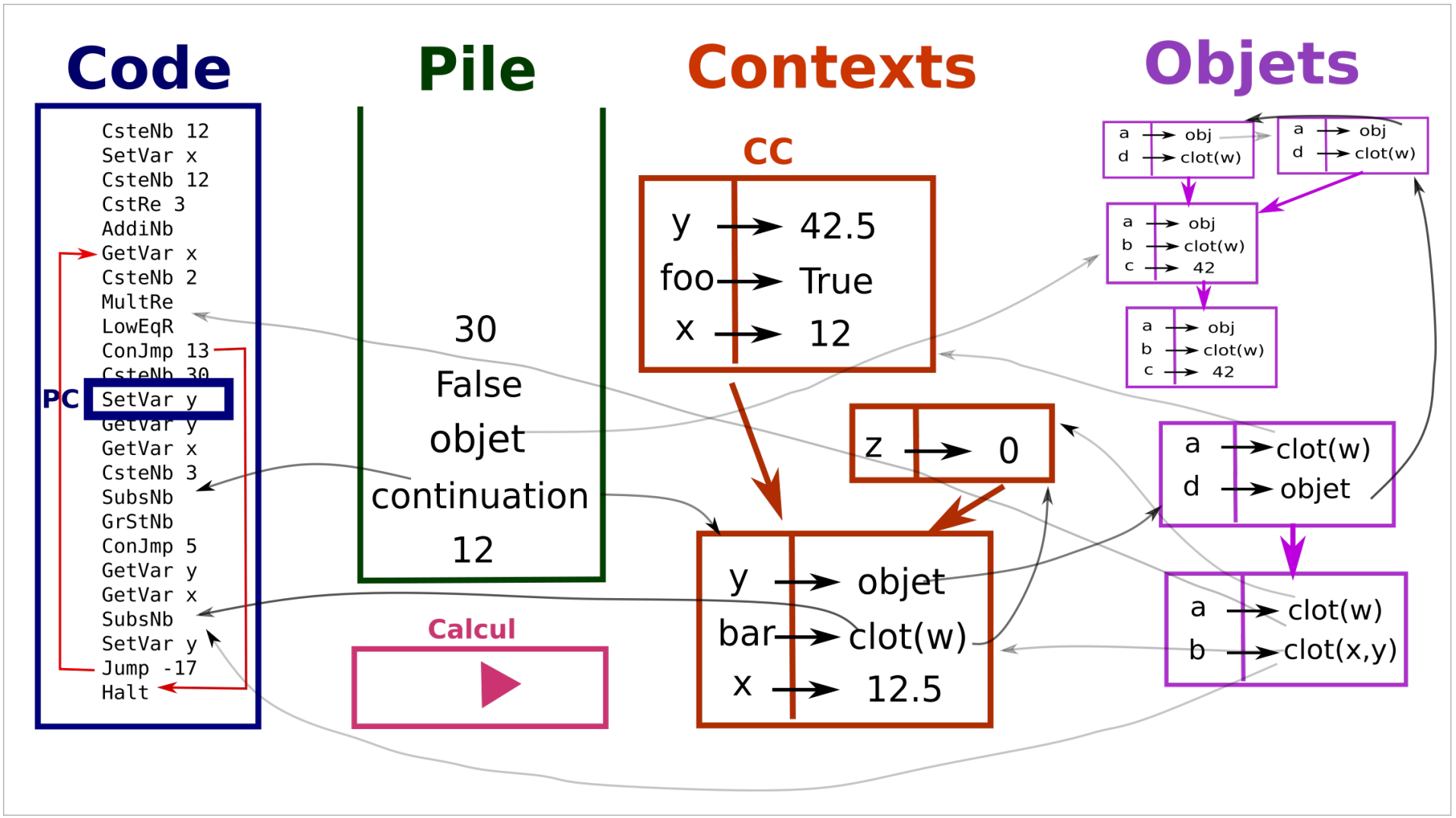
# CM 7 de Compilation : Mini-JS machine Arithm., variables and loops

Flavien BREUVART

March 11, 2026

# Internal structure of our Mini-JS machine

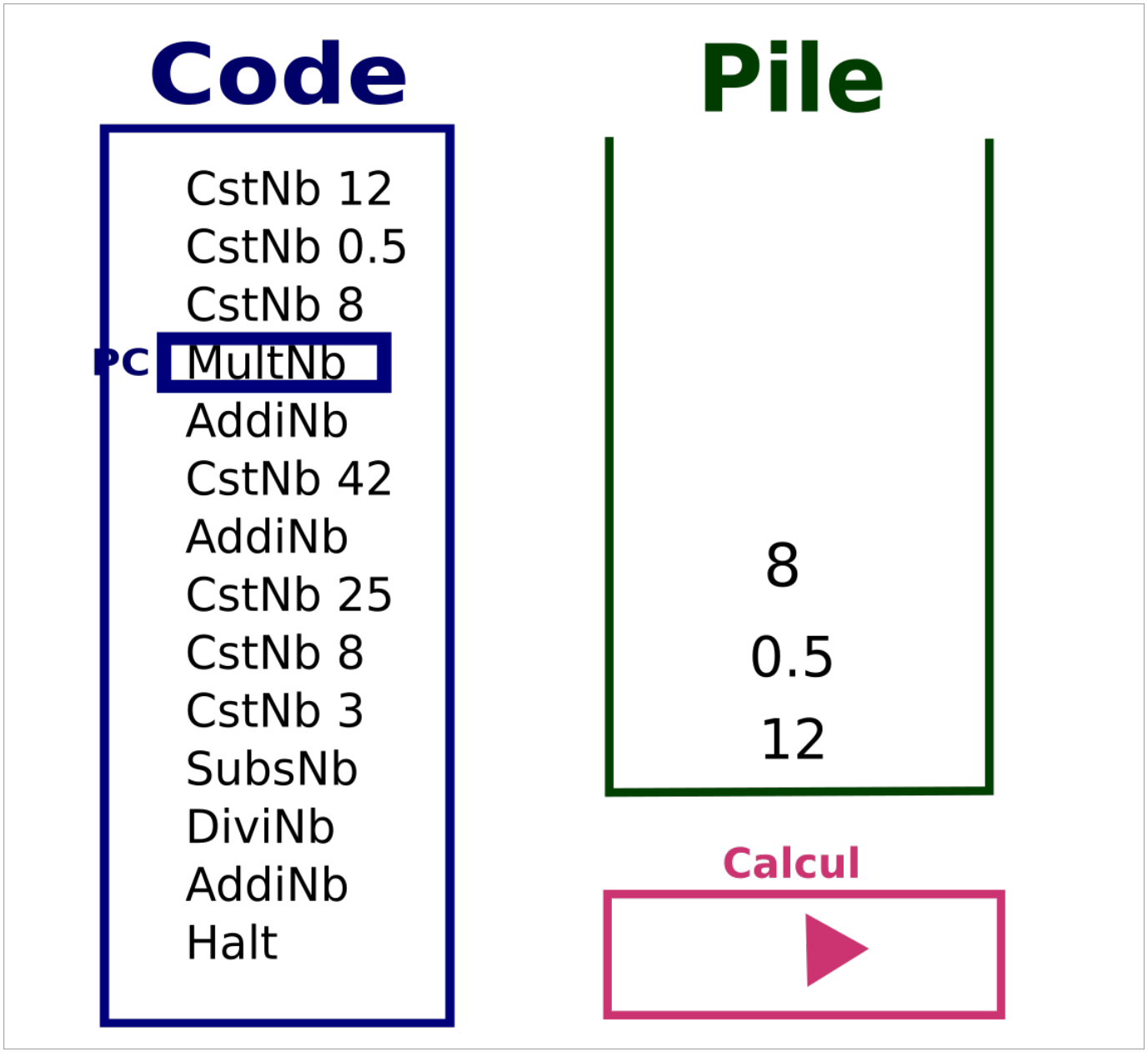


# Access to our Mini-JS machine

# DEMO

**https://lipn.univ-paris13.fr/∼breuvart/compilation/interpreteurJS/**

# For now : Code, Stacks and arithmetic op.



# Code area for different machines

The executed code is available in a specific memory area

## Separated from memory ? Writing rights ?

Virtual machine generally allocates a separated, write only,
memory area for the code.

## "Auto-compiling" code

On physical machine, it is possible to write binaries that modify themselves. Used for code obfuscation (virus and copyrighted programs...).

```
 0 : DecVar f
 1 : NewClo 32
 2 : DecArg x
 3 : SetVar f
 4 : GetVar f
 5 : StCall
 6 : CsteNb 1
 7 : SetArg
 8 : CALL
 9 : SetVar g1
10 : GetVar f
11 : StCall
12 : CsteNb 0
13 : SetArg
14 : CALL
15 : SetVar g2
16 : GetVar g1
17 : StCall
18 : GetVar g1
19 : StCall
20 : GetVar g2
```

# Code area in our VM

## Instructions

Name of the instruction plus 0, 1 or 2 arguments,

- arguments : float, string, boolean, offset or identifier,
- name : CamelCase notation (lexer flexible on the choice of lower case)

Example :
**CsteNb** 4.2

## Code area

Instructions tabular:

- read only,
- filled by input code,
- 1st line = starting point,
- last line: instruction **Halt**

[**CsteNb** 8.5, **CsteNb** 33.5, **AddiNb**, **Halt**]

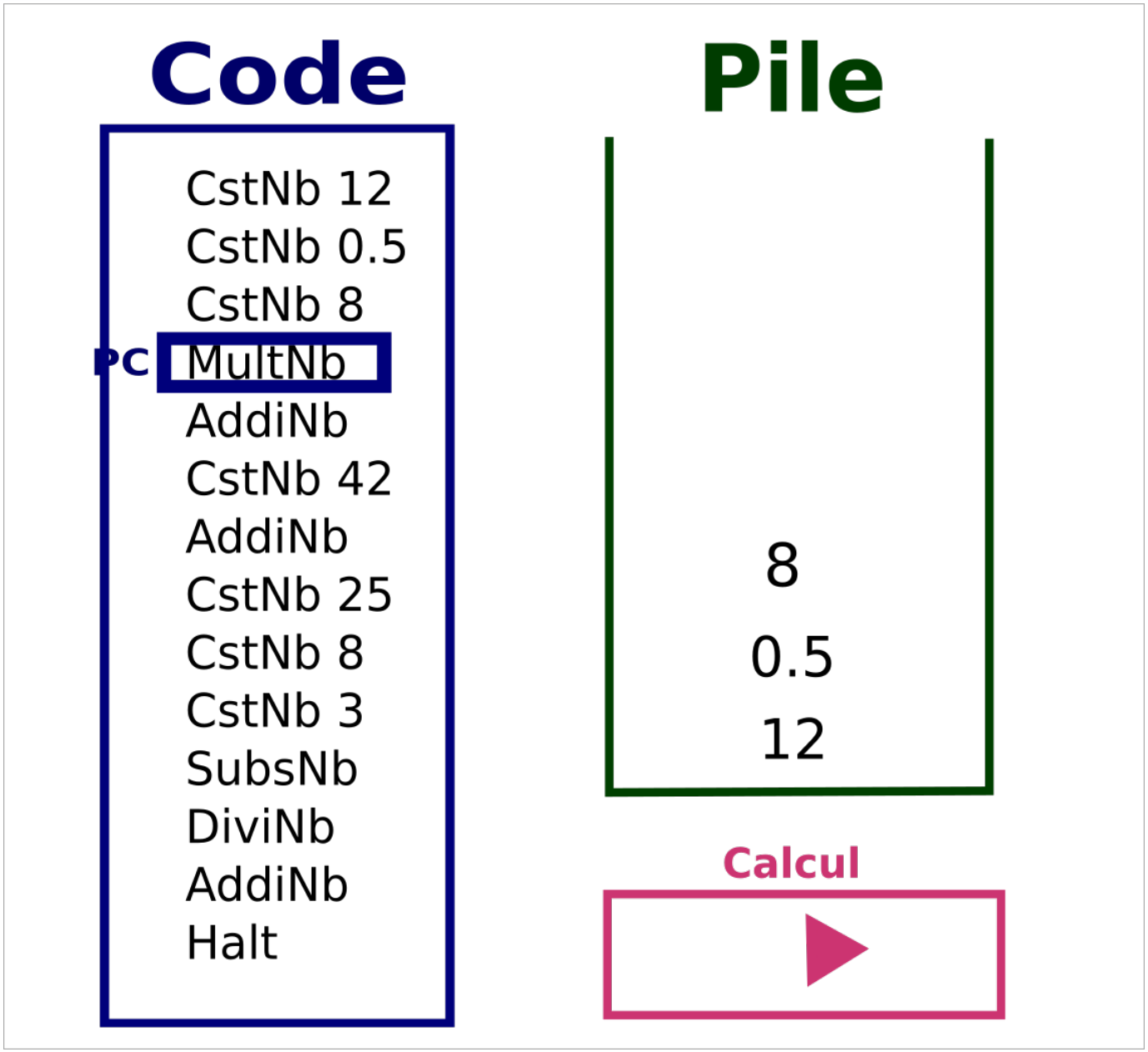
## Vertical representation of our code

for readability.

## PC : "Program Counter" (or "Pointeur de Code")

Points the instruction to execute.

# For now : Code, Stacks and arithmetic op.



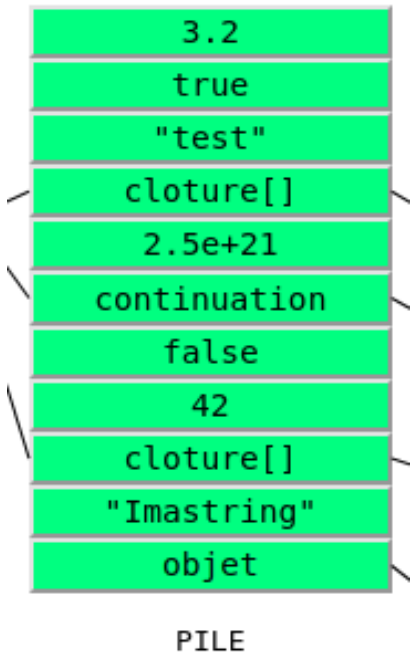
# Datas / Values

## Data types

Physical machines: bits representing pointer+ints+floats +...
Virtual machines : dynamic types, pre-constructed structures

## Some pre-constructed structures

- base types (int, floats, etc...)
- objects
- vectors/string
- closure/continuation (cf dedicated course)
- ...



# Registers vs Stacks

## Target assembly instructions

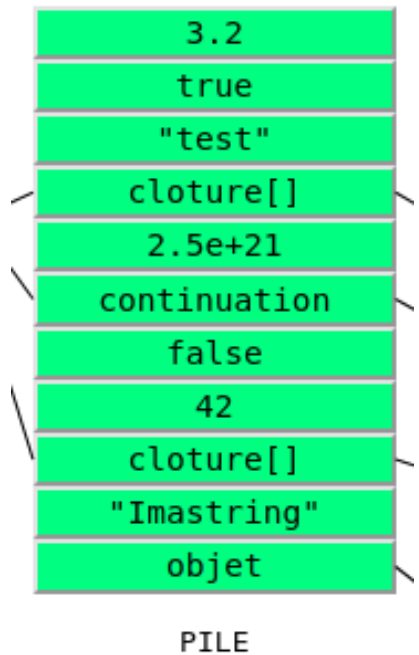
Physical machines : instructions operates on registers
Virtual machines : instructions generally operates on the stack

## Difficult to completely bypass stacks

A floating sum with a balanced AST of height $h$ needs $h$ registers.

## Registers are kind of "level 0" caches

Faster than the stack which is a situated in a level 1 cache




Only introduce additional compilation issues.
Remark: this is also a way to avoid too much redundancy with L2 architecture course

# (Execution) Stack/Stack/LIFO

## The execution stack is unic

(in a given thread/process)

## Filled by values

- Numbers (= floats !)
- Booleans
- Undefined
- Objects
- Closures (= functions)
- Continuations

## Execution stack

Stack of values handled by pseudo-code :

- `Push(x)` (*empiler* `x`),
- `Pop` : (*dépiler*),
- `Peak` : (*regarder sommet*).

Starts empty

## Pseudo-code

We use a specific decode to describe instructions.

No register, the stack is used for everything...

# Stack handling

| Instruction | meaning | semantics | stack before | after |
|---|---|---|---|---|
| **Noop** | do nothing | ; | `stack` | `stack` |
| **CsteNb x** | push the number $x$ | `Push(x)` | `stack` | `n:stack` |
| **Drop** | pop and forget | `Pop` | `v:stack` | `stack` |
| **Copy** | copy the top | `Push(Peak)` | `v:stack` | `v:v:stack` |

## Types before/after

The stacks is multi-varied, we can only describe the types of values at the top of it; $n, v, s...$ plays two role : that of types and meta-variable

- "`n:stk`" denotes a stack starting by an integer $n$,
- "`v:stk`" denotes a stack starting by any value $v$,

## **CsteNb 42** on stack $[6.6, 75]$ give $[42, 6.6, 75]$

**CsteNb x** push the numerical constant `x` on the stack.

# Floating point arithmetic

| Instruction | semantics | stack before | after |
|---|---|---|---|
| **AddiNb** | $\texttt{Push}(\texttt{Pop} +_f \texttt{Pop});$ | $n_1{:}n_2{:}stk$ | $n_3{:}stk$ |
| **SubsNb** | $\texttt{Push}(\texttt{Pop} -_f \texttt{Pop});$ | $n_1{:}n_2{:}stk$ | $n_3{:}stk$ |
| **MultNb** | $\texttt{Push}(\texttt{Pop} *_f \texttt{Pop});$ | $n_1{:}n_2{:}stk$ | $n_3{:}stk$ |
| **DiviNb** | $\texttt{Push}(\texttt{Pop}/_f\texttt{Pop});$ | $n_1{:}n_2{:}stk$ | $n_3{:}stk$ |
| **NegaNb** | $\texttt{Push}((-1) *_f \texttt{Pop});$ | $n_1{:}stk$ | $n_3{:}stk$ |

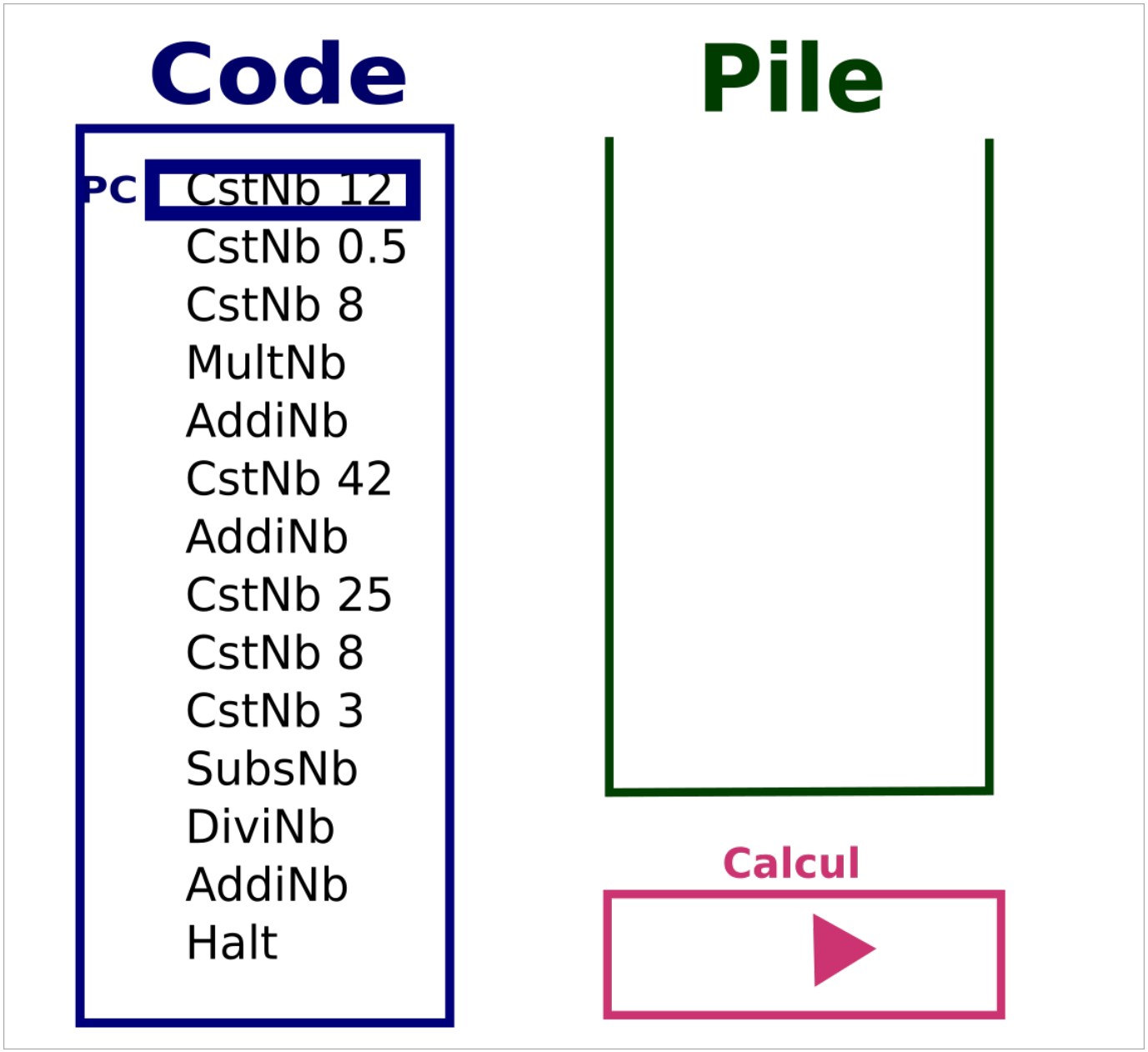
**Warning: our pseudo-code use a right-to-left execution**

Unusual convention that us well adapted to stack manipulations.

