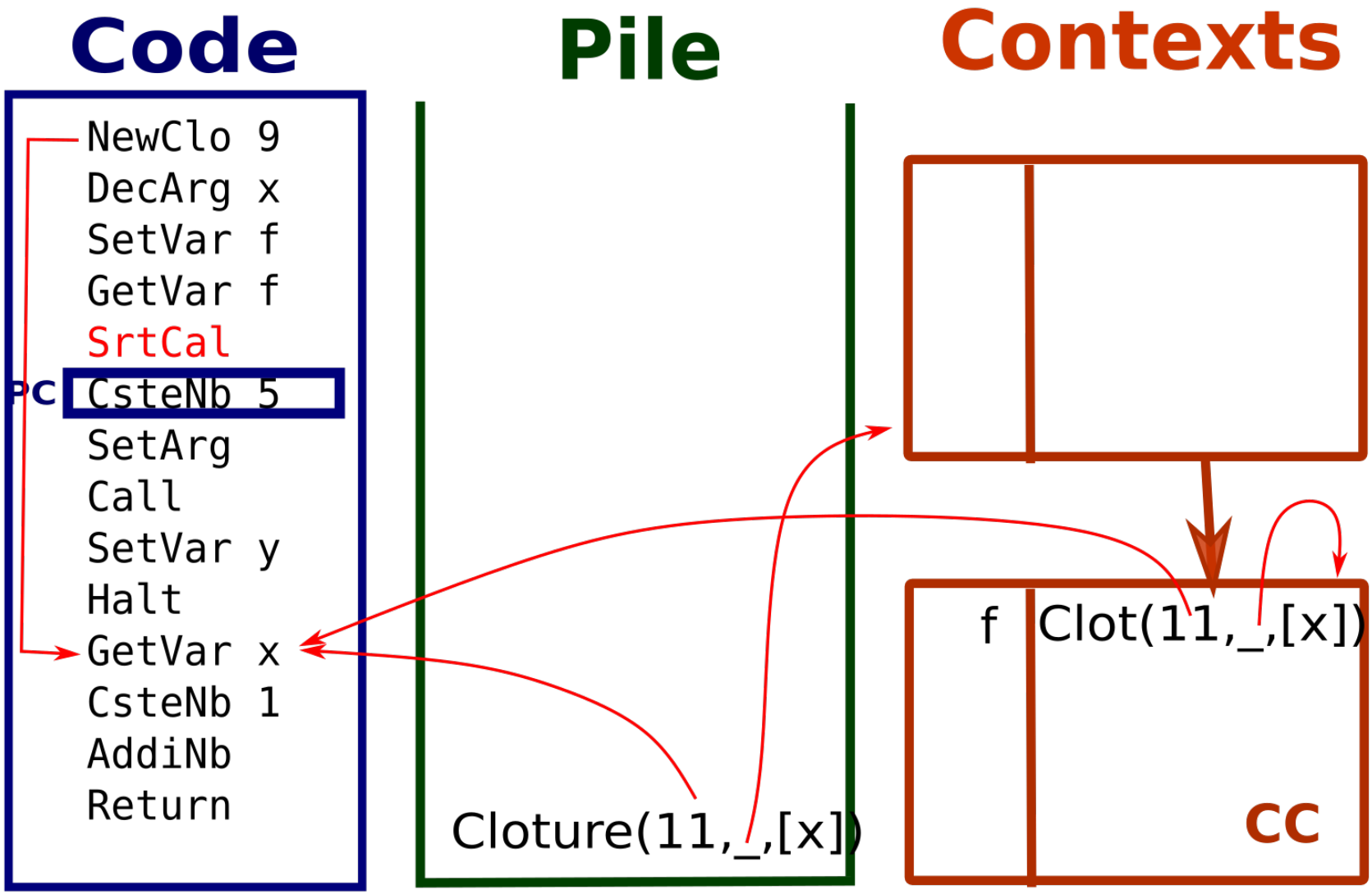
Correction : y=f(5); function f(x) {return x+1;}



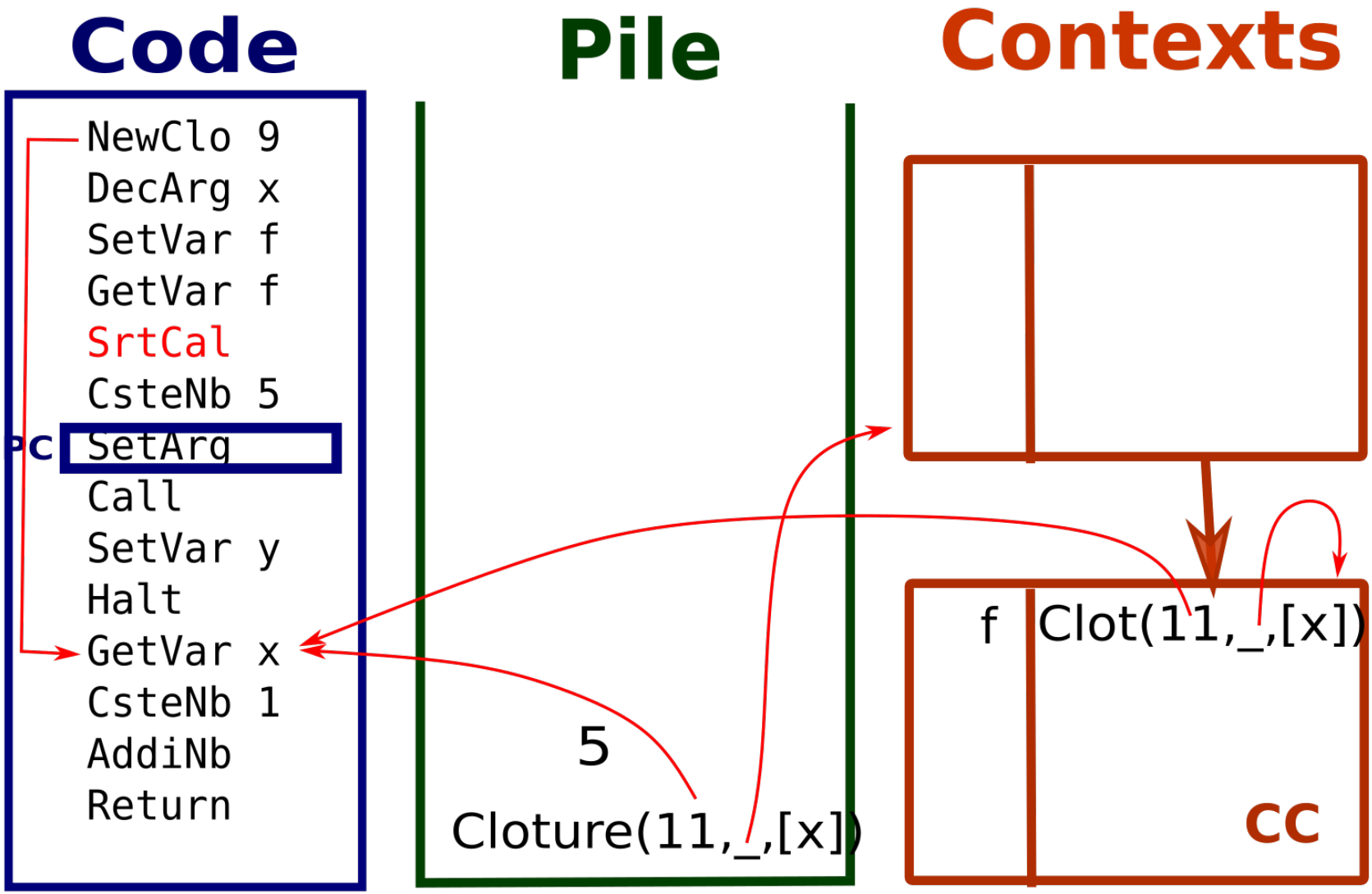
Correction : y=f(5); function f(x) {return x+1;}



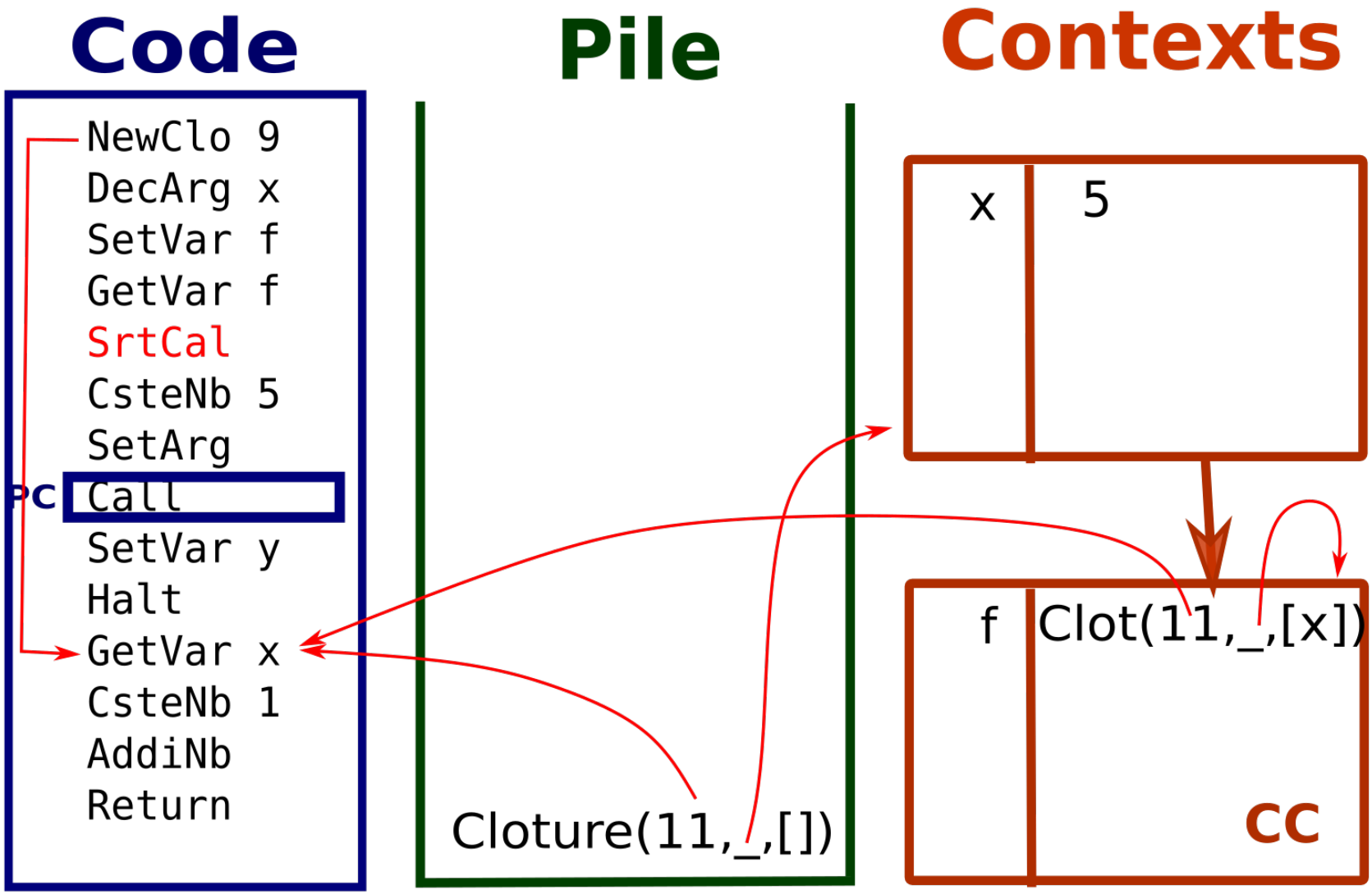
Correction : y=f(5); function f(x) {return x+1;}