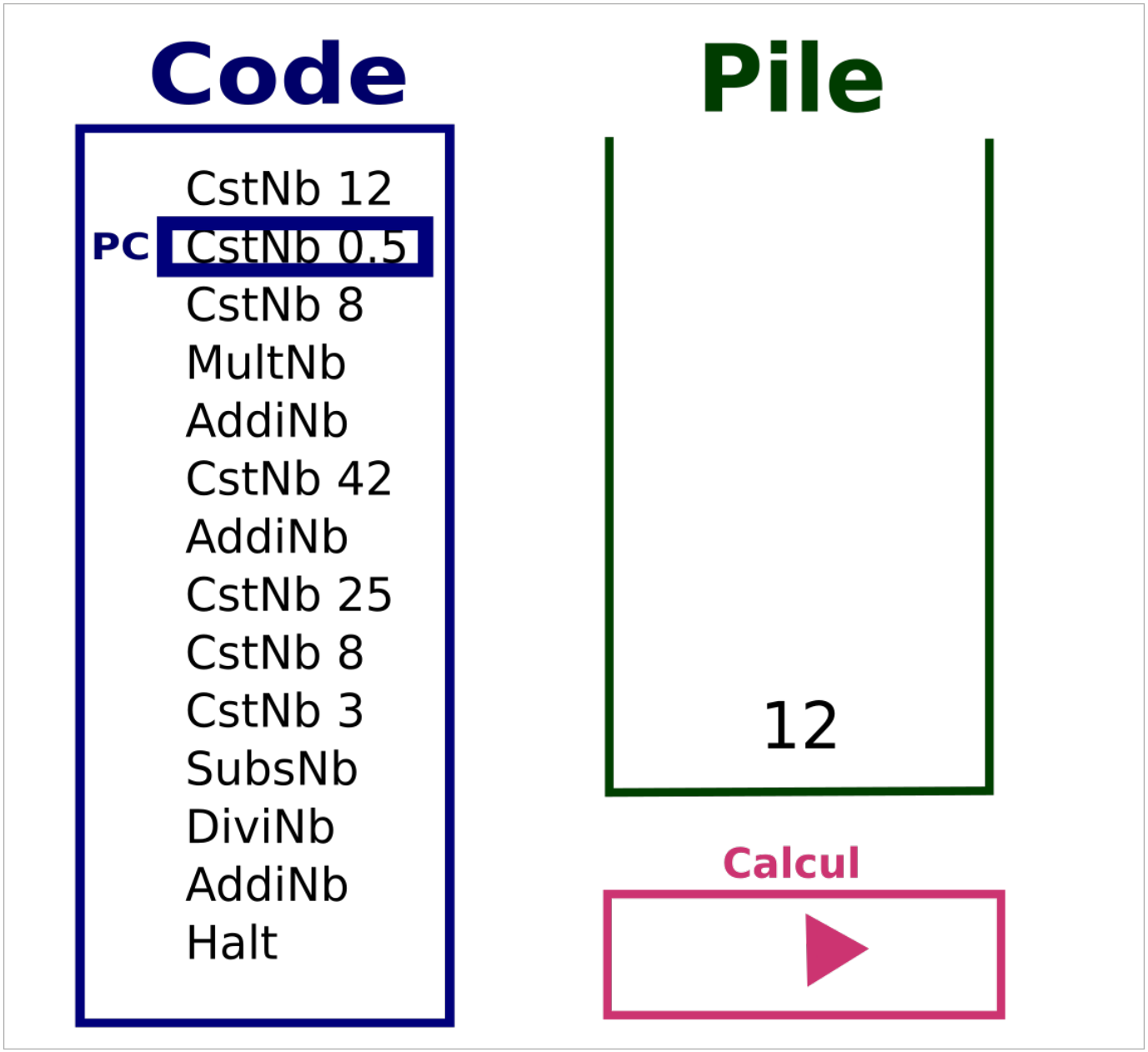
**SubsNb** on $[2.5, 25, 6.6, 75]$ results in $[22.5, 6.6, 75]$

**SubsNb** subtract the 1st value of the stack from the second
(read it right-to-left !)

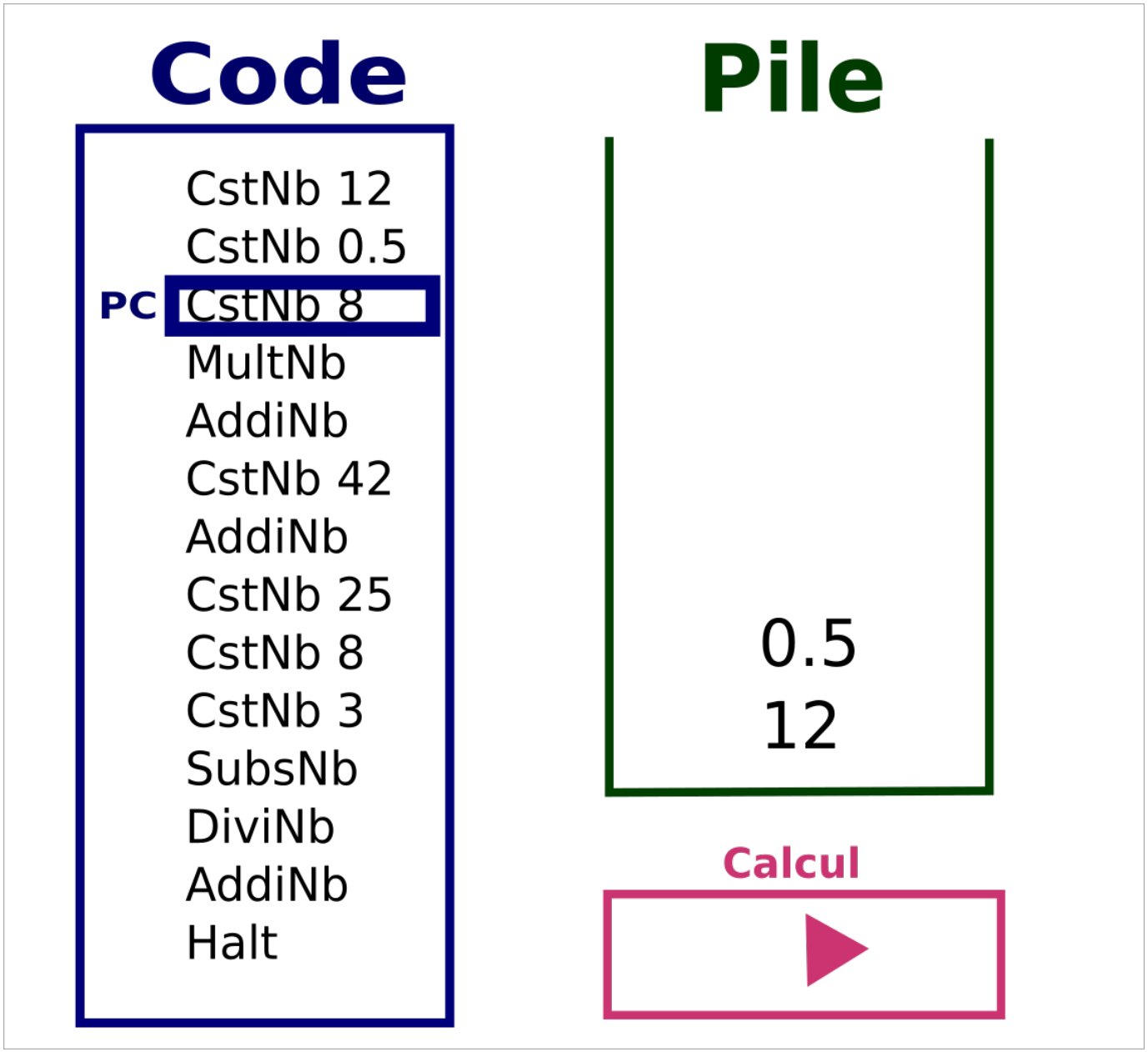
# Example


Code
Pile
PC
CstNb 12
CstNb 0.5
CstNb 8
MultNb
AddiNb
CstNb 42
AddiNb
CstNb 25
CstNb 8
CstNb 3
SubsNb
DiviNb
AddiNb
Halt
Calcul

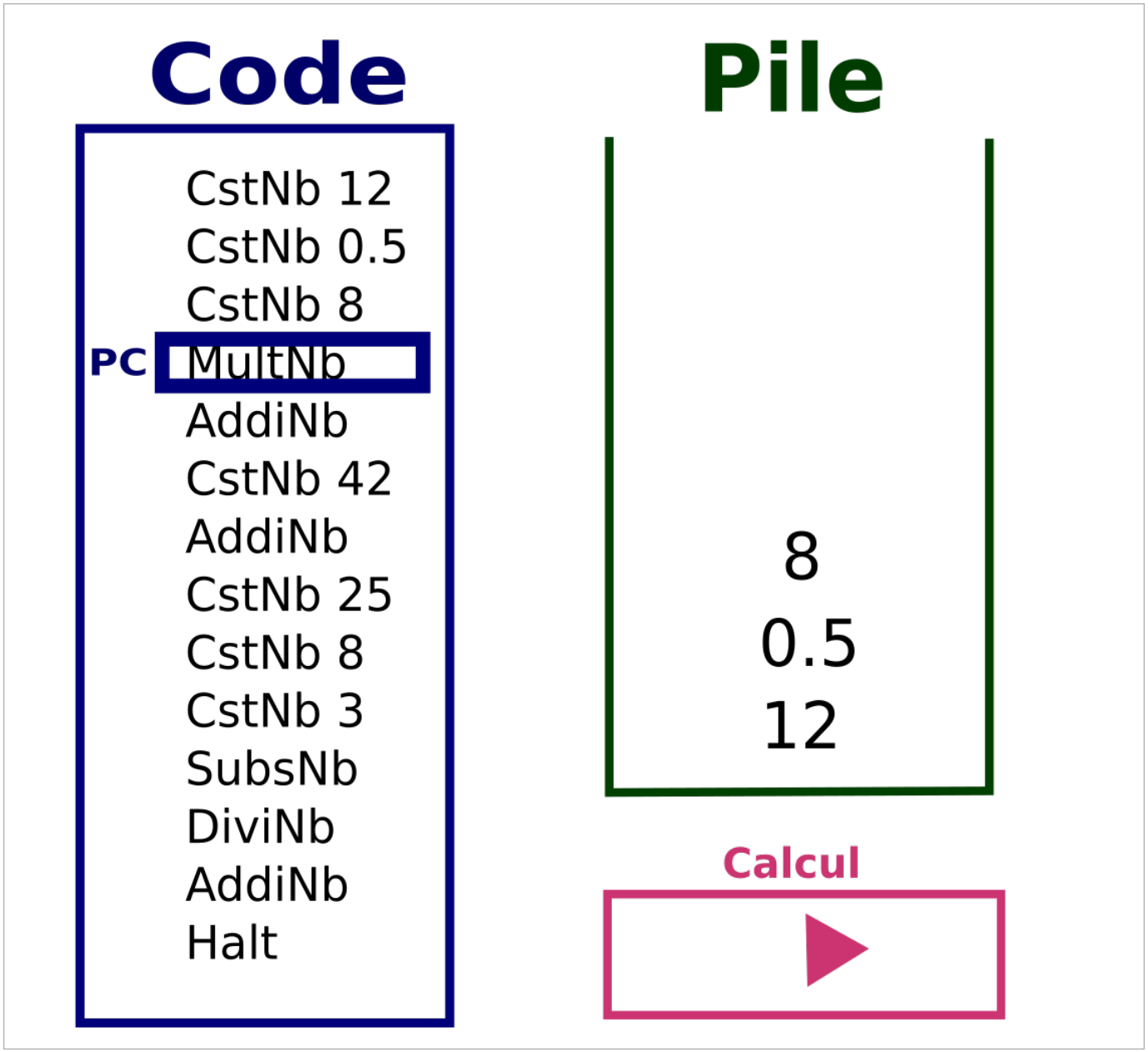
# Example


Code
CstNb 12
PC
CstNb 0.5
CstNb 8
MultNb
AddiNb
CstNb 42
AddiNb
CstNb 25
CstNb 8
CstNb 3
SubsNb
DiviNb
AddiNb
Halt
Pile
12
Calcul

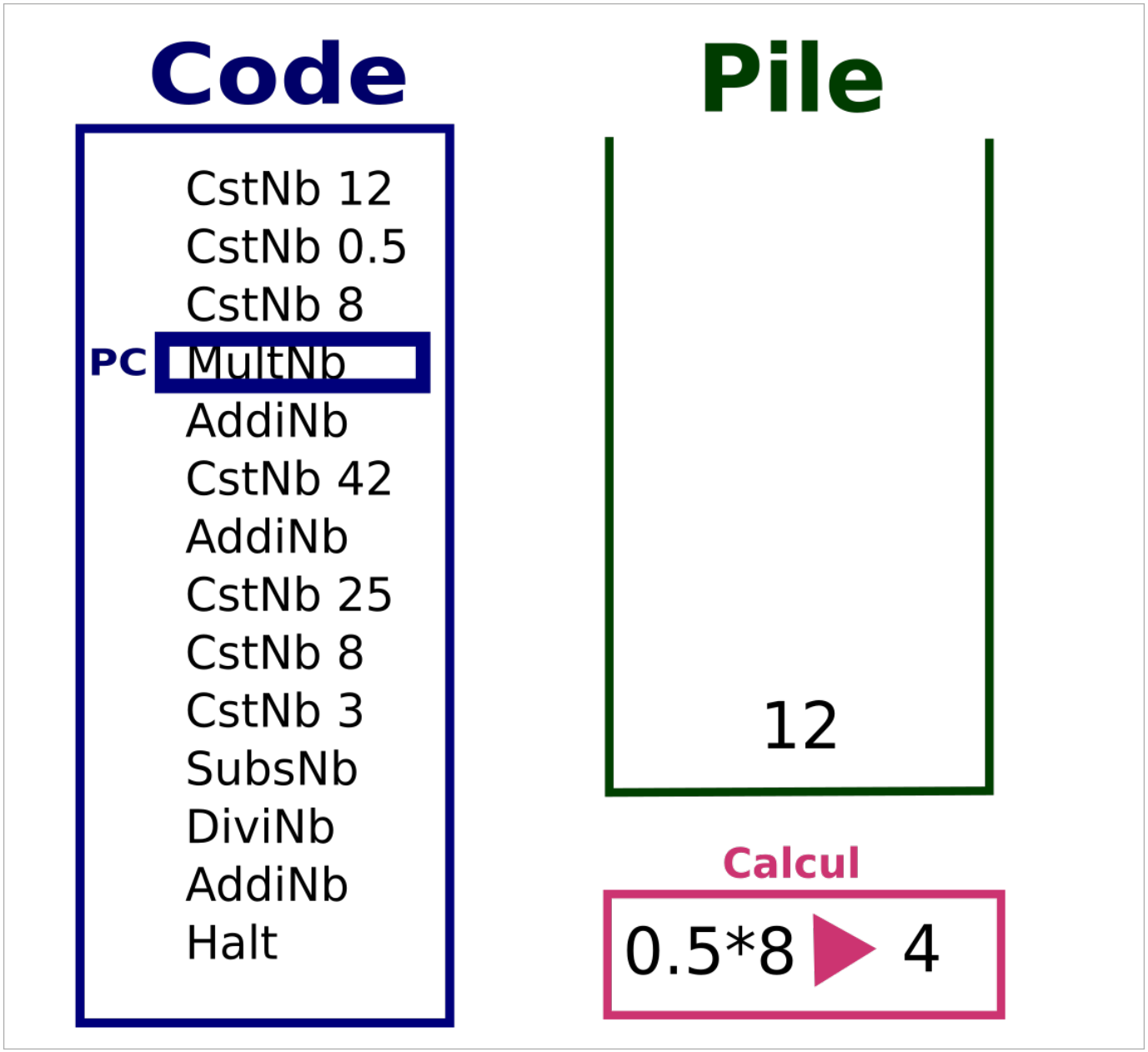
# Example


Code
CstNb 12
CstNb 0.5
PC
CstNb 8
MultNb
AddiNb
CstNb 42
AddiNb
CstNb 25
CstNb 8
CstNb 3
SubsNb
DiviNb
AddiNb
Halt
Pile
0.5
12
Calcul

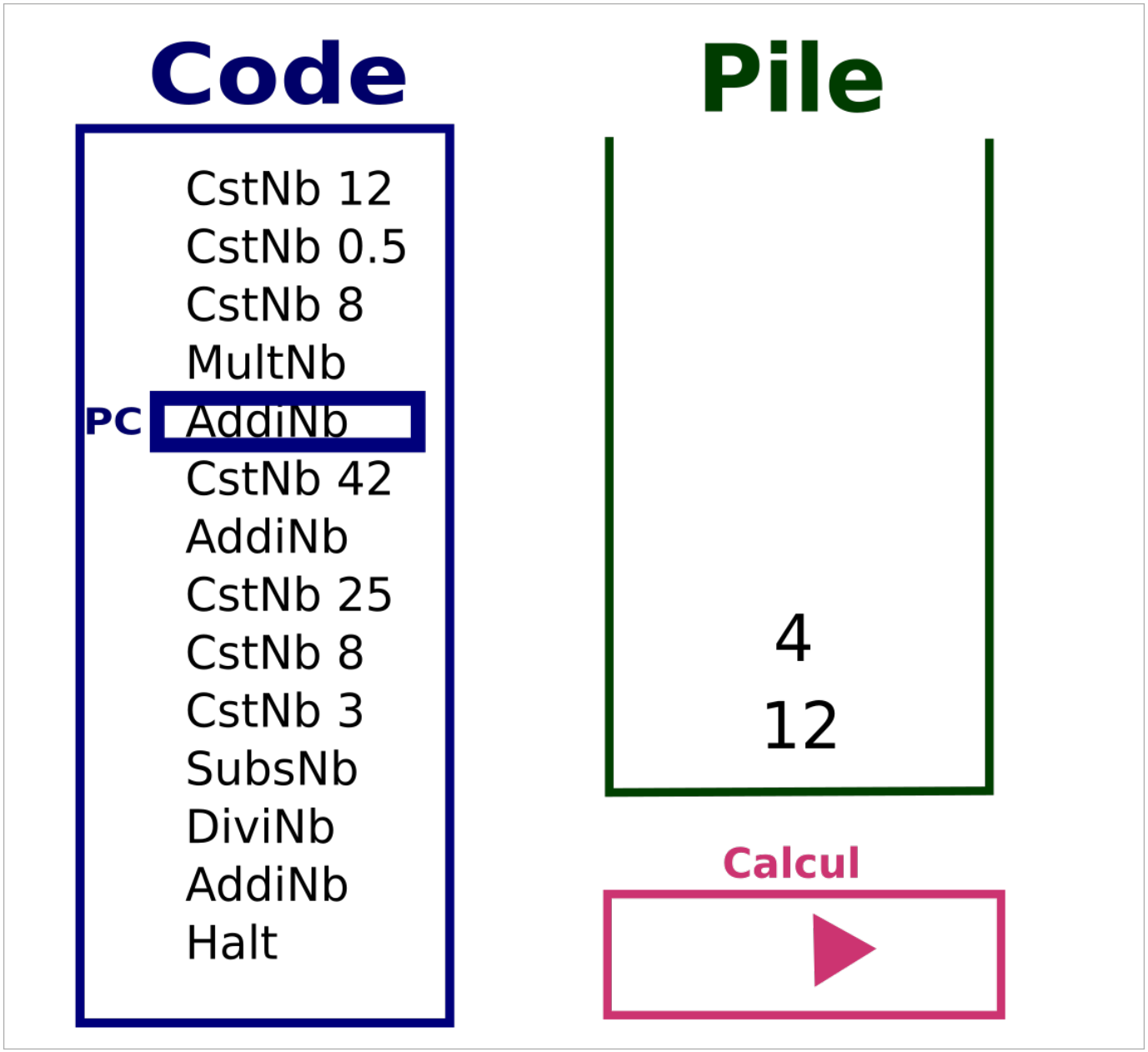
# Example


Code
CstNb 12
CstNb 0.5
CstNb 8
PC
MultNb
AddiNb
CstNb 42
AddiNb
CstNb 25
CstNb 8
CstNb 3
SubsNb
DiviNb
AddiNb
Halt
Pile
8
0.5
12
Calcul

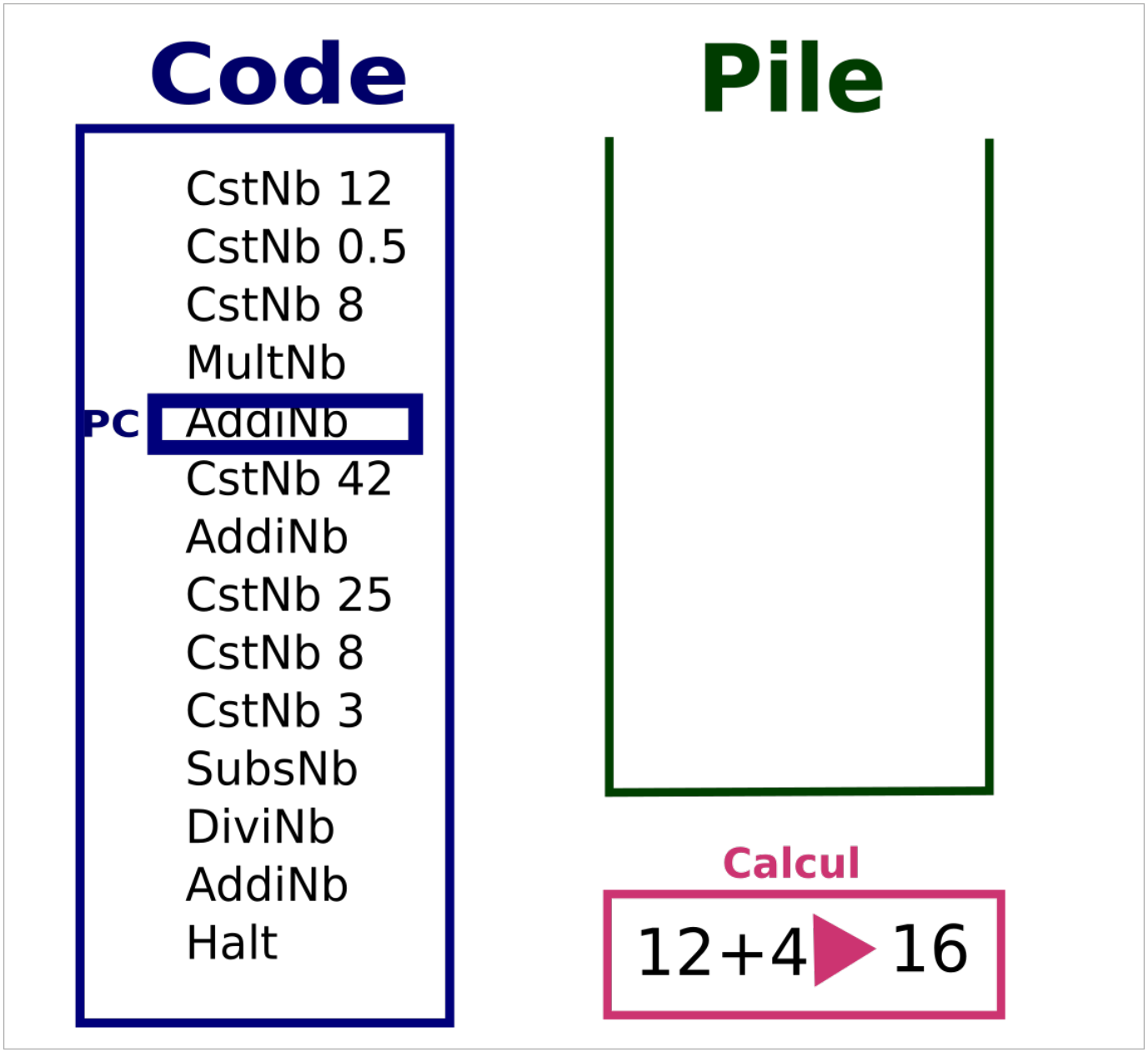
# Example


Code
CstNb 12
CstNb 0.5
CstNb 8
PC
MultNb
AddiNb
CstNb 42
AddiNb
CstNb 25
CstNb 8
CstNb 3
SubsNb
DiviNb
AddiNb
Halt
Pile
12
Calcul
0.5*8
4