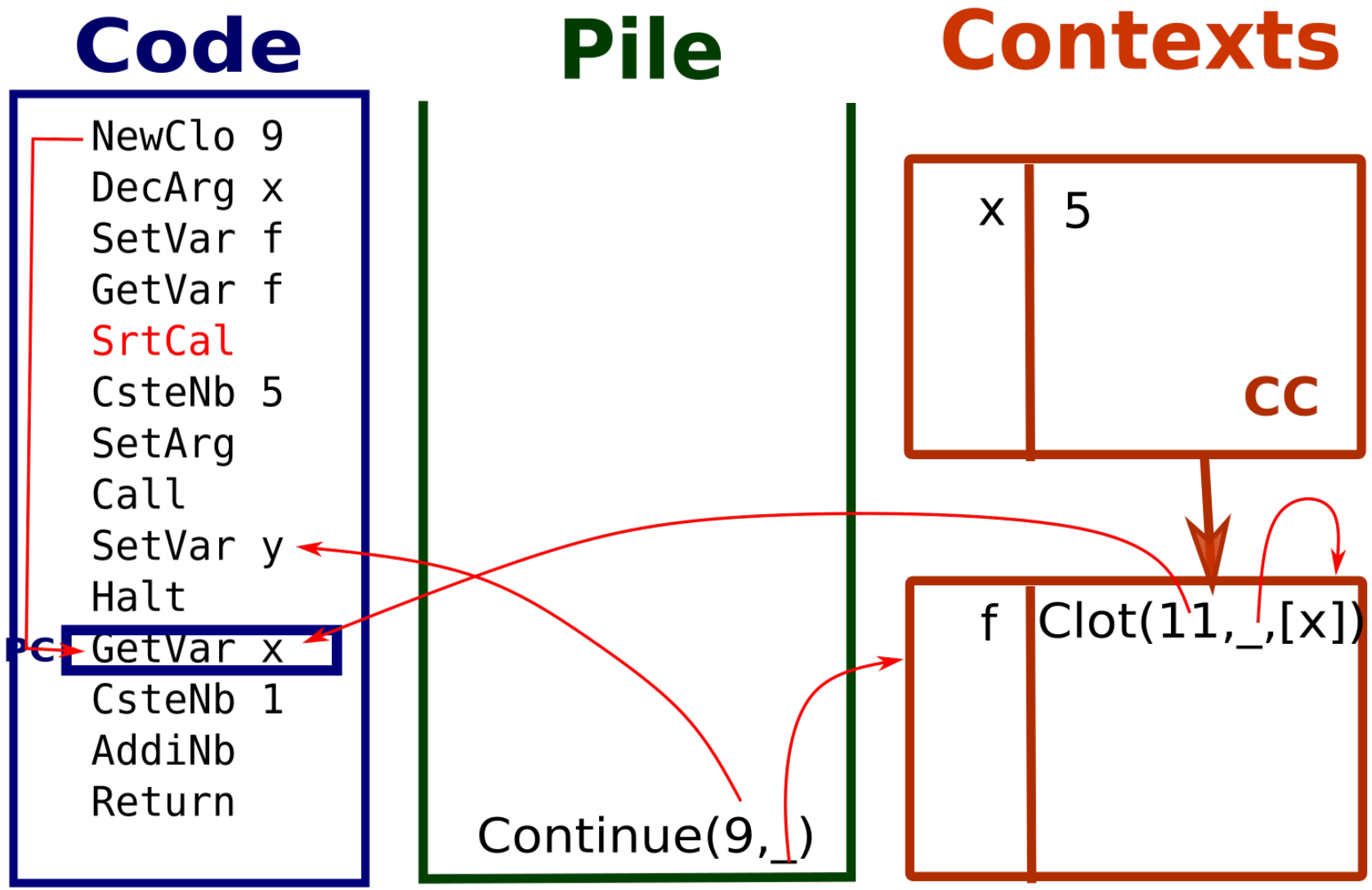
Correction : y=f(5); function f(x) {return x+1;}



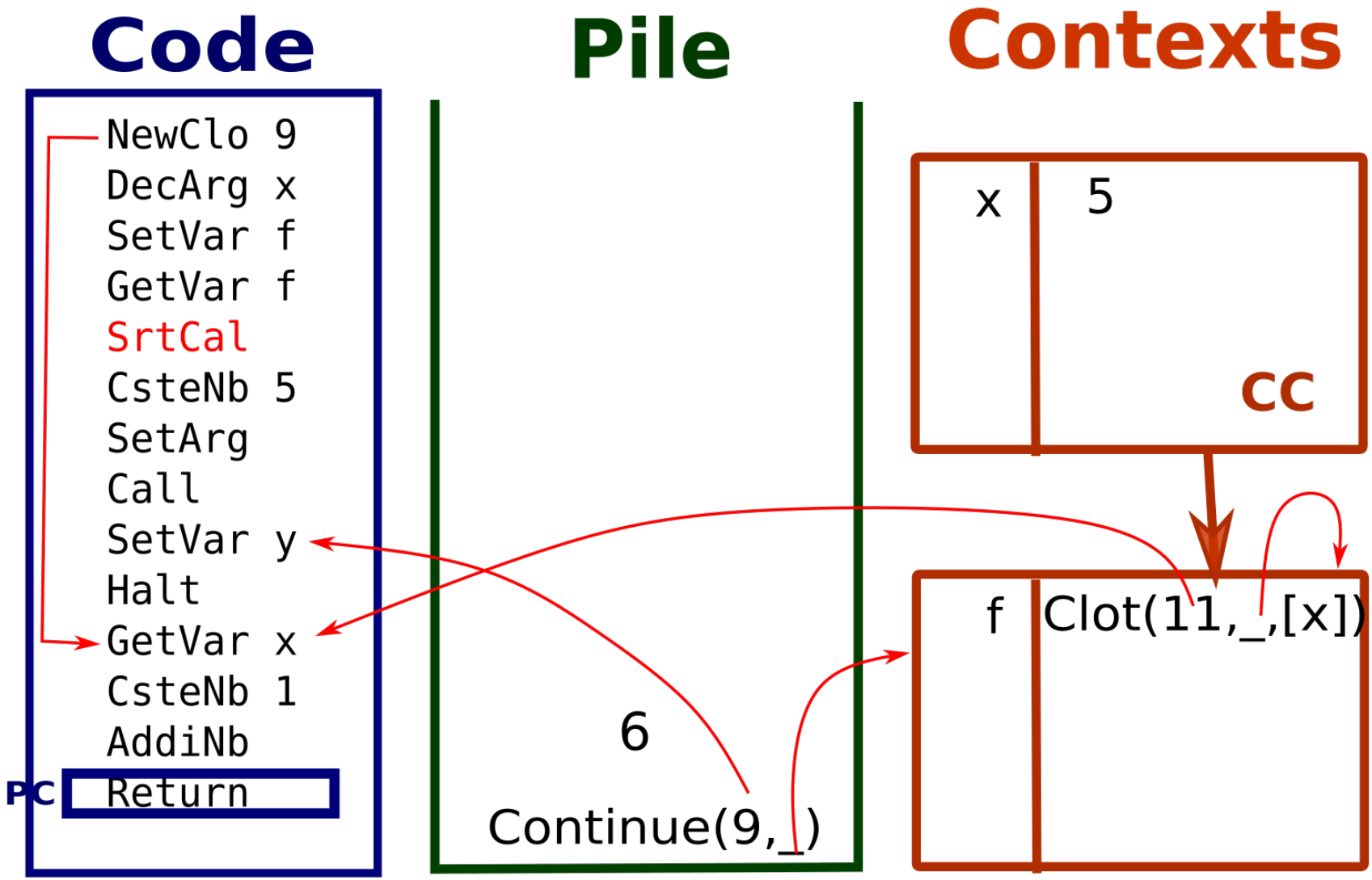
Correction : y=f(5); function f(x) {return x+1;}