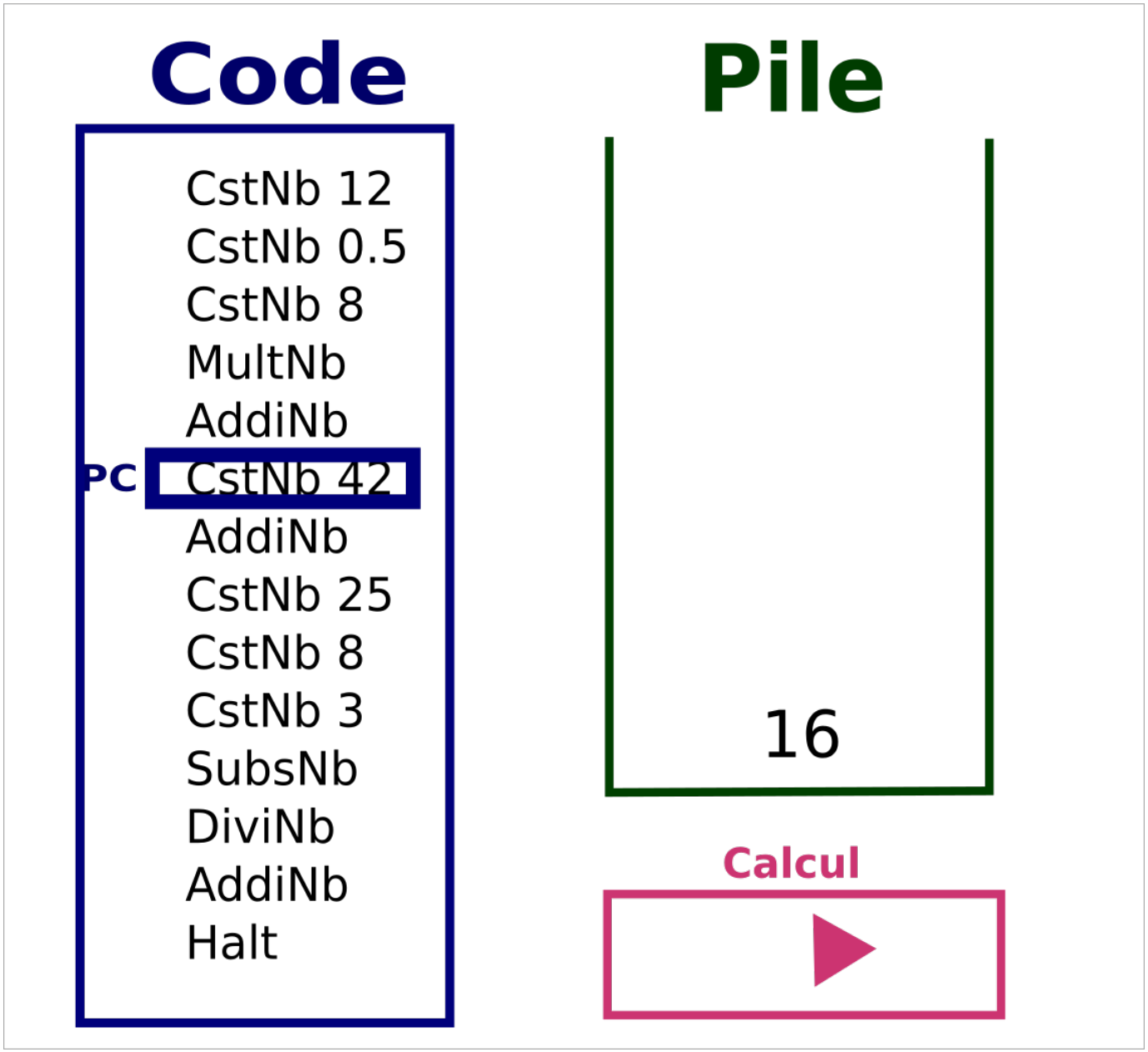
# Example


Code
CstNb 12
CstNb 0.5
CstNb 8
MultNb
PC
AddiNb
CstNb 42
AddiNb
CstNb 25
CstNb 8
CstNb 3
SubsNb
DiviNb
AddiNb
Halt
Pile
4
12
Calcul

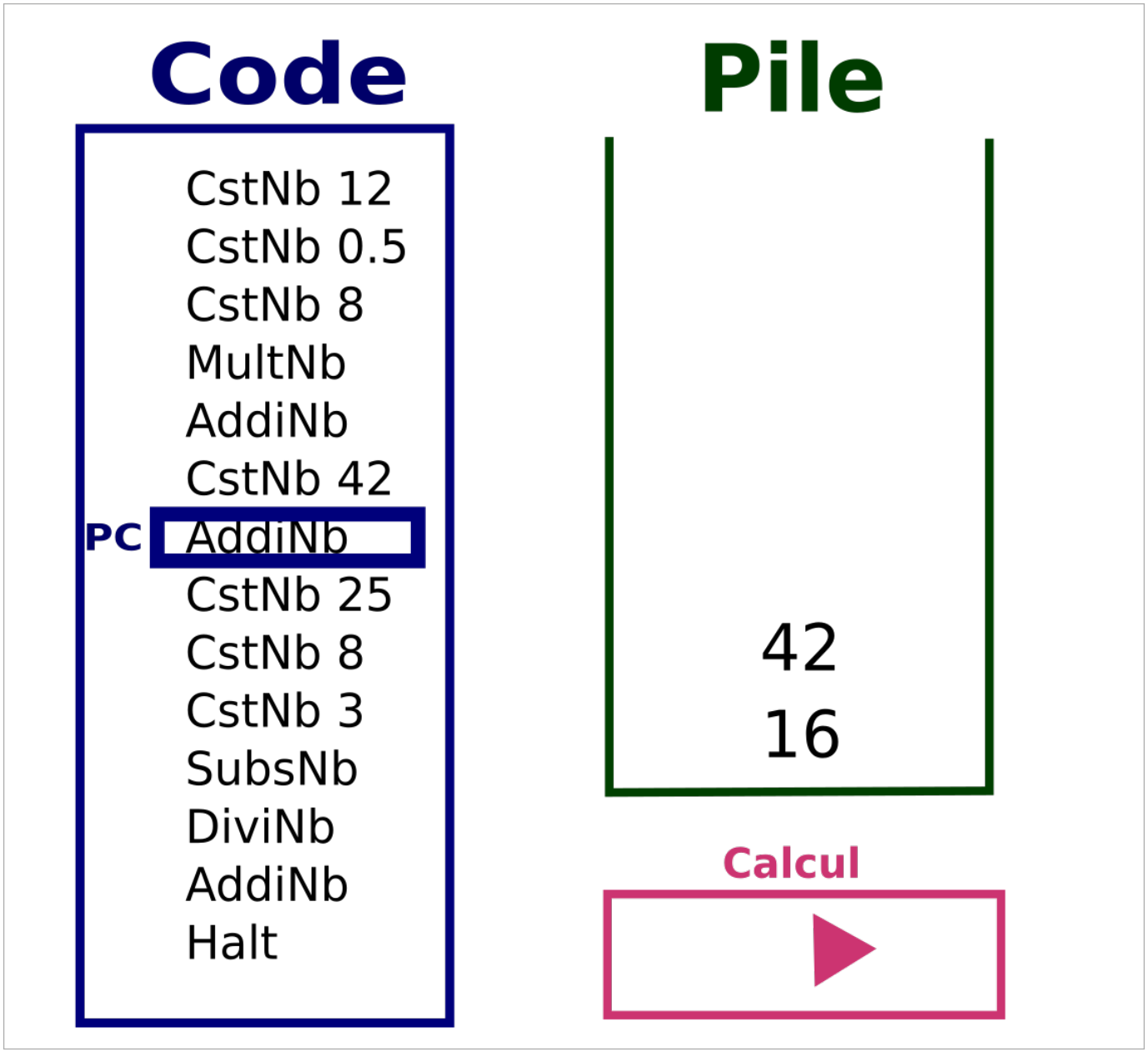
# Example


Code
CstNb 12
CstNb 0.5
CstNb 8
MultNb
PC
AddiNb
CstNb 42
AddiNb
CstNb 25
CstNb 8
CstNb 3
SubsNb
DiviNb
AddiNb
Halt
Pile
Calcul
12+4
16

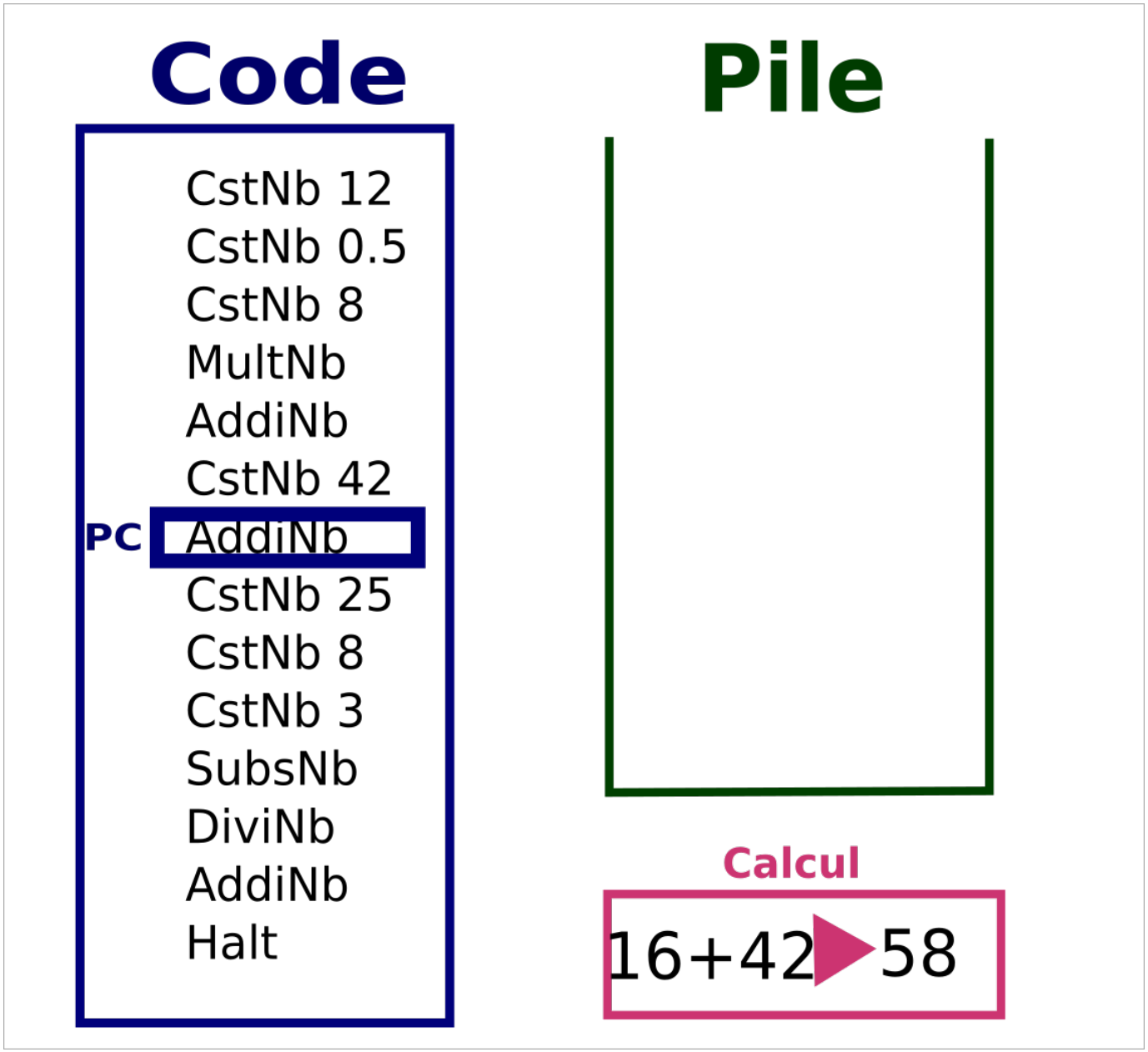
# Example


Code
CstNb 12
CstNb 0.5
CstNb 8
MultNb
AddiNb
PC
CstNb 42
AddiNb
CstNb 25
CstNb 8
CstNb 3
SubsNb
DiviNb
AddiNb
Halt
Pile
16
Calcul

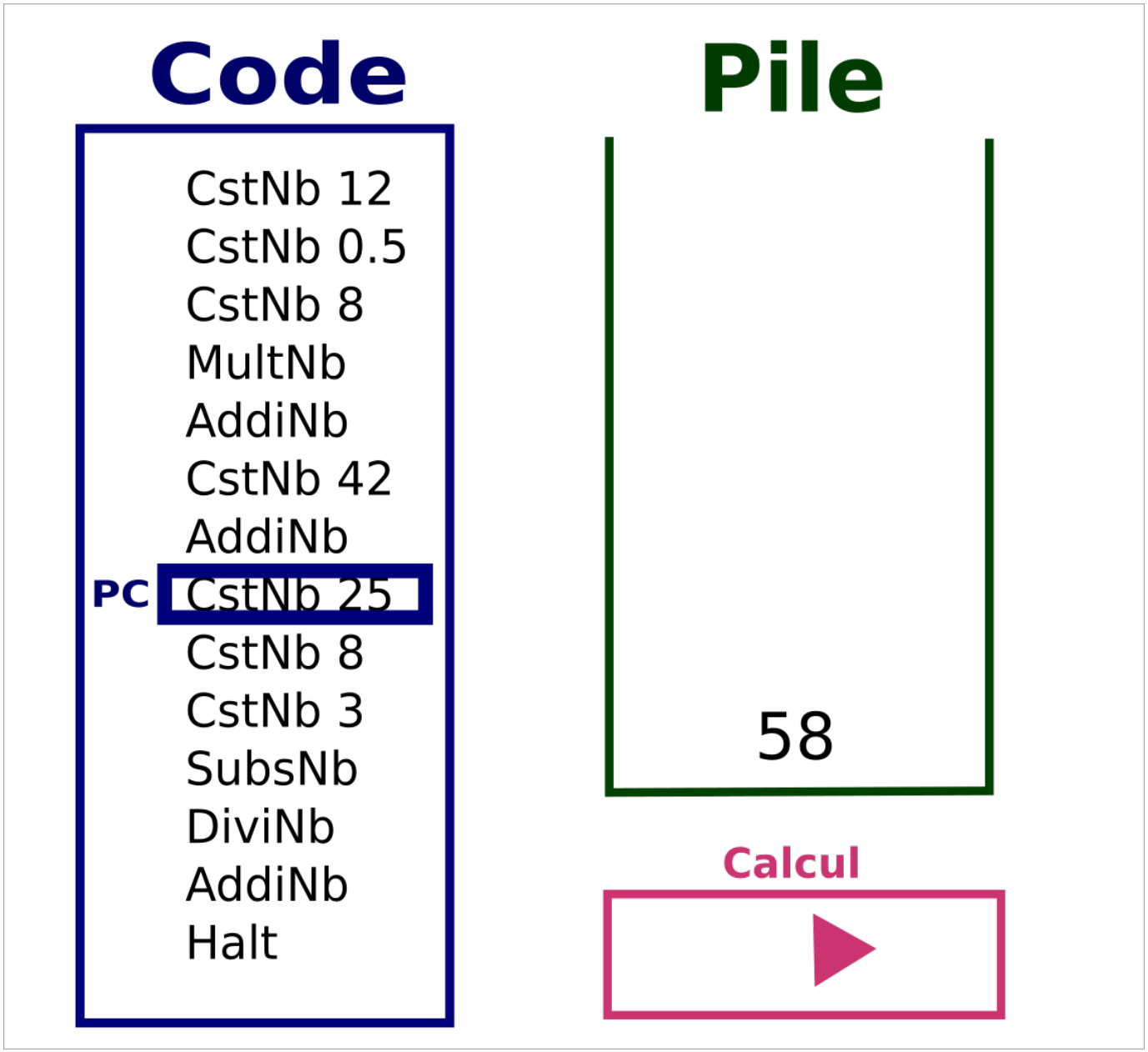
# Example


Code
CstNb 12
CstNb 0.5
CstNb 8
MultNb
AddiNb
CstNb 42
PC
AddiNb
CstNb 25
CstNb 8
CstNb 3
SubsNb
DiviNb
AddiNb
Halt
Pile
42
16
Calcul

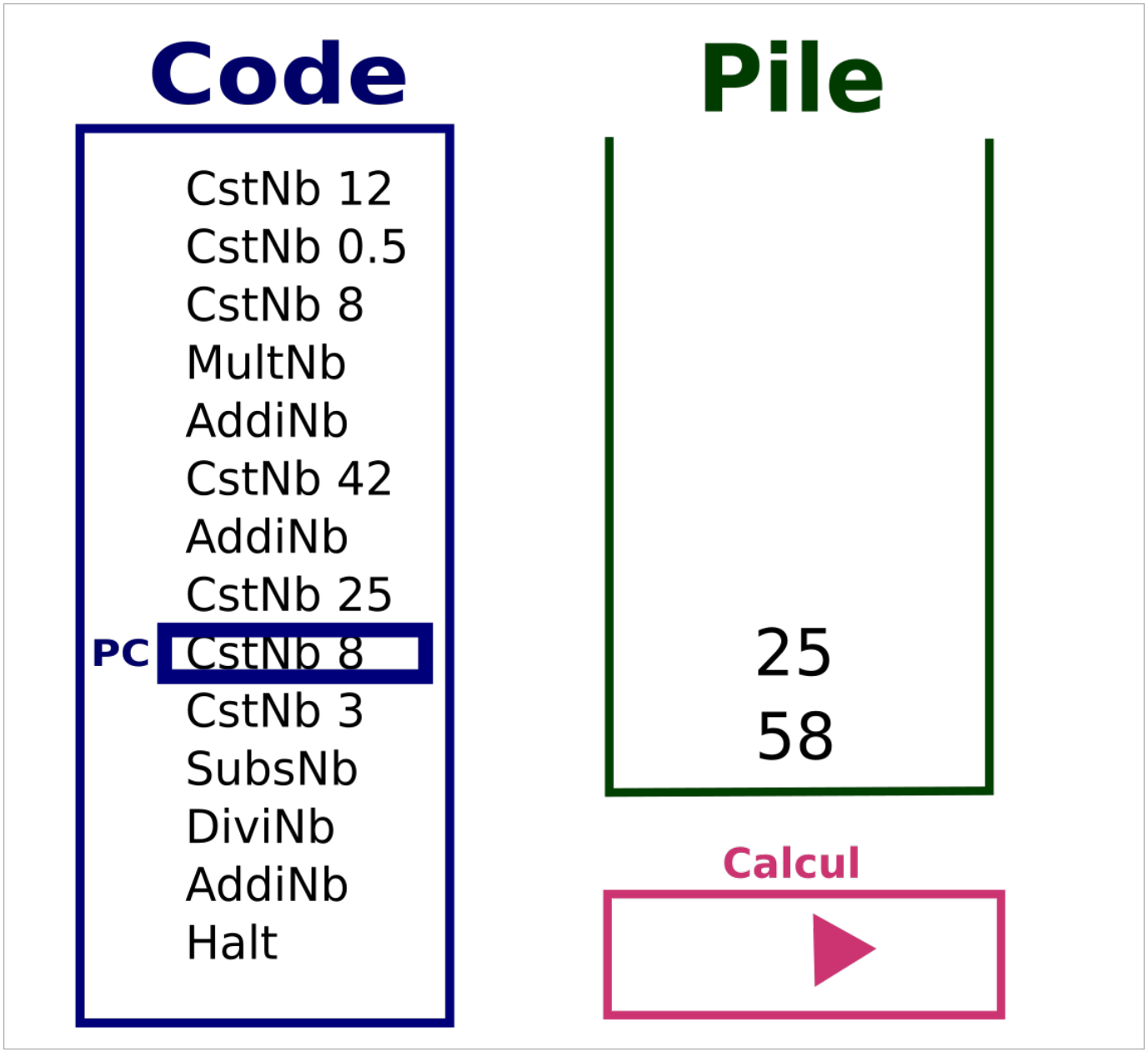
# Example


Code
CstNb 12
CstNb 0.5
CstNb 8
MultNb
AddiNb
CstNb 42
PC
AddiNb
CstNb 25
CstNb 8
CstNb 3
SubsNb
DiviNb
AddiNb
Halt
Pile
Calcul
16+42
58

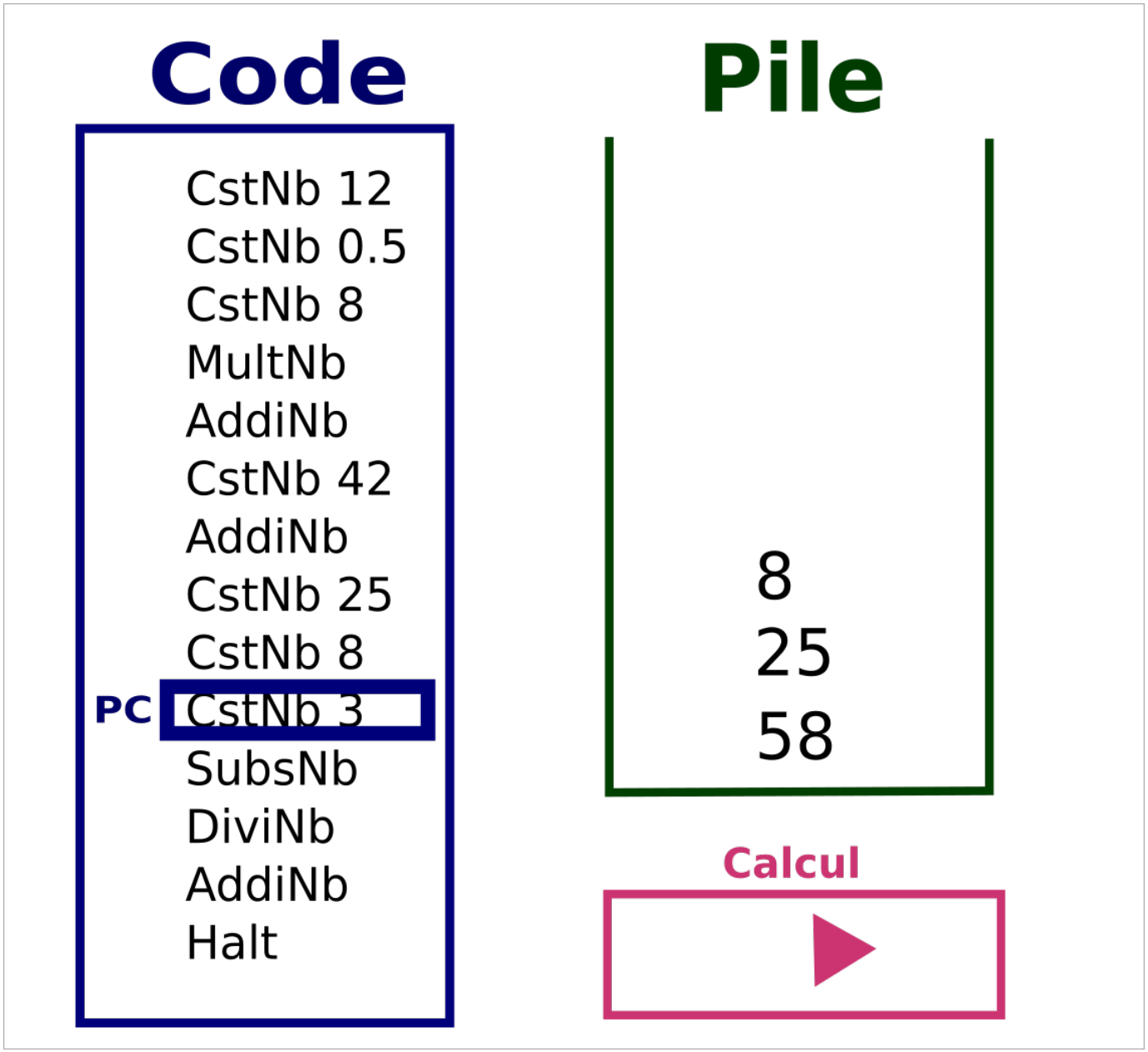
# Example


Code
CstNb 12
CstNb 0.5
CstNb 8
MultNb
AddiNb
CstNb 42
AddiNb
PC
CstNb 25
CstNb 8
CstNb 3
SubsNb
DiviNb
AddiNb
Halt
Pile
58
Calcul

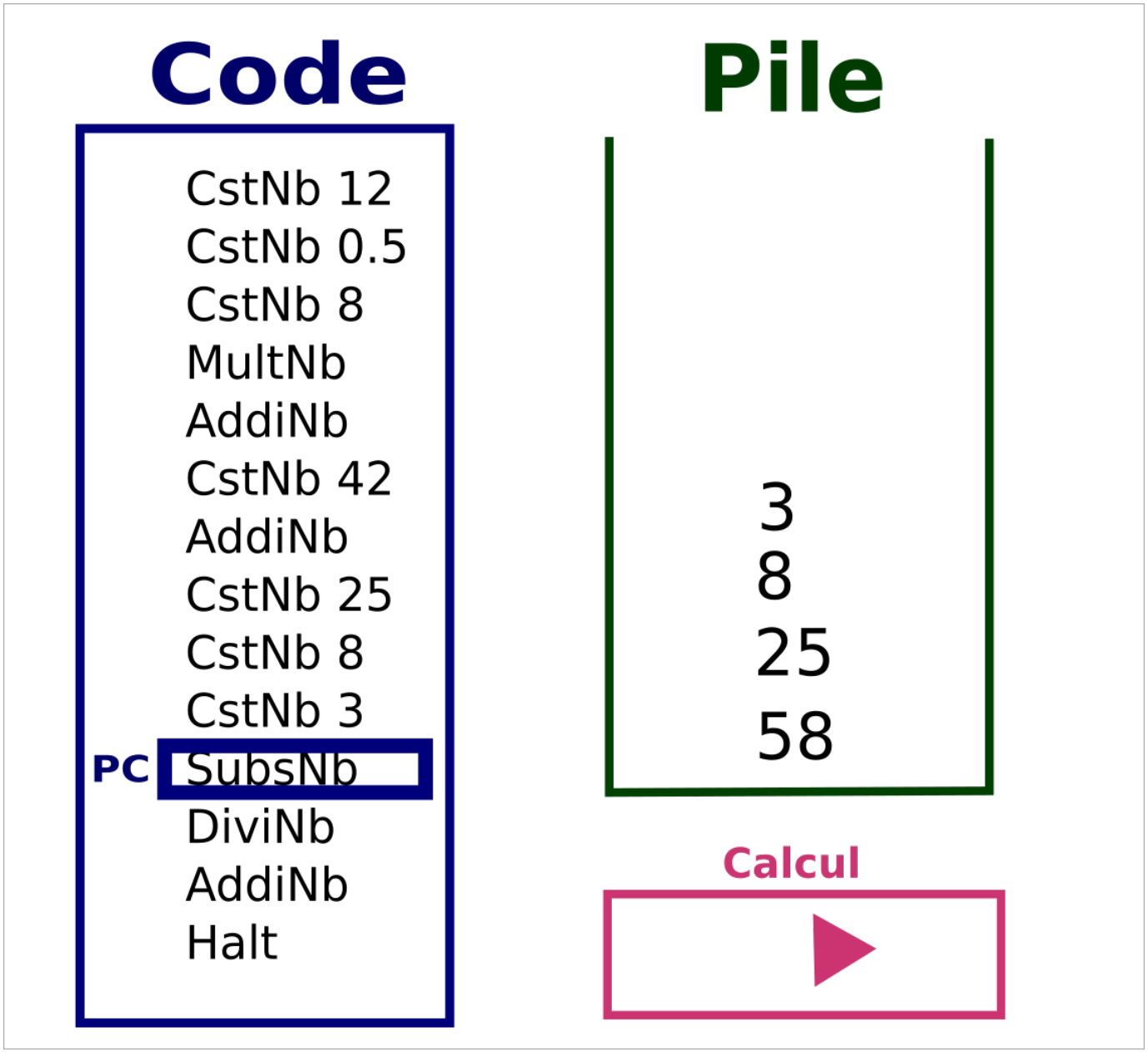
# Example


Code
CstNb 12
CstNb 0.5
CstNb 8
MultNb
AddiNb
CstNb 42
AddiNb
CstNb 25
PC
CstNb 8
CstNb 3
SubsNb
DiviNb
AddiNb
Halt
Pile
25
58
Calcul

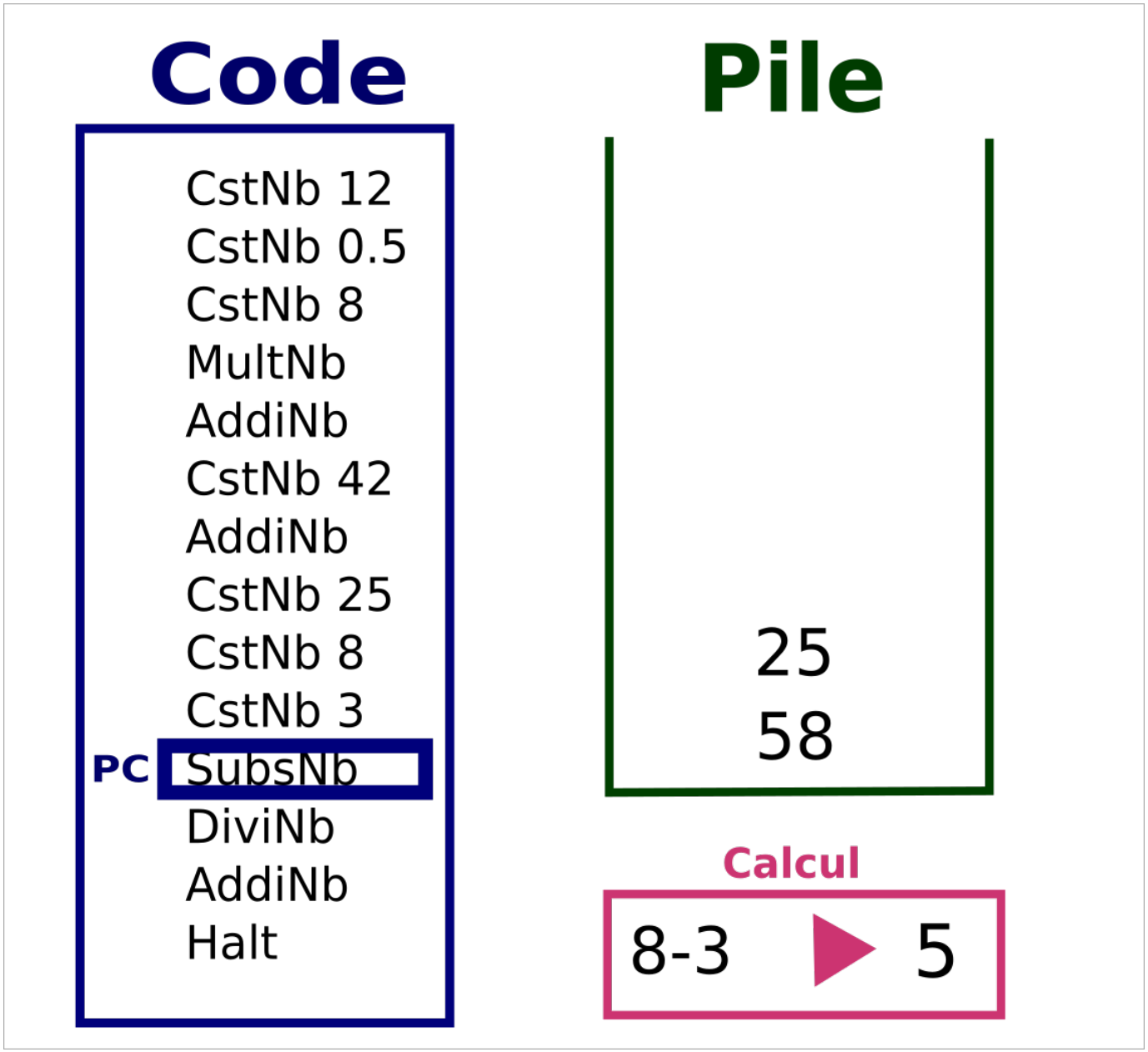
# Example


Code
CstNb 12
CstNb 0.5
CstNb 8
MultNb
AddiNb
CstNb 42
AddiNb
CstNb 25
CstNb 8
PC
CstNb 3
SubsNb
DiviNb
AddiNb
Halt
Pile
8
25
58
Calcul

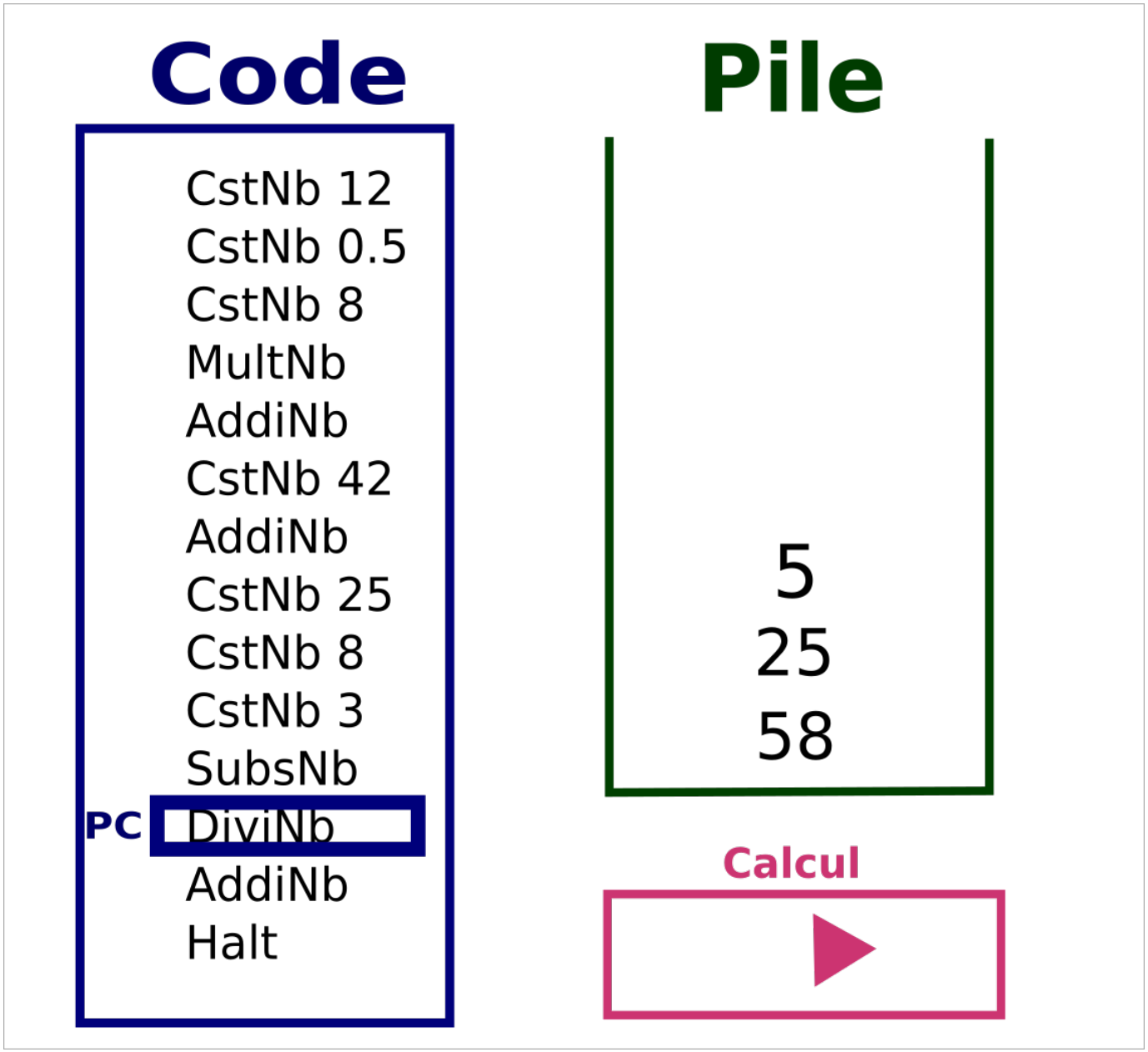
# Example


Code
CstNb 12
CstNb 0.5
CstNb 8
MultNb
AddiNb
CstNb 42
AddiNb
CstNb 25
CstNb 8
CstNb 3
PC
SubsNb
DiviNb
AddiNb
Halt
Pile
3
8
25
58
Calcul

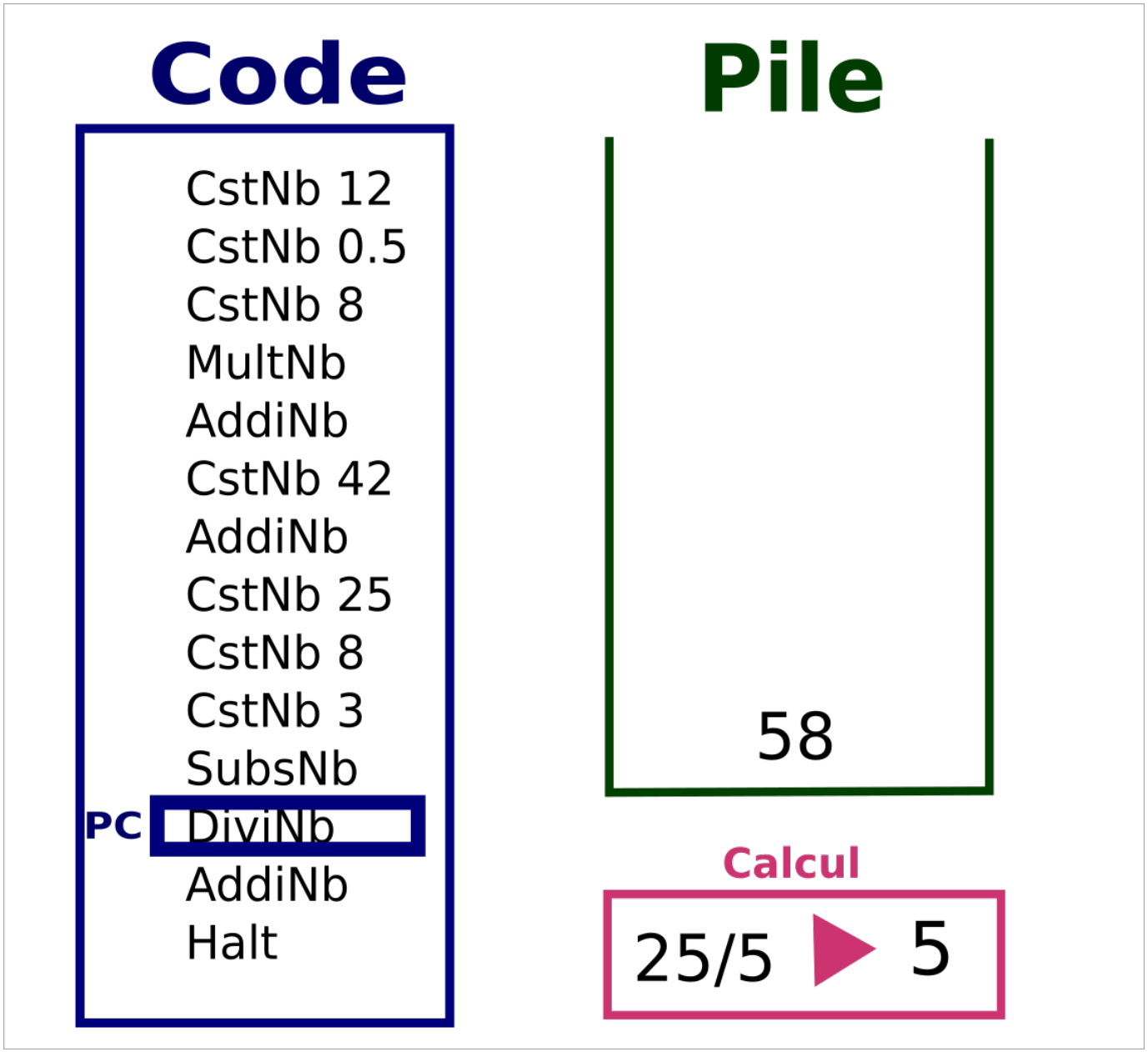
# Example


Code
CstNb 12
CstNb 0.5
CstNb 8
MultNb
AddiNb
CstNb 42
AddiNb
CstNb 25
CstNb 8
CstNb 3
PC
SubsNb
DiviNb
AddiNb
Halt
Pile
25
58
Calcul
8-3
5

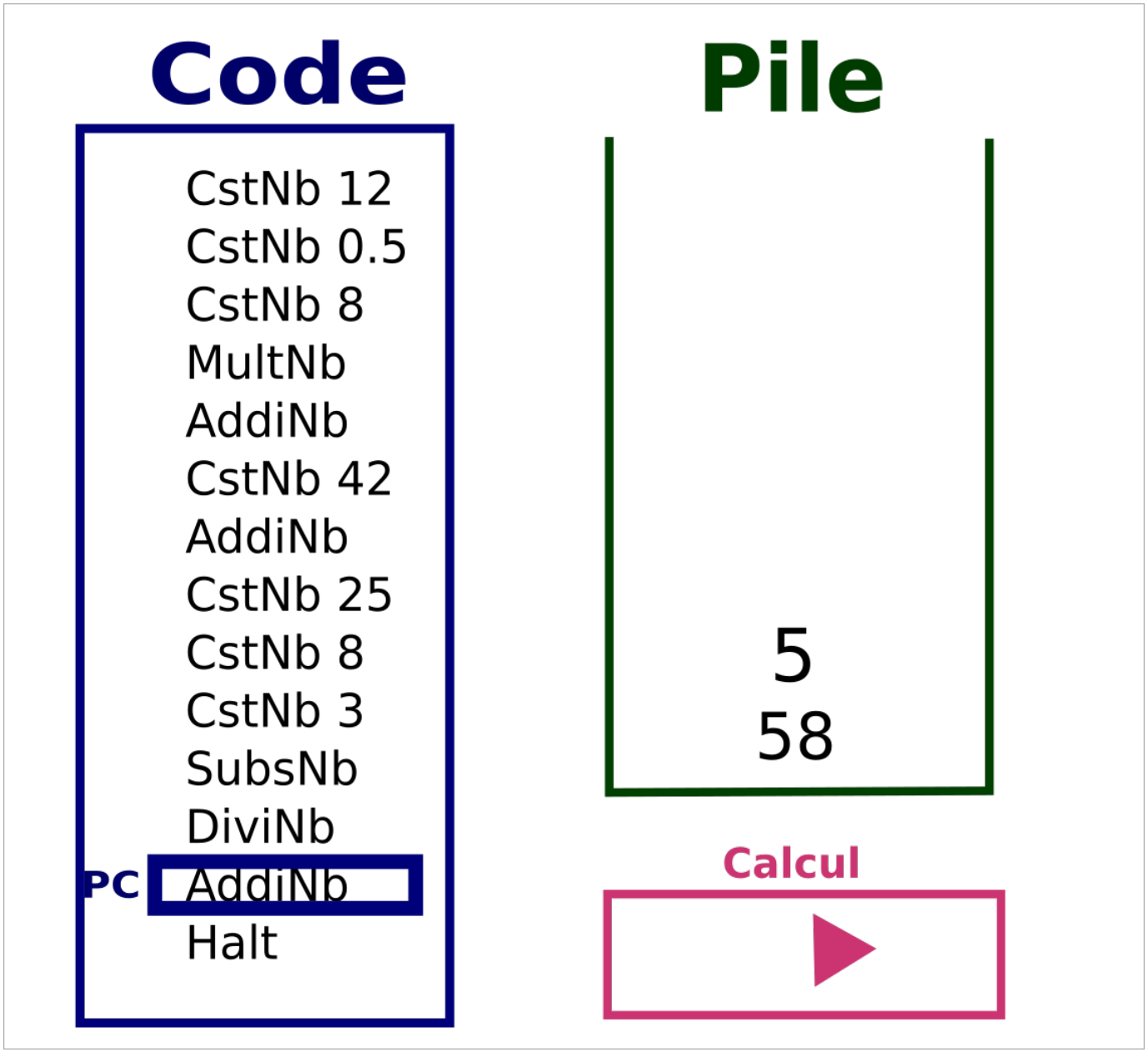
# Example


Code
CstNb 12
CstNb 0.5
CstNb 8
MultNb
AddiNb
CstNb 42
AddiNb
CstNb 25
CstNb 8
CstNb 3
SubsNb
PC
DiviNb
AddiNb
Halt
Pile
5
25
58
Calcul