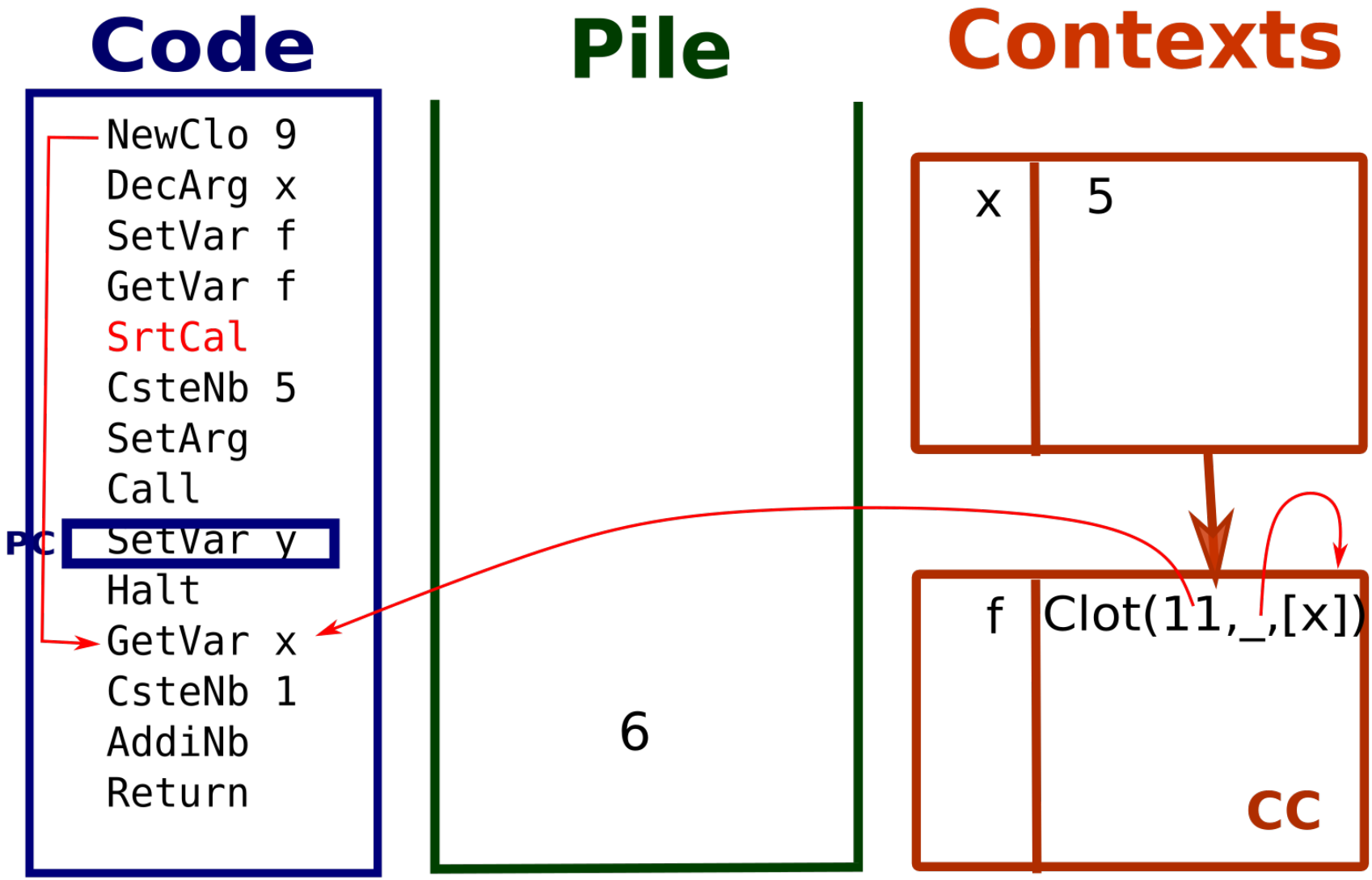
Correction : y=f(5); function f(x) {return x+1;}



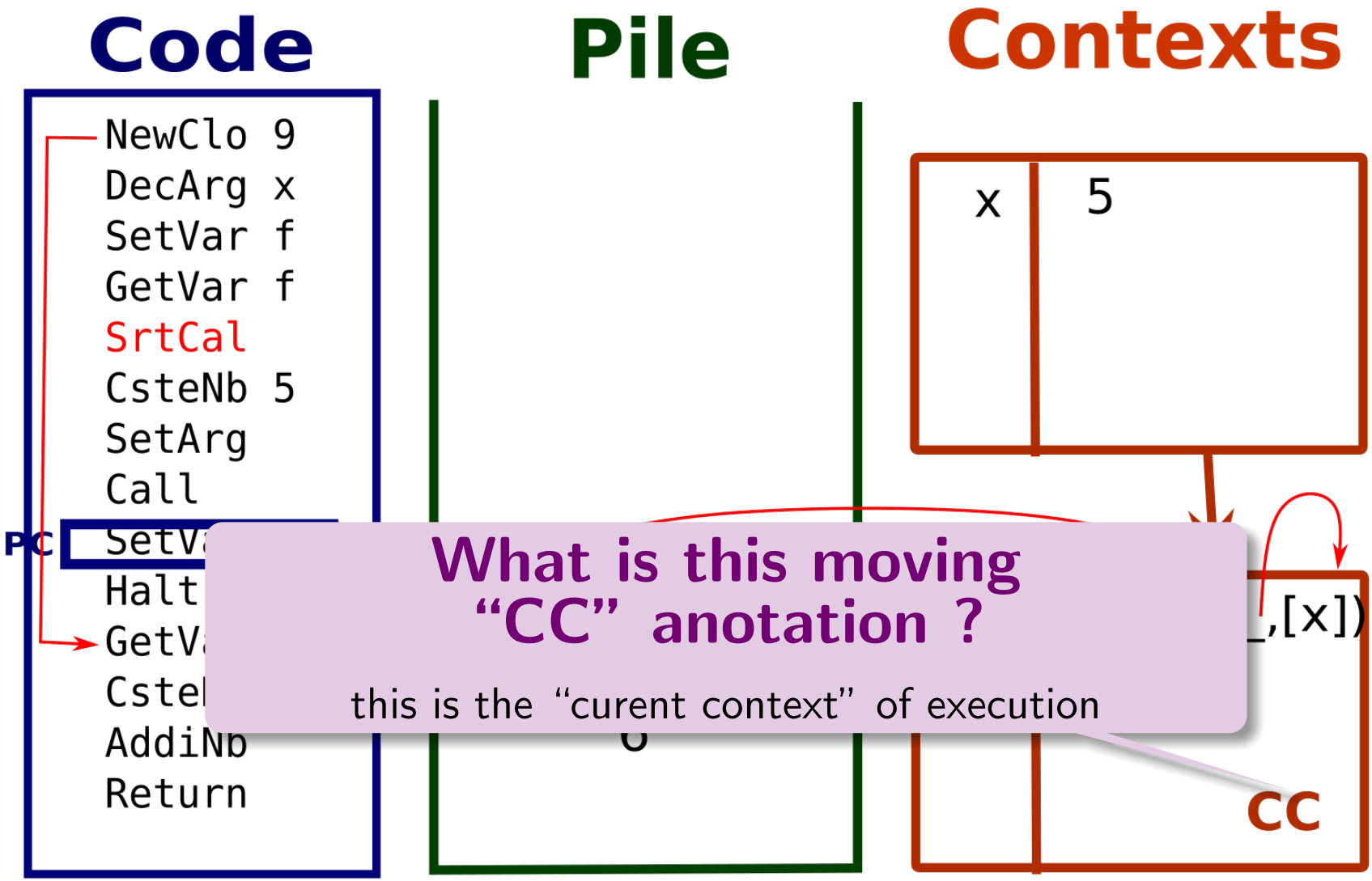
Correction : y=f(5); function f(x) {return x+1;}



Correction : y=f(5); function f(x) {return x+1;}



Correction : y=f(5); function f(x) {return x+1;}



Closures (bis)

A cloture contains

- a pointer toward the code of the function,
- a stack of names for variables,
- **a context**

NewClo off	`Push(NewCloture{ code = PC+off+1, cont = CC })`	`stk`	`l:stk`
StrCall	`Pull.setContext(NewContext(CC))`	`l:stk`	`l:stk`
DclArg name	`Pull.args.Push(name)`	`l:stk`	`l:stk`
SetArg	`Set(Pull.args.Pop, Pop)`	`v:l:stk`	`l:stk`

Continuation (bis)

A continuation contains

- a pointer toward the line that called the function,
- a context
- ... **(more later)**

Call	`clot := Pop;`
	`Push(NewContinuation`
	`{code := PC, cont := CC})`
	`PC := clot.code`
	`CC := clot.cont`
		`l:stk`	`t:stk`

Return	`res := Pop; c := Pop;`
	`PC := c.code`
	`CC := c.cont;`
	`Push(res);`
		`v:t:stk`	`v:stk`

Demonstration on the VM

Notice the action of the garbae collector (GC) that remove the context.

Higher order + mutability : explosive cocktail

Exercice : Find the result of the following program

```
g1 = f(1); g2 = f(0);
g1(g1(g2(g2(10))));

function f(x) {
  var y = x;
  return g;
  function g(z){
    y = y+z;
    return y;
  }
}
```

Higher order + mutability : explosive cocktail

Exercice : Find the result of the following program

```
g1 = f(1); g2 = f(0);
g1(g1(g2(g2(10))));

function f(x) {
  var y = x;
  return g;
  function g(z){
    y = y+z;
    return y;
  }
}
```

42

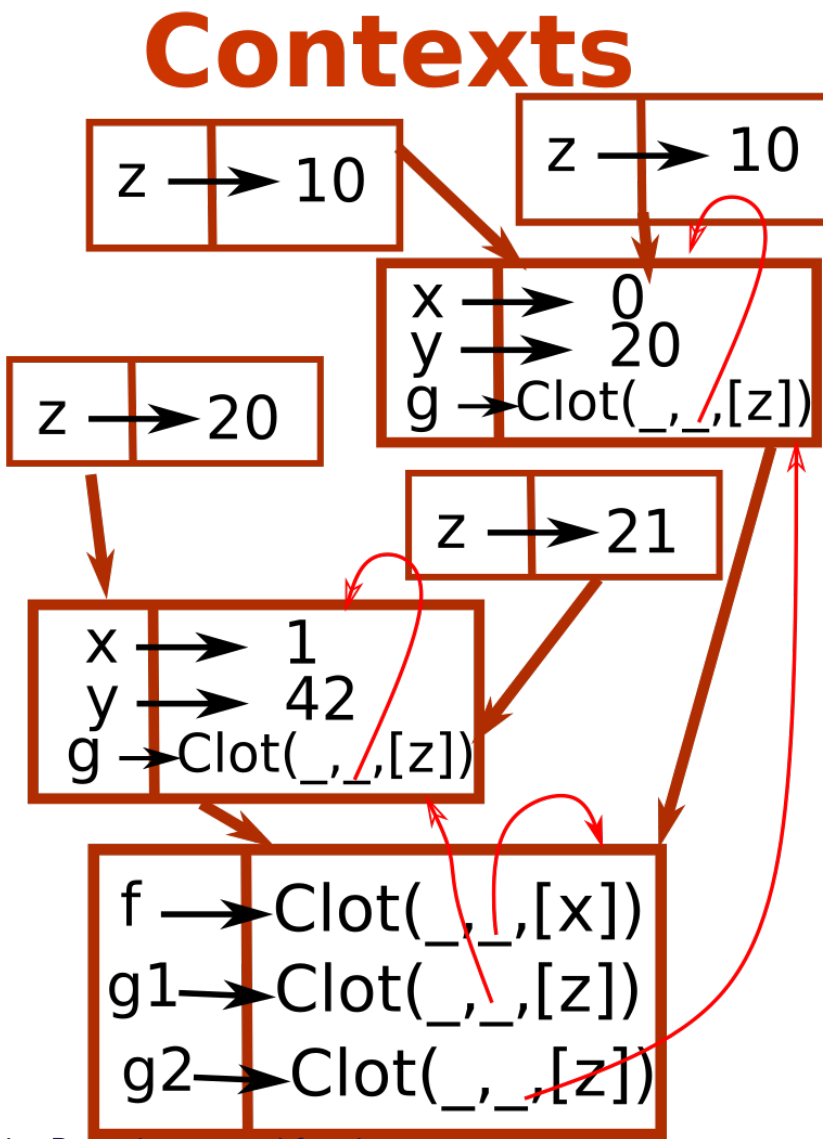
Higher order + mutability : explosive cocktail

Exercice : Find the result of the following program

```
g1 = f(1); g2 = f(0);
g1(g1(g2(g2(10))));

function f(x) {
  var y = x;
  return g;
  function g(z){
    y = y+z;
    return y;
  }
}
```

42



Code generation: three distinct parts

Function calls

- context creation
- arguments instanciation
- call

Function code

- local var. declarations
- code
- returns

Header

- closure creation
- argument decl.
- function decl.