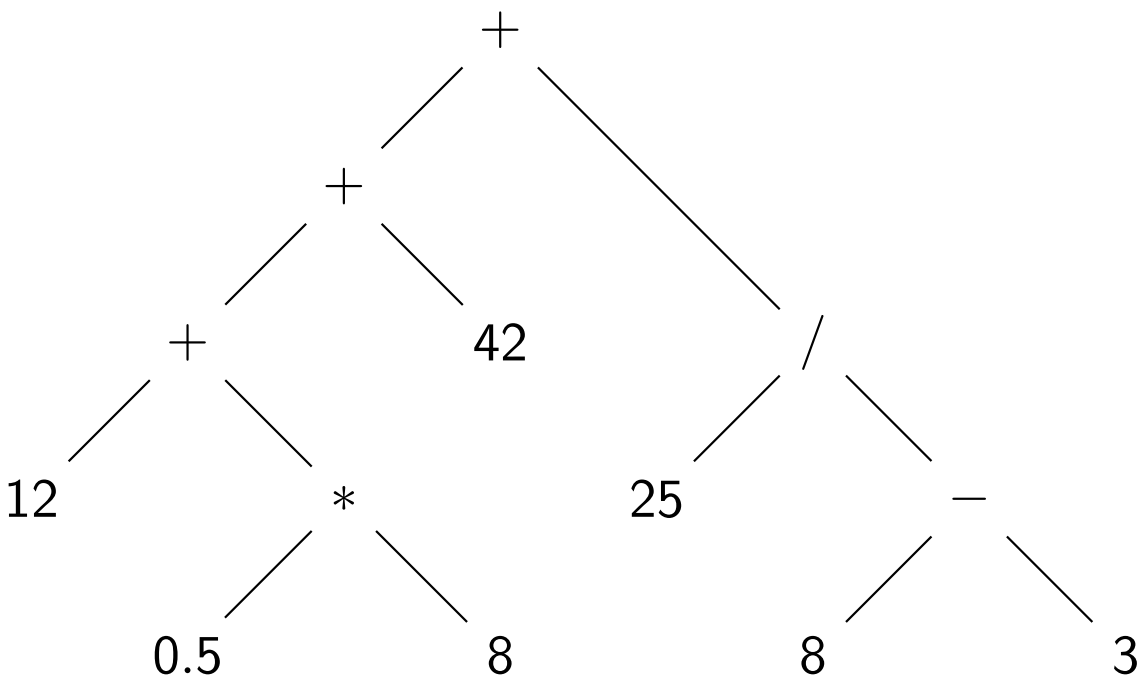
# Example


Code
CstNb 12
CstNb 0.5
CstNb 8
MultNb
AddiNb
CstNb 42
AddiNb
CstNb 25
CstNb 8
CstNb 3
SubsNb
PC
DiviNb
AddiNb
Halt
Pile
58
Calcul
25/5
5

# Example

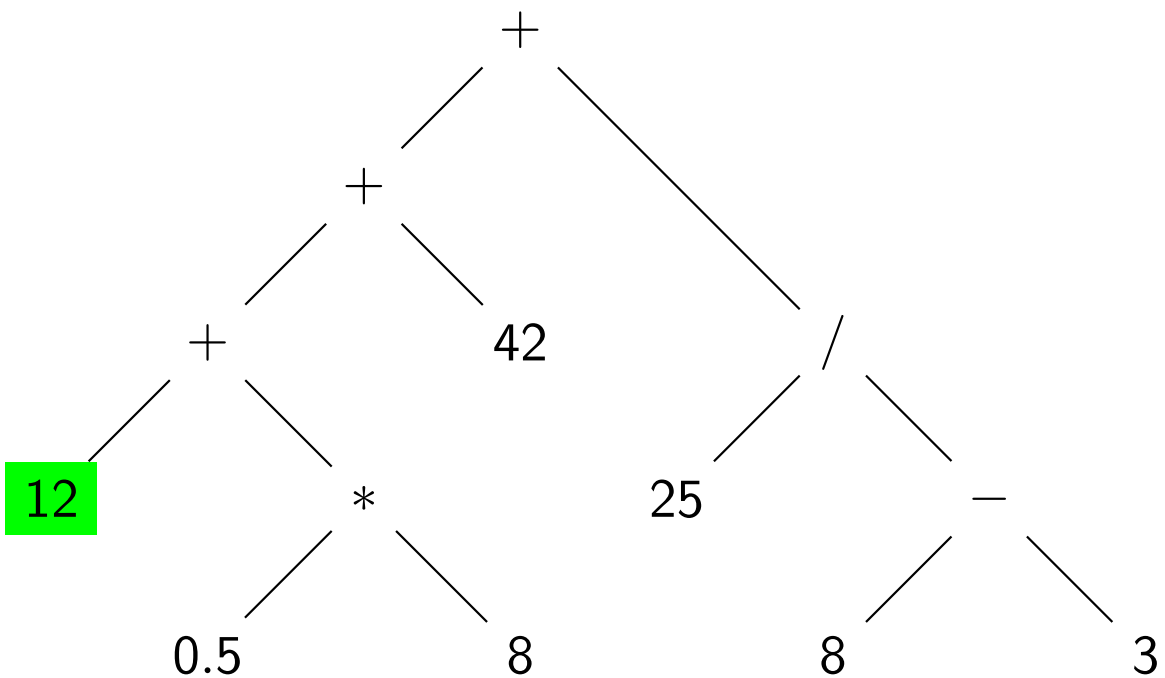

Code
CstNb 12
CstNb 0.5
CstNb 8
MultNb
AddiNb
CstNb 42
AddiNb
CstNb 25
CstNb 8
CstNb 3
SubsNb
DiviNb
PC
AddiNb
Halt
Pile
5
58
Calcul

# From AST to assembly : Postfixe encoding !




## Produced code

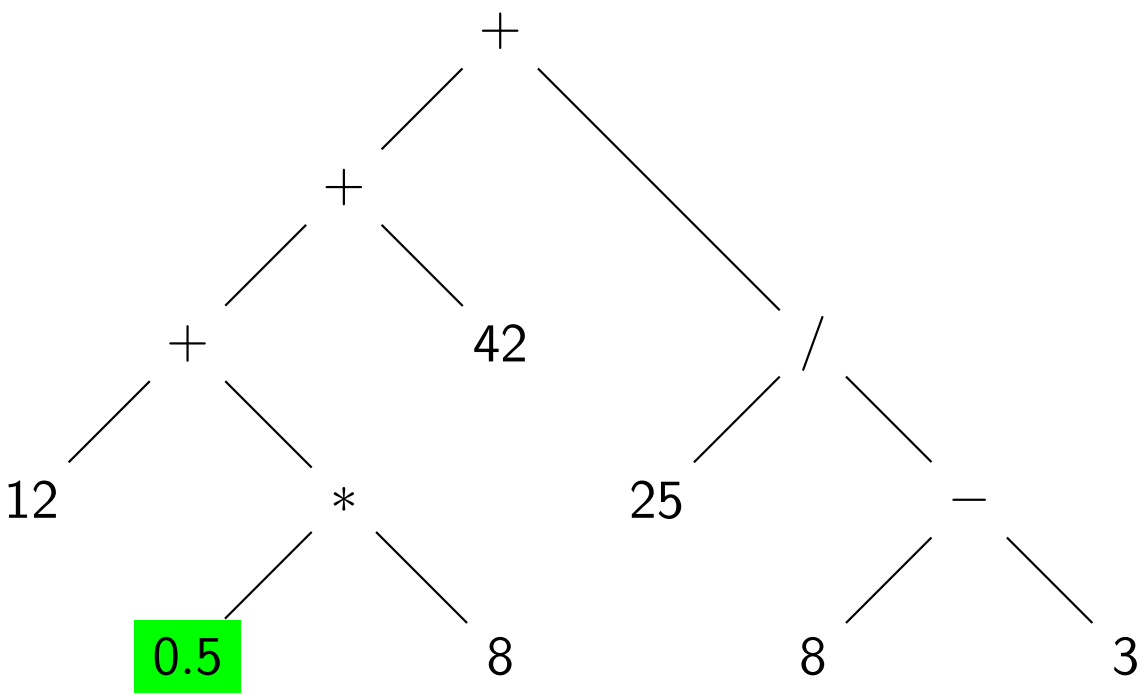
# From AST to assembly : Postfixe encoding !




## Produced code

**CsteNb** 12

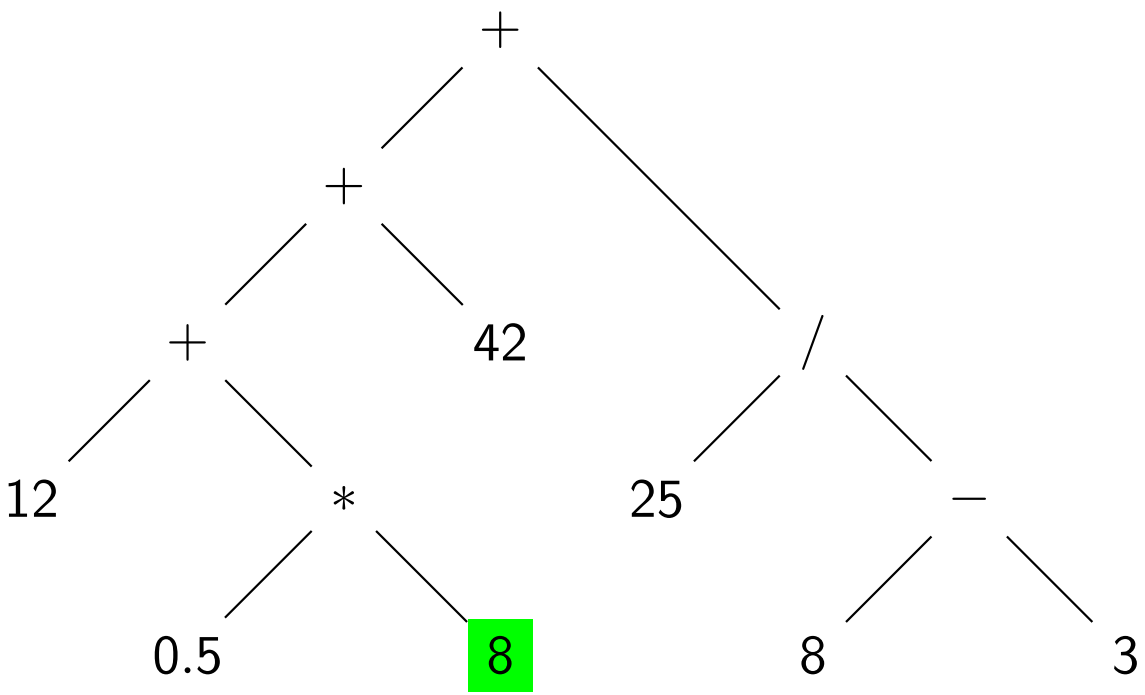
# From AST to assembly : Postfixe encoding !




## Produced code

**CsteNb** 12
**CsteNb** 0.5

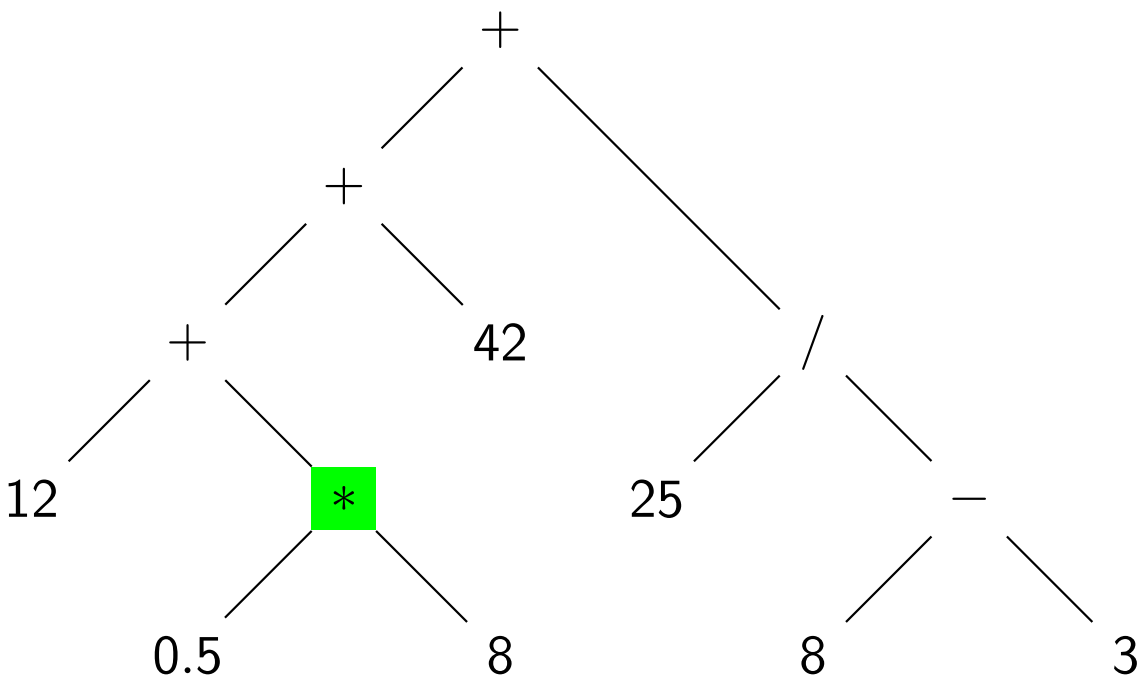
# From AST to assembly : Postfixe encoding !




## Produced code

**CsteNb** 12
**CsteNb** 0.5
**CsteNb** 8

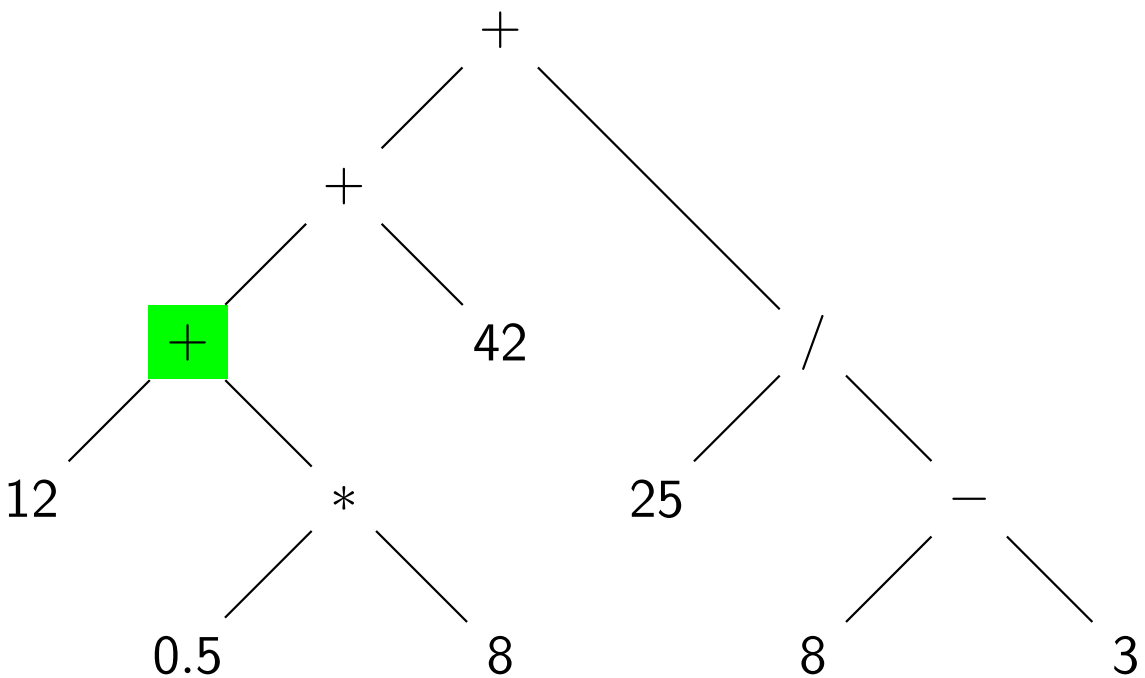
# From AST to assembly : Postfixe encoding !




## Produced code

**CsteNb** 12
**CsteNb** 0.5
**CsteNb** 8
**MultNb**

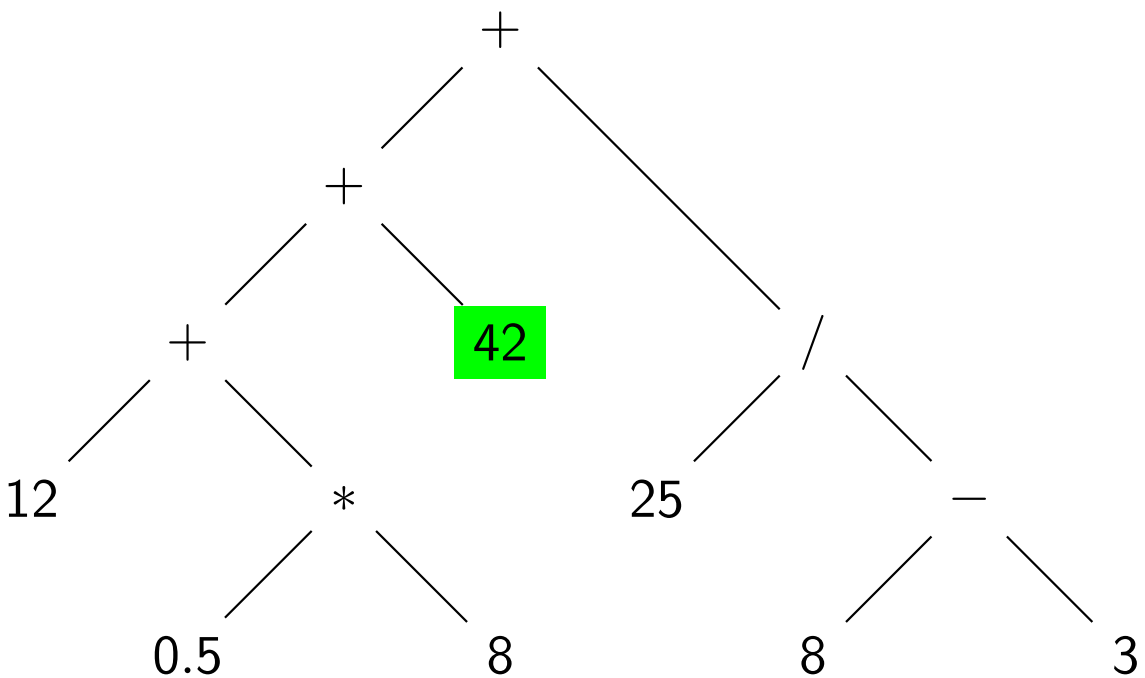
# From AST to assembly : Postfixe encoding !




## Produced code

**CsteNb** 12
**CsteNb** 0.5
**CsteNb** 8
**MultNb**
**AddiNb**

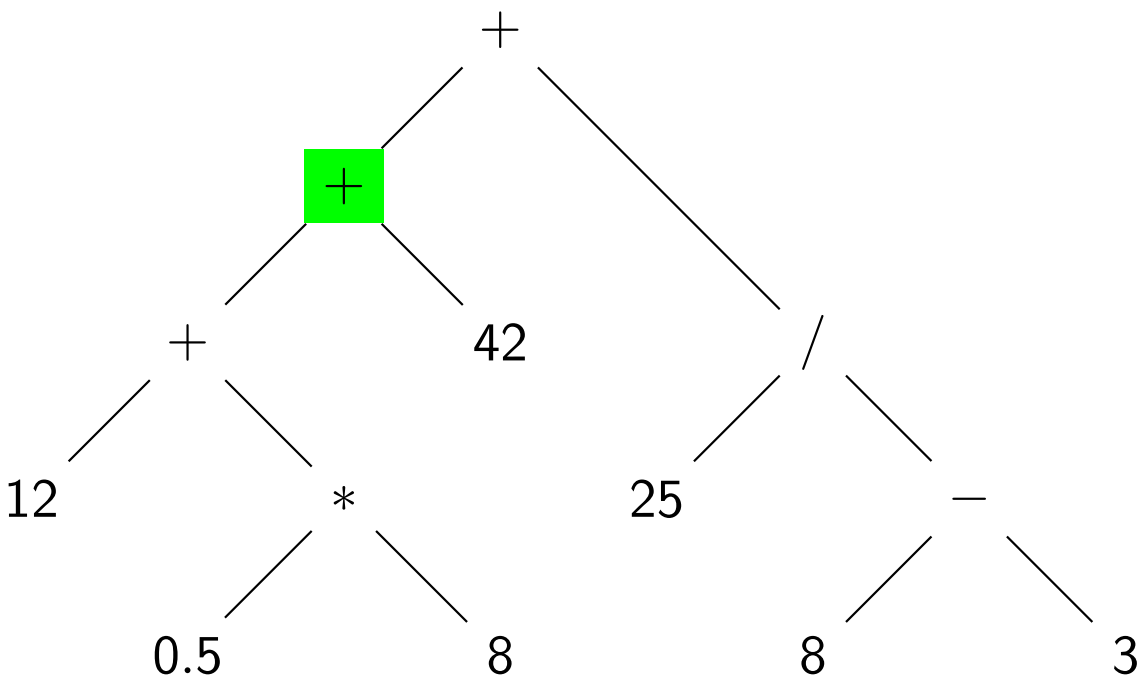
# From AST to assembly : Postfixe encoding !




## Produced code

**CsteNb** 12
**CsteNb** 0.5
**CsteNb** 8
**MultNb**
**AddiNb**
**CsteNb** 42

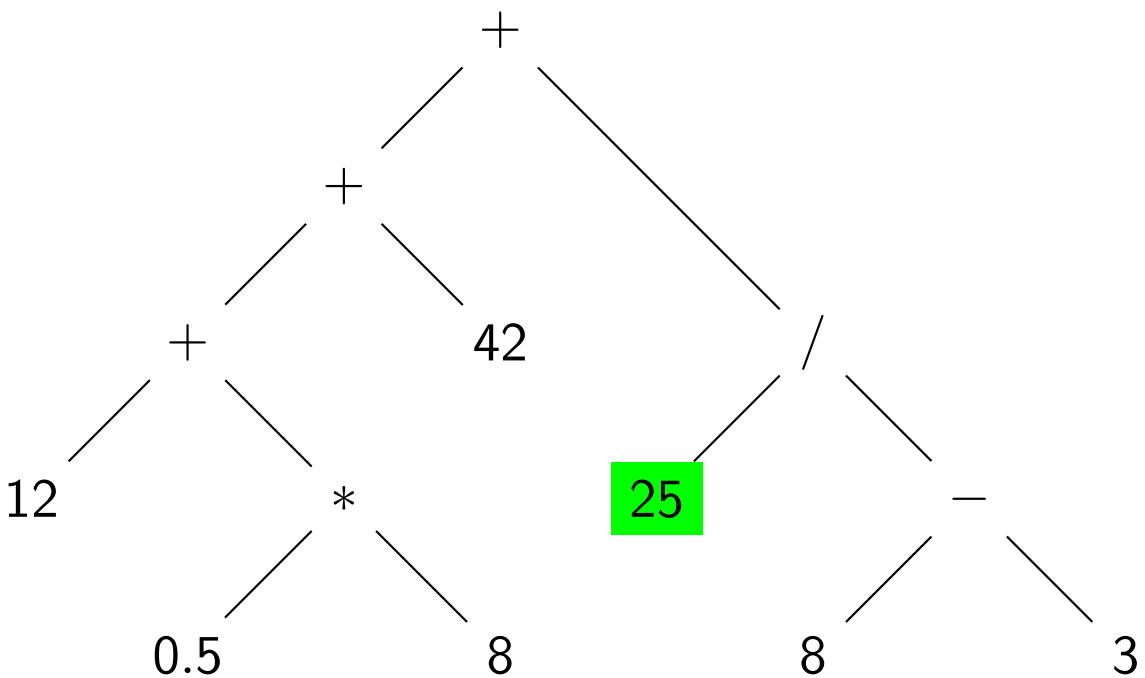
# From AST to assembly : Postfixe encoding !




## Produced code

**CsteNb** 12
**CsteNb** 0.5
**CsteNb** 8
**MultNb**
**AddiNb**
**CsteNb** 42
**AddiNb**

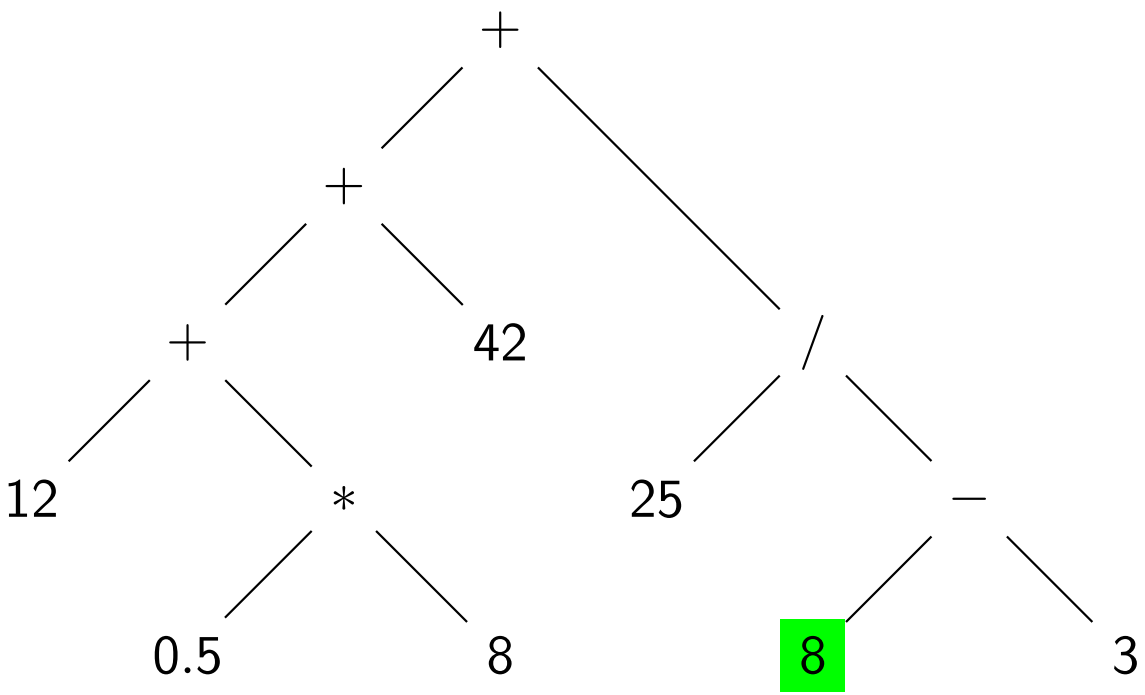
# From AST to assembly : Postfixe encoding !




## Produced code

**CsteNb** 12
**CsteNb** 0.5
**CsteNb** 8
**MultNb**
**AddiNb**
**CsteNb** 42
**AddiNb**
**CsteNb** 25

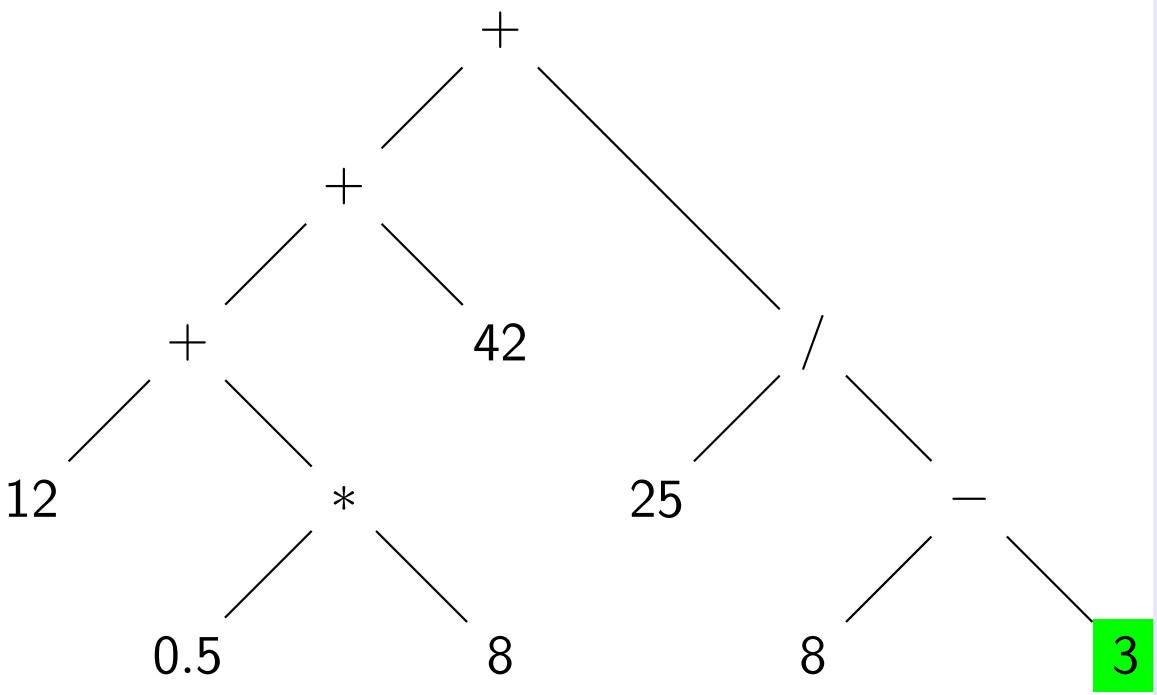
# From AST to assembly : Postfixe encoding !




## Produced code

**CsteNb** 12
**CsteNb** 0.5
**CsteNb** 8
**MultNb**
**AddiNb**
**CsteNb** 42
**AddiNb**
**CsteNb** 25
**CsteNb** 8

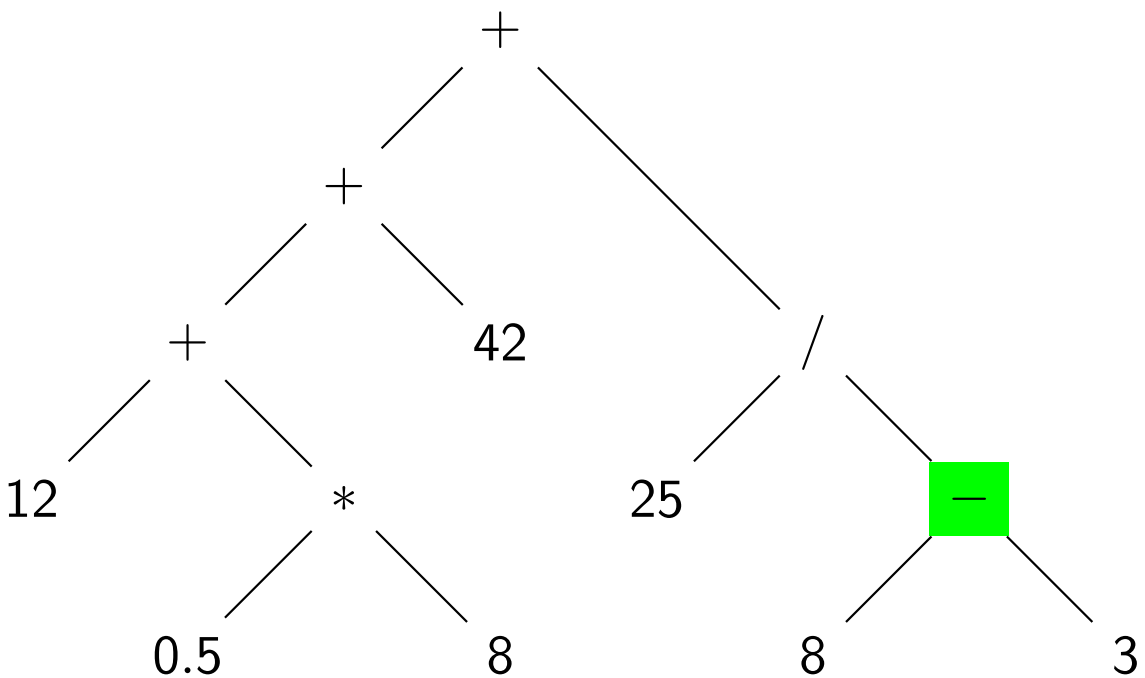
# From AST to assembly : Postfixe encoding !




## Produced code

**CsteNb** 12
**CsteNb** 0.5
**CsteNb** 8
**MultNb**
**AddiNb**
**CsteNb** 42
**AddiNb**
**CsteNb** 25
**CsteNb** 8
**CsteNb** 3

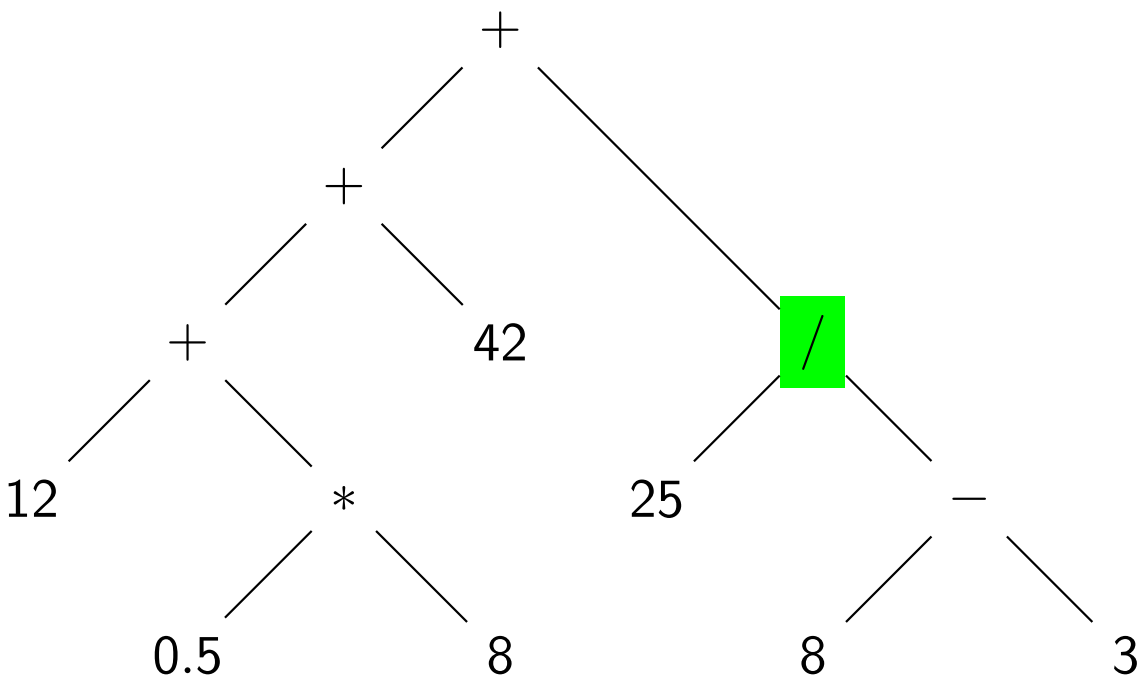
# From AST to assembly : Postfixe encoding !




## Produced code

**CsteNb** 12
**CsteNb** 0.5
**CsteNb** 8
**MultNb**
**AddiNb**
**CsteNb** 42
**AddiNb**
**CsteNb** 25
**CsteNb** 8
**CsteNb** 3
**SubsNb**

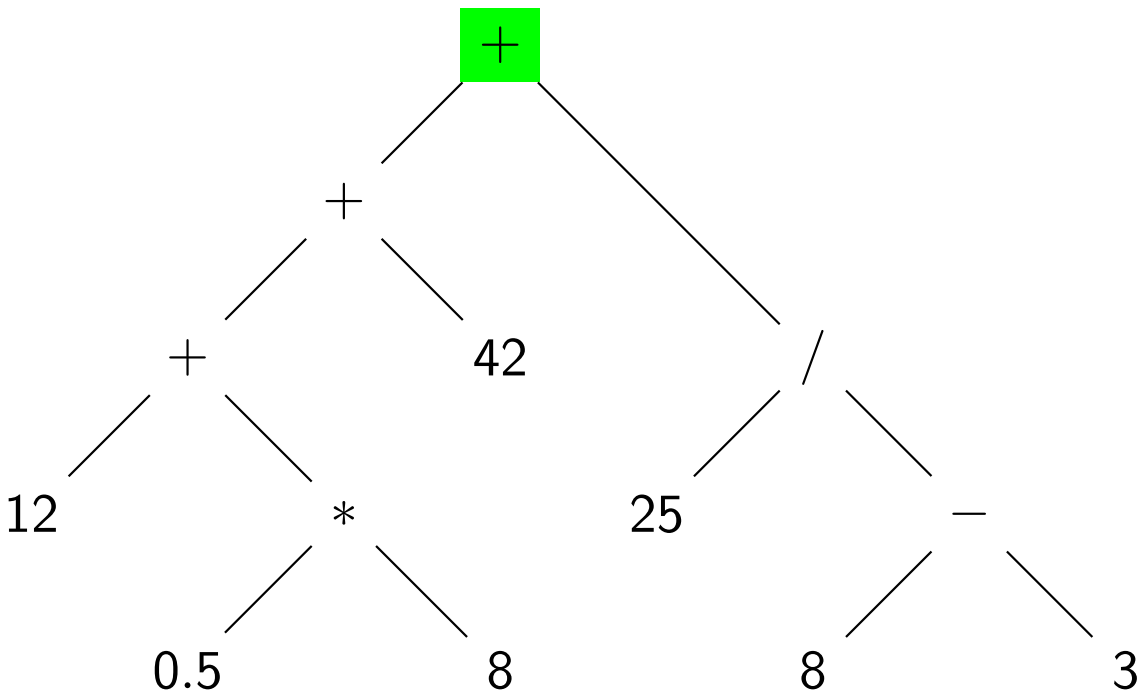
# From AST to assembly : Postfixe encoding !




## Produced code

**CsteNb** 12
**CsteNb** 0.5
**CsteNb** 8
**MultNb**
**AddiNb**
**CsteNb** 42
**AddiNb**
**CsteNb** 25
**CsteNb** 8
**CsteNb** 3
**SubsNb**
**DiviNb**

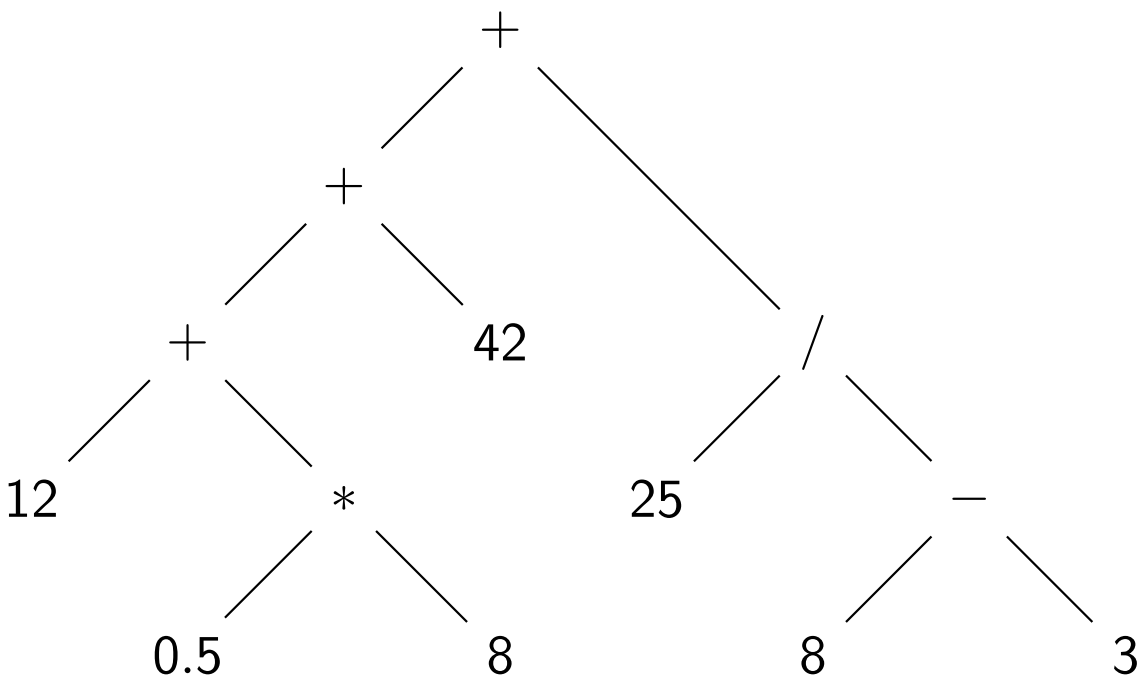
# From AST to assembly : Postfixe encoding !