Remainder: JS Hoisting creats headers

- the declarations are is hoisted as up as possible (header of the program/function).
- the function code is written in the end, after `Halt` (or remains were it is but with a `Jump` before)

Function call

$$[\![f(e_1, \ldots, e_n)]\!] = \begin{array}{|l} [\![f]\!] \\ \texttt{SrtCal} \\ [\![e_1]\!] \\ \texttt{SetArg} \\ \vdots \\ [\![e_n]\!] \\ \texttt{SetArg} \\ \texttt{Call} \end{array}$$

StrCal	`Pull.setContext(NewContext(CC))`	`l:stk`	`l:stk`
SetArg	`Set(Pull.args.Pop, Pop)`	`v:l:stk`	`l:stk`

Call	`clot = Pop;`
	`Push(NewContinue{code = PC, cont = CC})`
	`PC := clot.code; CC := clot.cont`
		`l:stk`	`t:stk`

Optimisation : Tail Recursion

If the call is in tail position (argument of a `return`), then we can use the instruction `TlCall` that do not push the continuation, so that the previous one will be use to return from the new function.

Function code

Where to write the function code ?

- at the end of the program: cleaner but more dificult
- remains where it is: do not forgot to add a `Jump`.

$$[\![\texttt{function } f(\ldots)\{\texttt{code}\}]\!] = \begin{array}{|l} \\ \textit{label\_f} \\ \\ \end{array} \begin{array}{|l} \texttt{Jump} \\ [\![\texttt{code}]\!] \\ \\ \end{array}$$

$$[\![\texttt{return } e;]\!] = \begin{array}{|c} [\![e]\!] \\ \texttt{Return} \end{array}$$

Return	`res = Pop; continu = Pop;`
	`PC := continu.code; CC := continu.cont;`
	`Push(res);`
		`v:t:stk`	`v:stk`

Header(s)

Where to write the header(s) ?

At the begining of the program if the function is global,
at the begining of the current function otherwise.

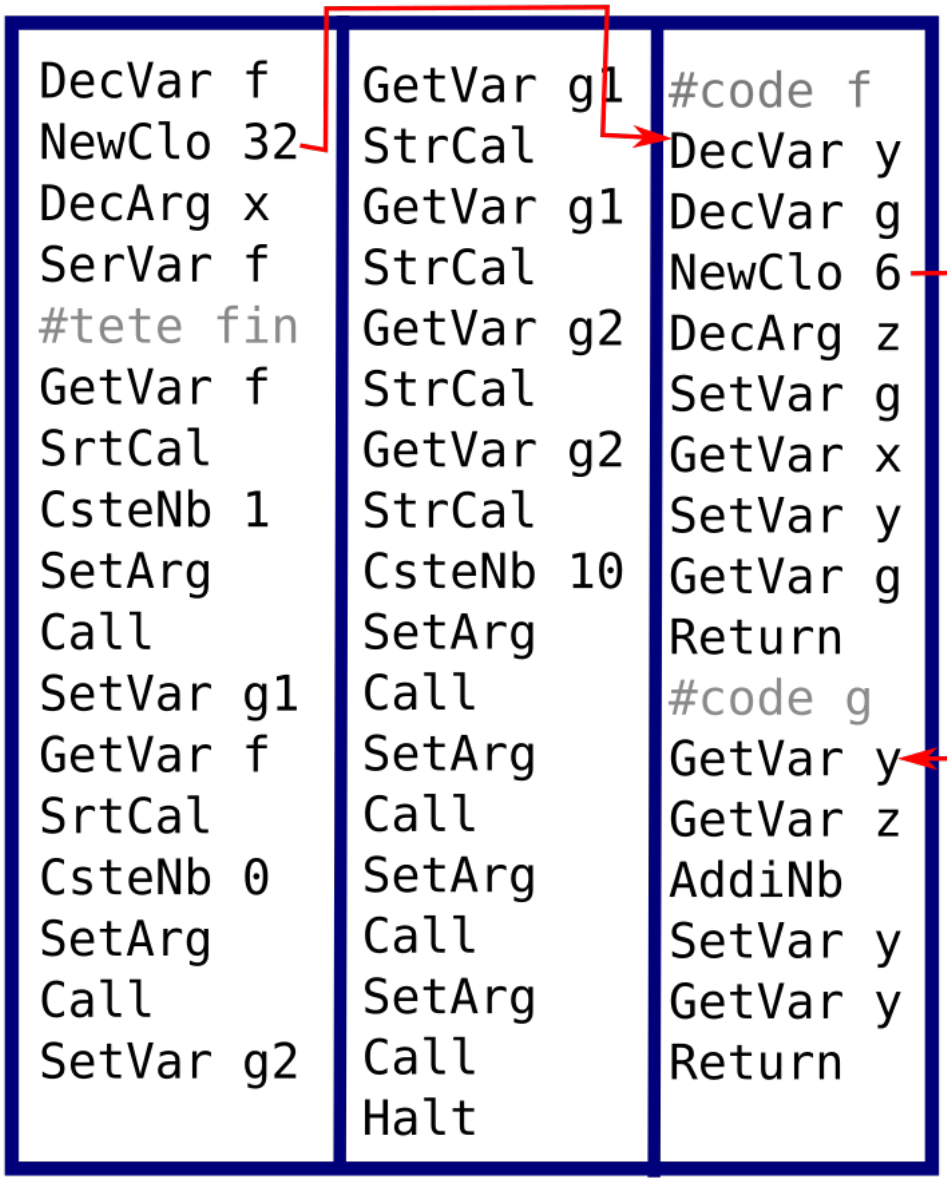
$$[\![\texttt{function } f(x_1 \ldots, x_n)\{\ldots\}]\!] = \left|\begin{array}{l} \texttt{DecVar } f \\ \texttt{NewClo label\_f} \\ \texttt{DecArg } x_1 \\ \quad \vdots \\ \texttt{DecArg } x_n \\ \texttt{SetVar } f \end{array}\right.$$