## Produced code

**CsteNb** 12
**CsteNb** 0.5
**CsteNb** 8
**MultNb**
**AddiNb**
**CsteNb** 42
**AddiNb**
**CsteNb** 25
**CsteNb** 8
**CsteNb** 3
**SubsNb**
**DiviNb**
**AddiNb**

# From AST to assembly : Postfixe encoding !




## Produced code

```
CsteNb 12
  CsteNb 0.5
  CsteNb 8
  MultNb
AddiNb
  CsteNb 42
AddiNb
  CsteNb 25
    CsteNb 8
    CsteNb 3
  SubsNb
DiviNb
AddiNb
```

**You can indent it**

# Booleans : exactly the same

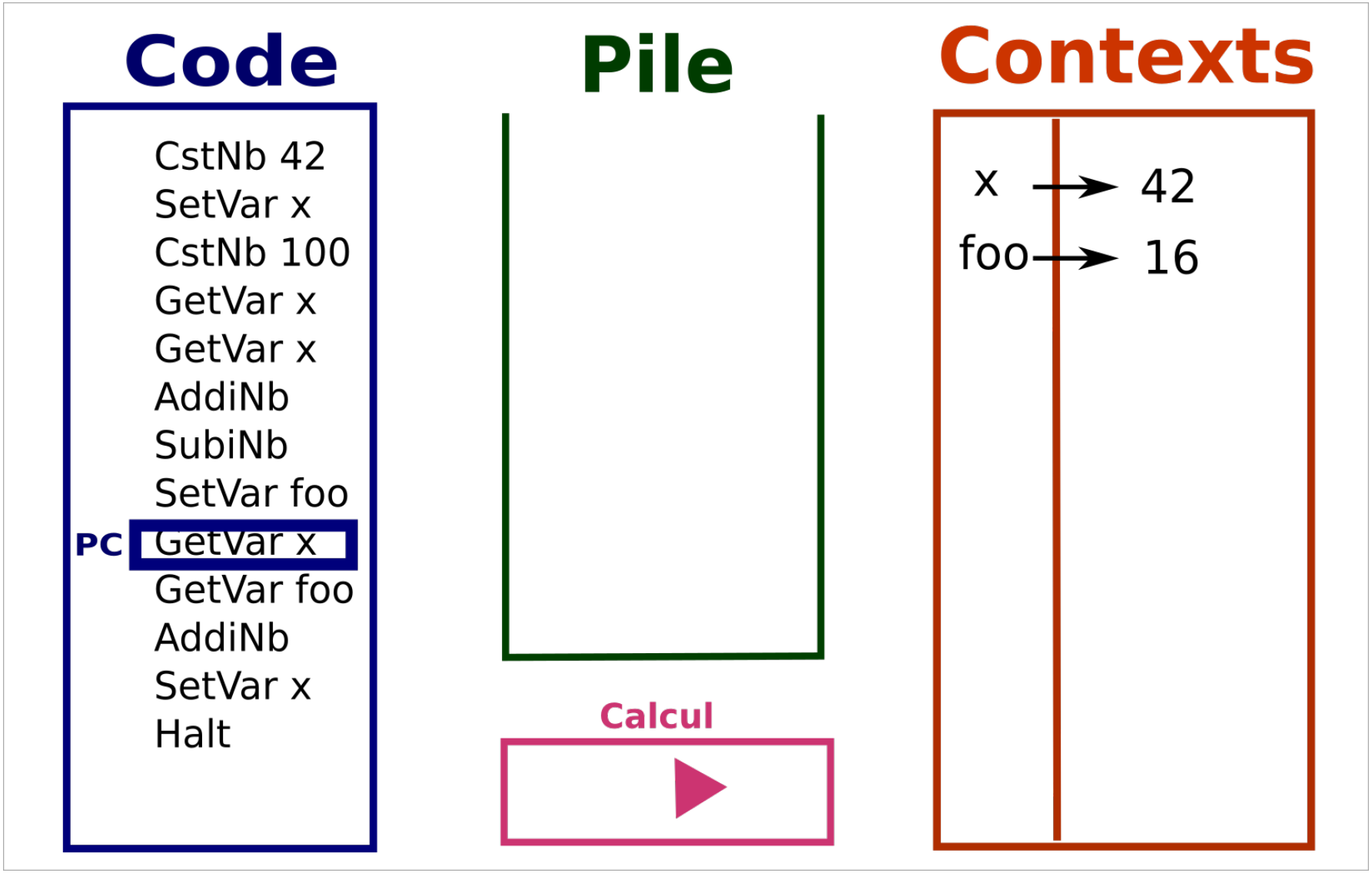
| Instruction | semantics | stack before | after |
|---|---|---|---|
| **CsteBo** True | Push(True); | stk | b:stk |
| **CsteBo** False | Push(False); | stk | b:stk |
| **Not** | Push(not(Pop)); | $b_1$:stk | $b_2$:stk |
| **Equals** | Push(Pop $===$ Pop); | $v_1$:$v_2$:stk | b:stk |
| **NotEql** | Push(Pop $\neq$ Pop); | $v_1$:$v_2$:stk | b:stk |
| **LoEqNb** | Push(Pop $\leq_f$ Pop); | $n_1$:$n_2$:stk | b:stk |
| **GrEqNb** | Push(Pop $\geq_f$ Pop); | $n_1$:$n_2$:stk | b:stk |
| **LoStNb** | Push(Pop $<_f$ Pop); | $n_1$:$n_2$:stk | b:stk |
| **GrStNb** | Push(Pop $>_f$ Pop); | $n_1$:$n_2$:stk | b:stk |

**Warning :** No booleans OR nor AND operations

## Respect types

The type of a boolean b and an integer n are different.
In assembly, a type mistake results in an indefinite behaviour.
Example : **LoEqNb** undefined on the stack [True, 3].
Exception : **Equals** and **NotEql** always defined.

# A context for variables



# Context (naive)

## Dictionary mapping identifier to their values

Two pseudo-code operands :

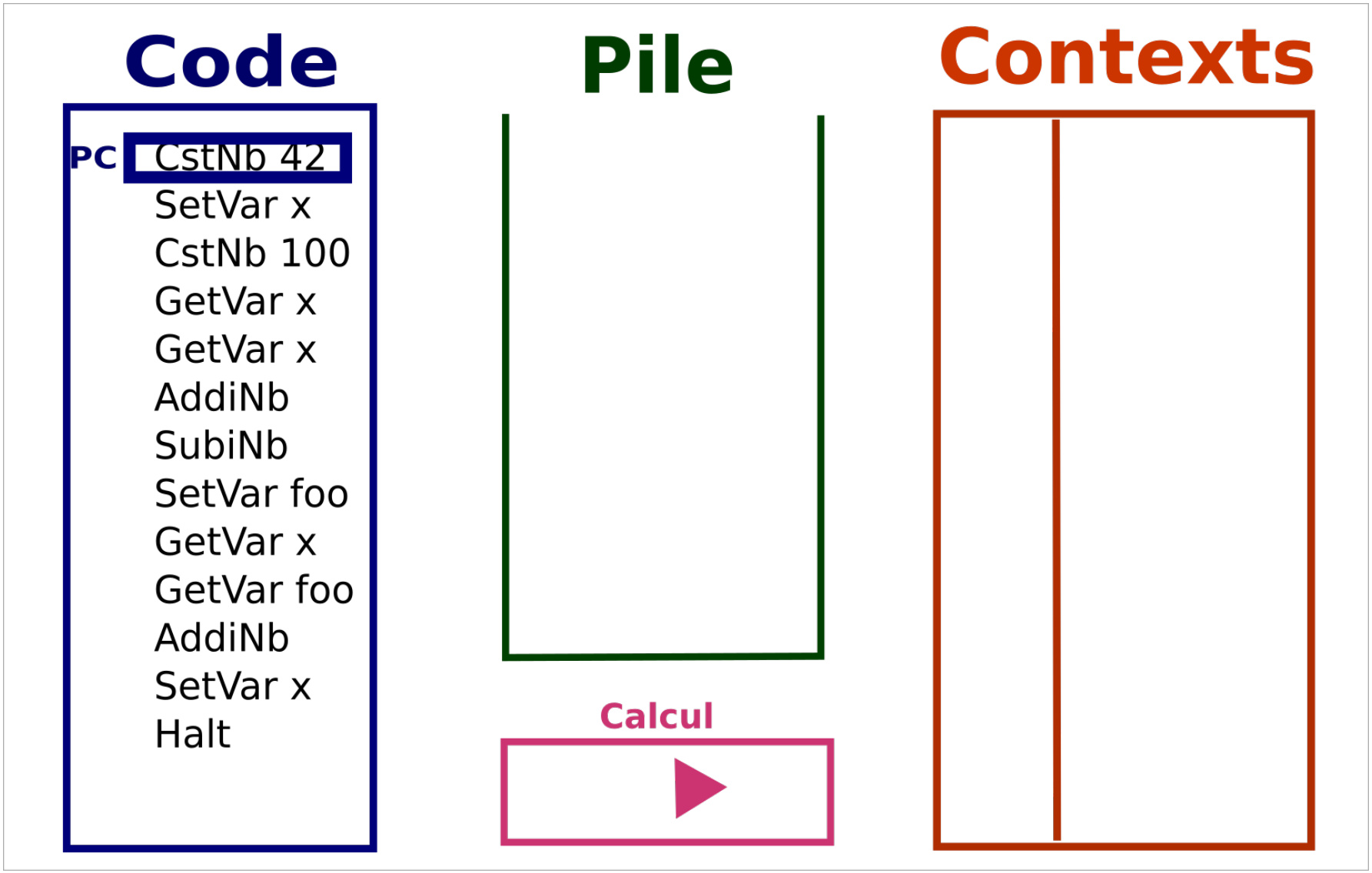
- **Set**(`id,val`) : maps the variable name `id` to the value `val` in the context.
- **Get**(`id`) : returns the value associated to the variable `id` in the context.

## CC : name of the context

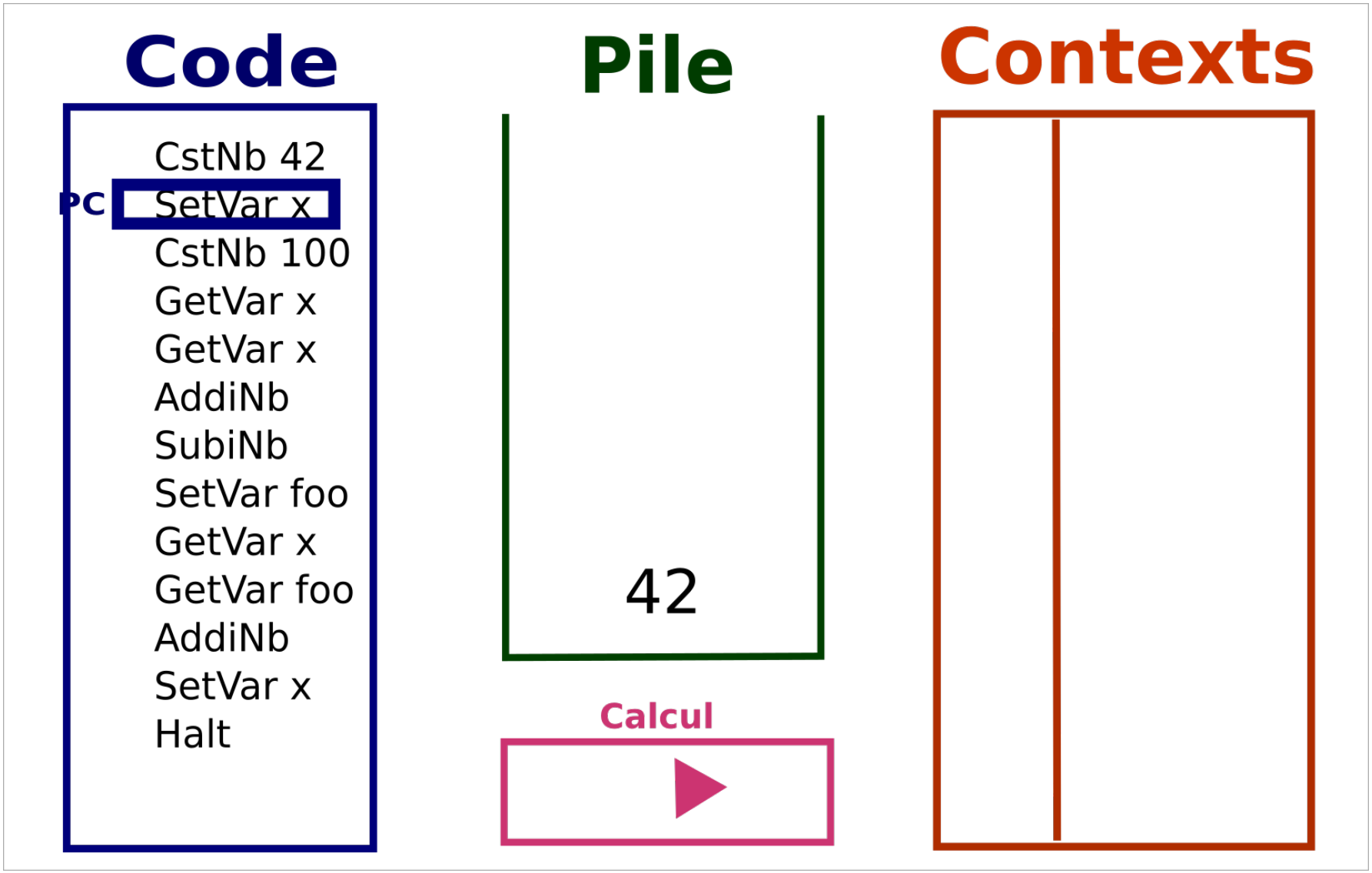
Latter we will use several contexts; **CC**, means "current context".

| Instruction | meaning | semantics | stack before | after |
|---|---|---|---|---|
| **SetVar n** | modify/create **n** | `Set(n,Pop);` | `v:stk` | `stk` |
| **GetVar n** | push the value of **n** | `Push(Get(n))` | `stk` | `v:stk` |

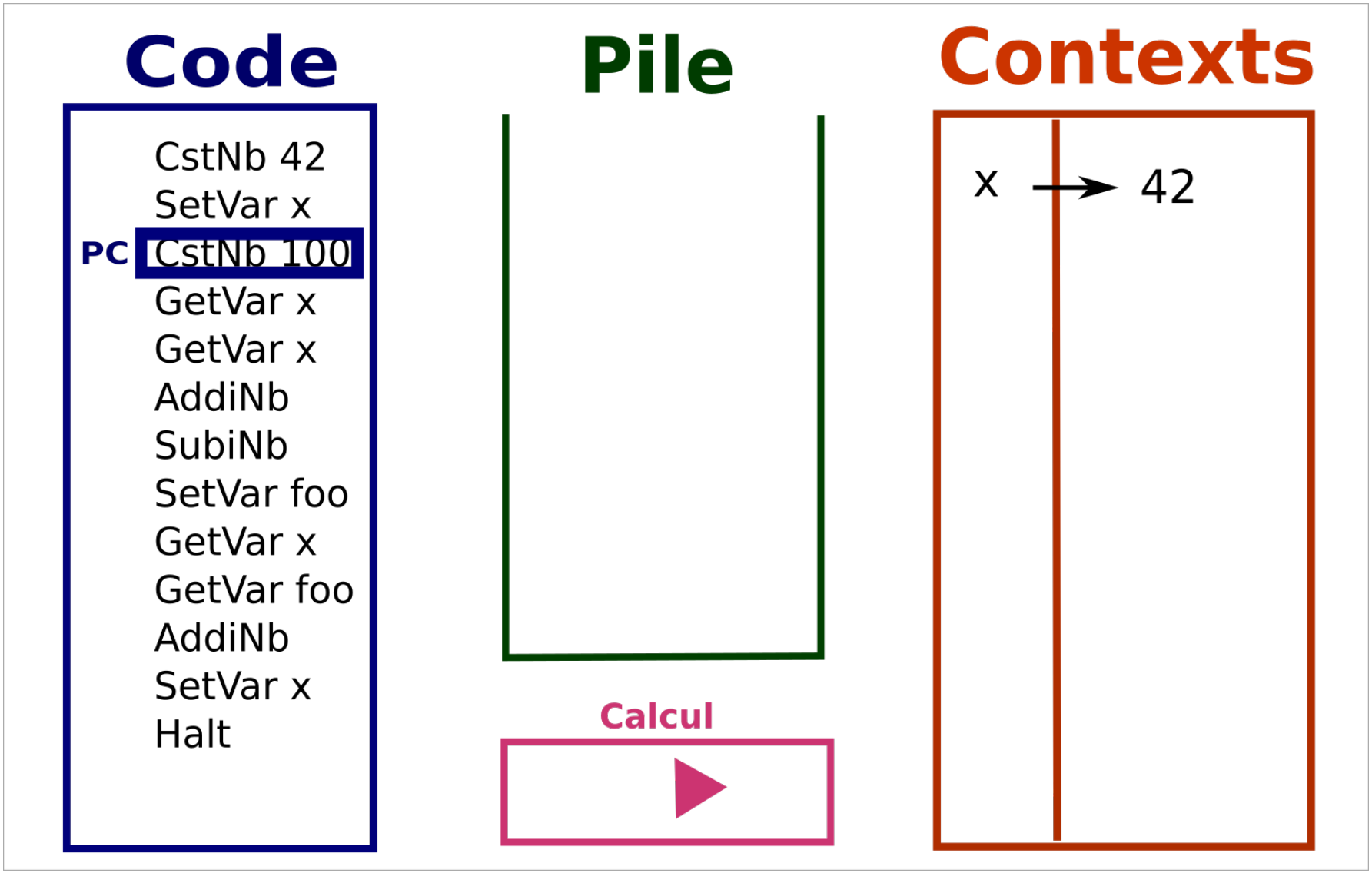
# Example


Code
PC
CstNb 42
SetVar x
CstNb 100
GetVar x
GetVar x
AddiNb
SubiNb
SetVar foo
GetVar x
GetVar foo
AddiNb
SetVar x
Halt
Pile
Calcul
Contexts

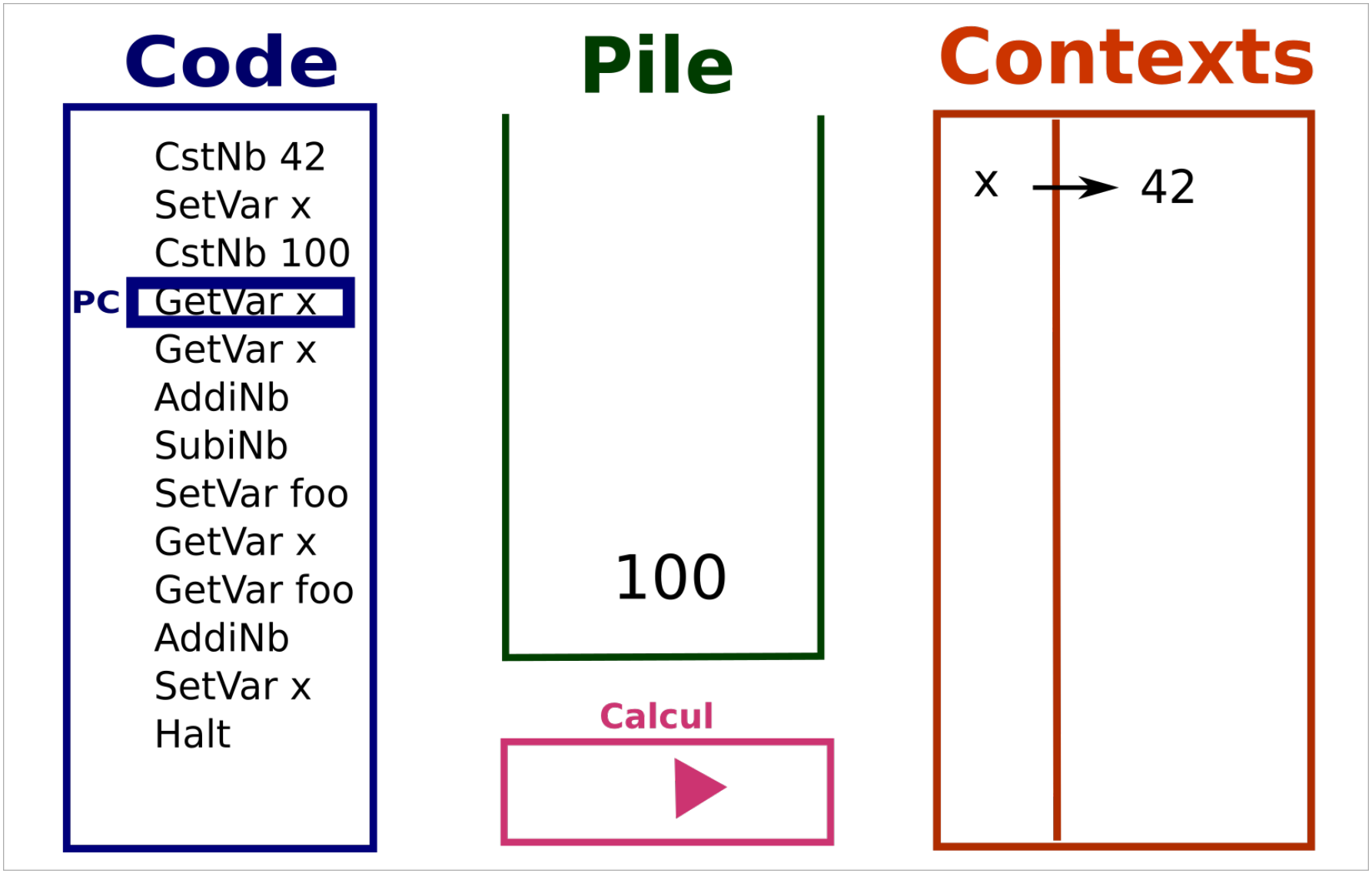
# Example


Code
CstNb 42
PC
SetVar x
CstNb 100
GetVar x
GetVar x
AddiNb
SubiNb
SetVar foo
GetVar x
GetVar foo
AddiNb
SetVar x
Halt
Pile
42
Calcul
Contexts

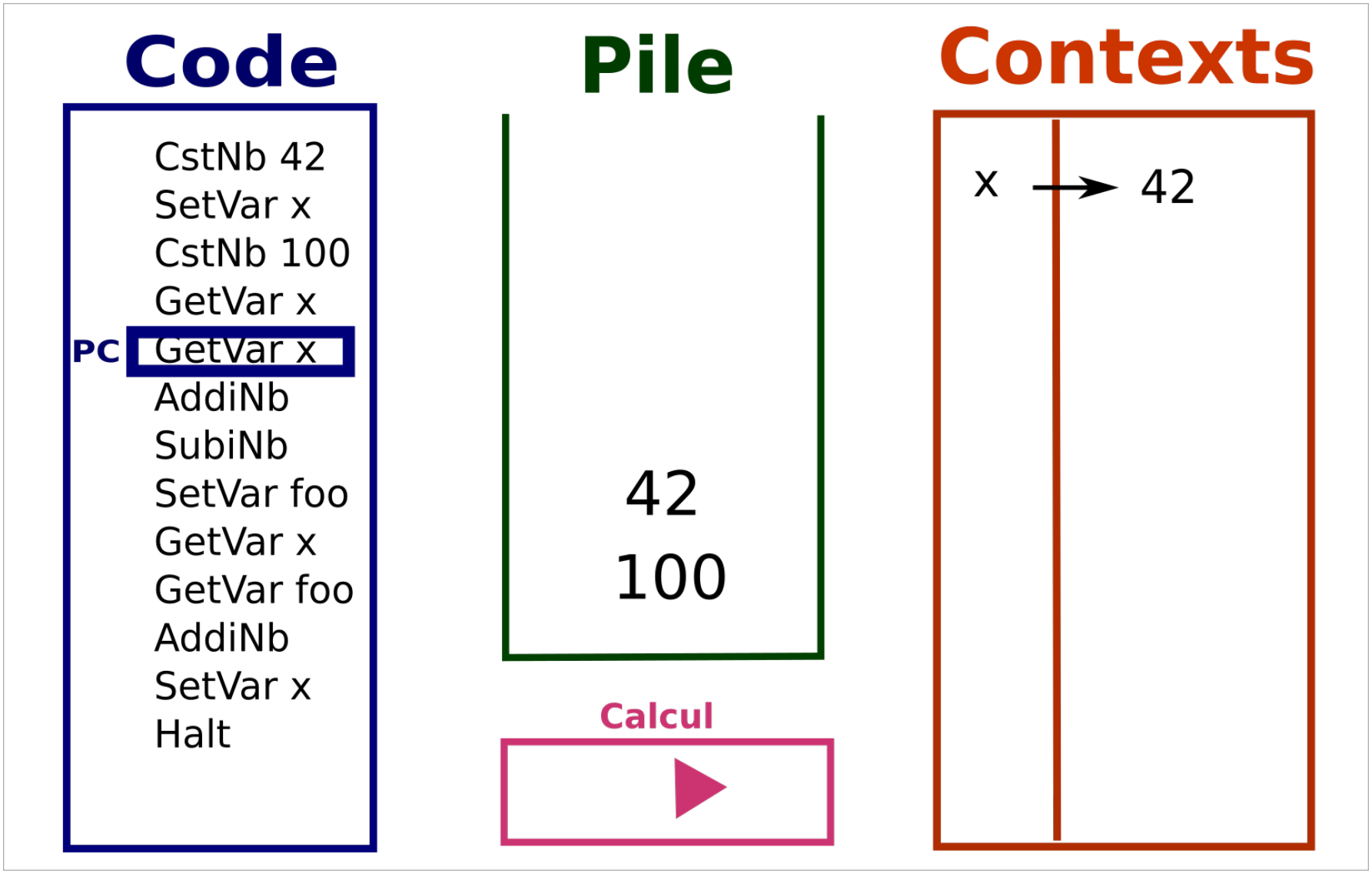
# Example


Code
CstNb 42
SetVar x
PC
CstNb 100
GetVar x
GetVar x
AddiNb
SubiNb
SetVar foo
GetVar x
GetVar foo
AddiNb
SetVar x
Halt
Pile
Calcul
Contexts
x
42