NewClo off	Push(NewCloture{ code = PC+off+1, cont = CC})	stk	l:stk
DclArg n	Pull.args.Push(n)	l:stk	l:stk

Example

```
g1 = f(1); g2 = f(0);
g1(g1(g2(g2(10))));

function f(x) {
    var y = x;
    return g;
    function g(z){
        y = y+z;
        return y;
    }
}
```



Example

Global header

declaration of f(x)

```
g1 = f(1); g2 = f(0);
g1(g1(g2(g2(10))));

function f(x) {
    var y = x;
    return g;
    function g(z){
        y = y+z;
        return y;
    }
}
```

```
DecVar f
NewClo 32
DecArg x
SerVar f
#tete fin
GetVar f
SrtCal
CsteNb 1
SetArg
Call
SetVar g1
GetVar f
SrtCal
CsteNb 0
SetArg
Call
SetVar g2
```

```
GetVar g1
StrCal
GetVar g1
StrCal
GetVar g2
StrCal
GetVar g2
StrCal
CsteNb 10
SetArg
Call
SetArg
Call
SetArg
Call
SetArg
Call
Halt
```

```
#code f
DecVar y
DecVar g
NewClo 6
DecArg z
SetVar g
GetVar x
SetVar y
GetVar g
Return
#code g
GetVar y
GetVar z
AddiNb
SetVar y
GetVar y
Return
```

Example

Code of f

```
g1 = f(1); g2 = f(0);
g1(g1(g2(g2(10))));

function f(x) {
    var y = x;
    return g;
    function g(z){
        y = y+z;
        return y;
    }
}
```

```
DecVar f
NewClo 32
DecArg x
SerVar f
#tete fin
GetVar f
SrtCal
CsteNb 1
SetArg
Call
SetVar g1
GetVar f
SrtCal
CsteNb 0
SetArg
Call
SetVar g2
```

```
GetVar g1
StrCal
GetVar g1
StrCal
GetVar g2
StrCal
GetVar g2
StrCal
CsteNb 10
SetArg
Call
SetArg
Call
SetArg
Call
SetArg
Call
Halt
```

```
#code f
DecVar y
DecVar g
NewClo 6
DecArg z
SetVar g
GetVar x
SetVar y
GetVar g
Return
#code g
GetVar y
GetVar z
AddiNb
SetVar y
GetVar y
Return
```

Example

Header of f

decl. of y and g(z)

```
g1 = f(1); g2 = f(0);
g1(g1(g2(g2(10))));

function f(x) {
    var y = x;
    return g;
    function g(z){
        y = y+z;
        return y;
    }
}
```

```
DecVar f
NewClo 32
DecArg x
SerVar f
#tete fin
GetVar f
SrtCal
CsteNb 1
SetArg
Call
SetVar g1
GetVar f
SrtCal
CsteNb 0
SetArg
Call
SetVar g2
```

```
GetVar g1
StrCal
GetVar g1
StrCal
GetVar g2
StrCal
GetVar g2
StrCal
CsteNb 10
SetArg
Call
SetArg
Call
SetArg
Call
SetArg
Call
Halt
```

```
#code f
DecVar y
DecVar g
NewClo 6
DecArg z
SetVar g
GetVar x
SetVar y
GetVar g
Return
#code g
GetVar y
GetVar z
AddiNb
SetVar y
GetVar y
Return
```

Interlude : entrailes of `GetVar` and `SetVar`

Algorithm of `Getvar x`

Go down the contexts and return the value first found `x`
(or `undefined` if x not found)

```
GetVar x := CC.get(x)
c.get(x) := if (c.head.is_in(x)) return c.head.get(x);
            else if (c.is_bottom) return undefined;
            else return c.tail.get(x);
```

Algorithm of `Setvar x`

Go down the context and insert in the first found `x`
(or in the global context if x not found)

```
SetVar n x := CC.set(n,x)
c.set(n,x) := if (c.head.is_in(x)) c.head.set(n,x);
              else if (c.is_bottom) c.head.set(n,x);
              else c.tail.set(x);
```