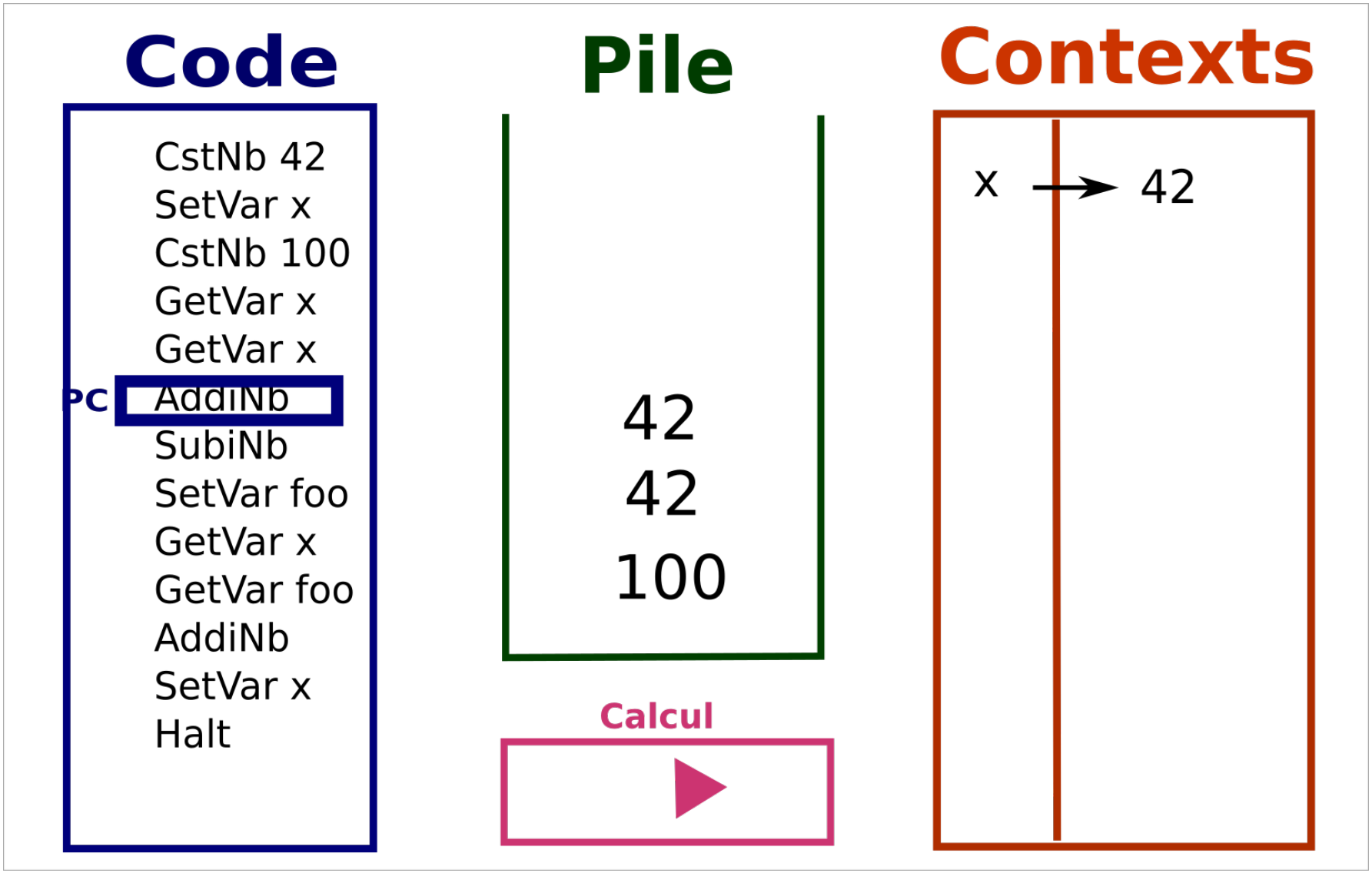
# Example


Code
CstNb 42
SetVar x
CstNb 100
PC
GetVar x
GetVar x
AddiNb
SubiNb
SetVar foo
GetVar x
GetVar foo
AddiNb
SetVar x
Halt
Pile
100
Calcul
Contexts
x
42

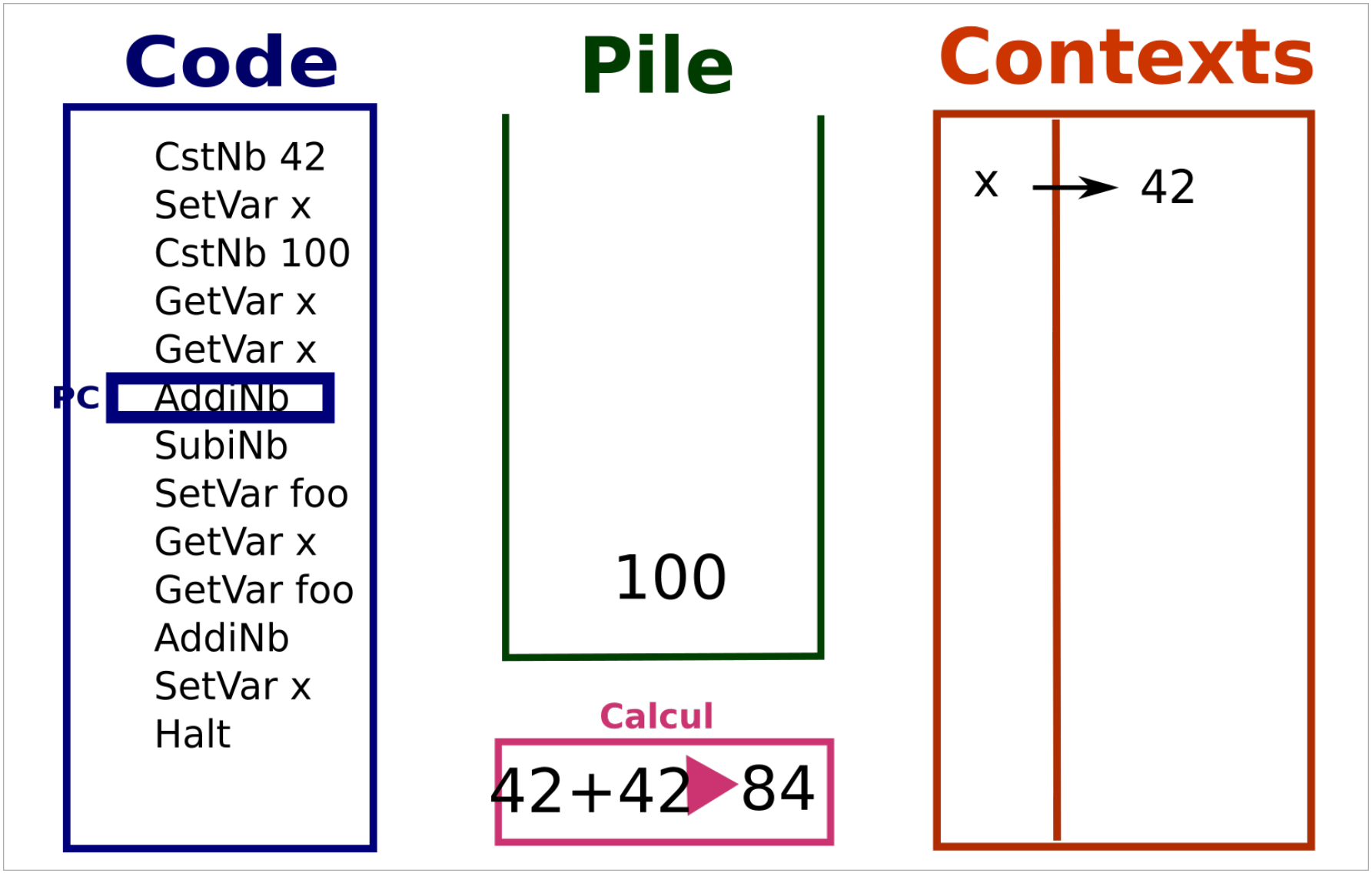
# Example


Code
CstNb 42
SetVar x
CstNb 100
GetVar x
PC
GetVar x
AddiNb
SubiNb
SetVar foo
GetVar x
GetVar foo
AddiNb
SetVar x
Halt
Pile
42
100
Calcul
Contexts
x
42

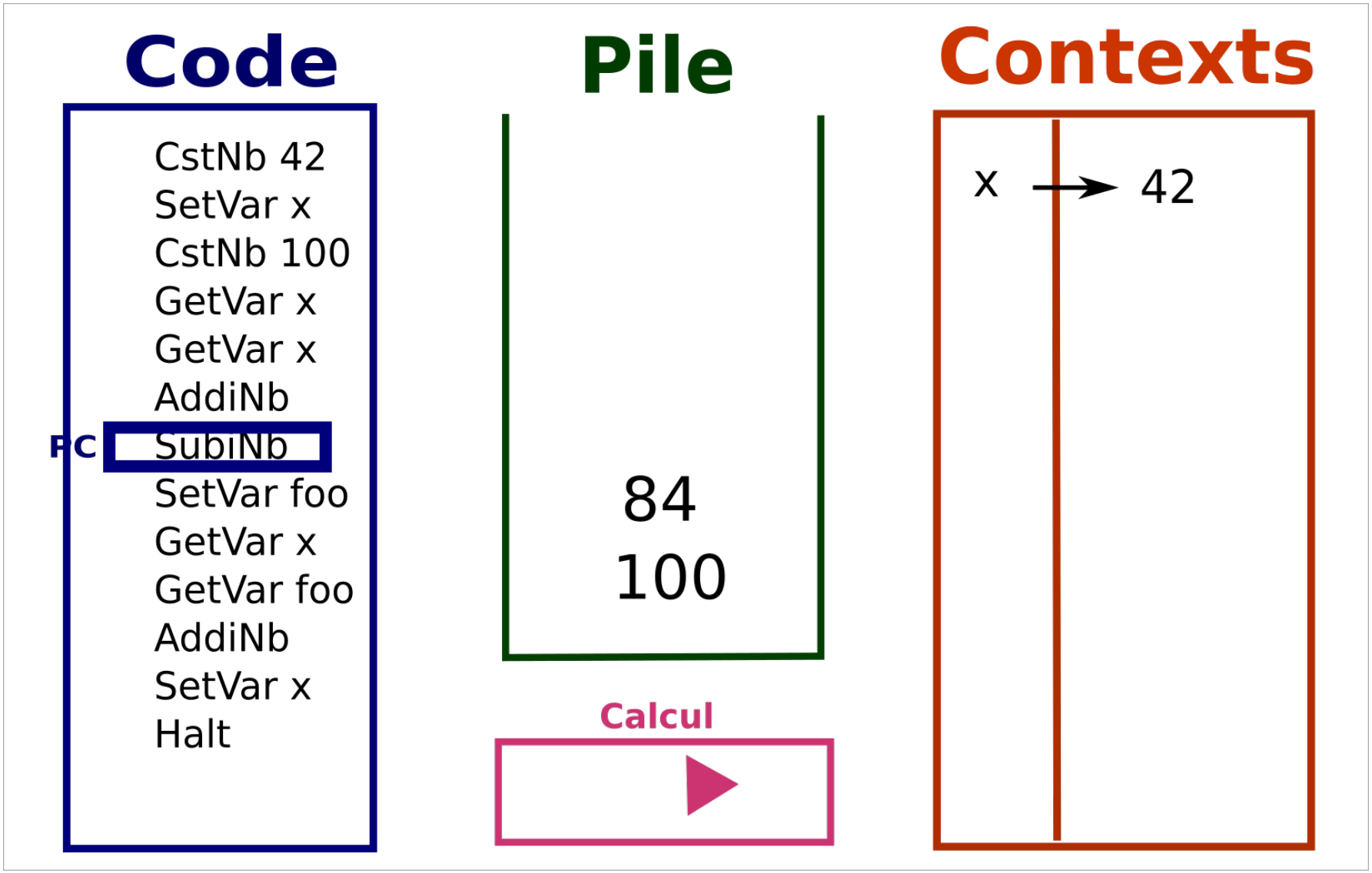
# Example


Code
CstNb 42
SetVar x
CstNb 100
GetVar x
GetVar x
PC
AddiNb
SubiNb
SetVar foo
GetVar x
GetVar foo
AddiNb
SetVar x
Halt
Pile
42
42
100
Calcul
Contexts
x
42

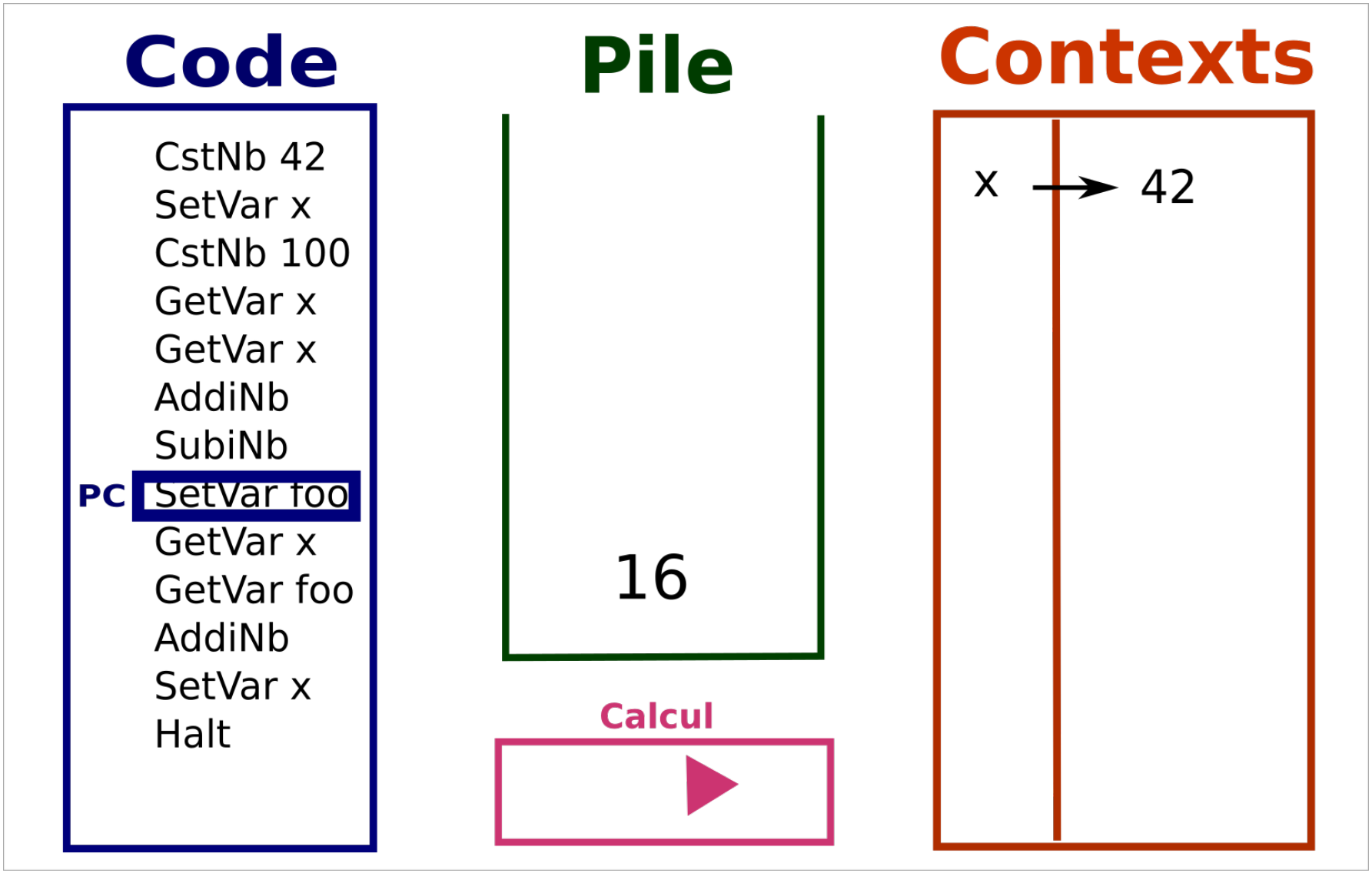
# Example


Code
CstNb 42
SetVar x
CstNb 100
GetVar x
GetVar x
PC
AddiNb
SubiNb
SetVar foo
GetVar x
GetVar foo
AddiNb
SetVar x
Halt
Pile
100
Calcul
42+42
84
Contexts
x
42

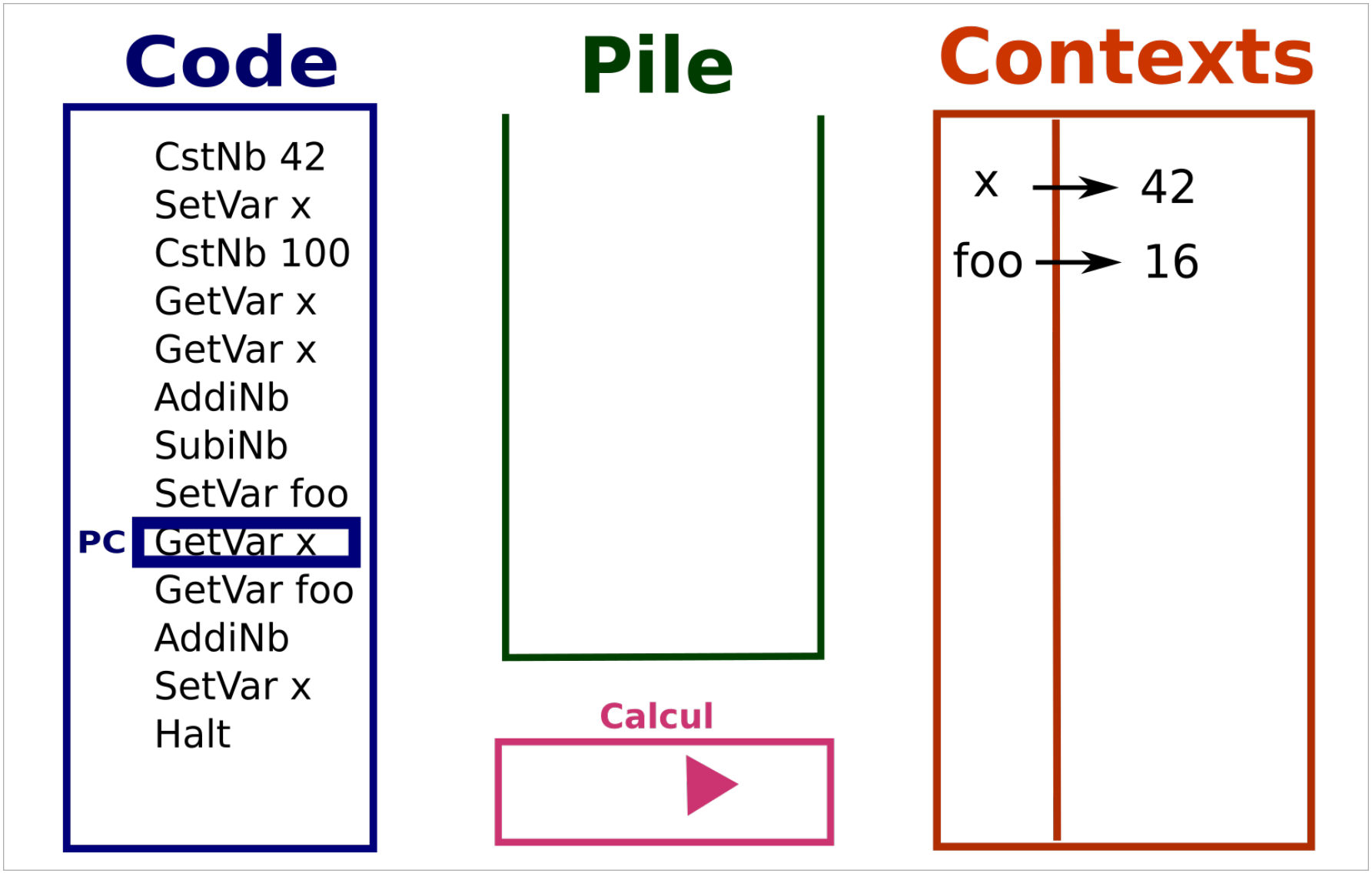
# Example


Code
CstNb 42
SetVar x
CstNb 100
GetVar x
GetVar x
AddiNb
PC
SubiNb
SetVar foo
GetVar x
GetVar foo
AddiNb
SetVar x
Halt
Pile
84
100
Calcul
Contexts
x
42

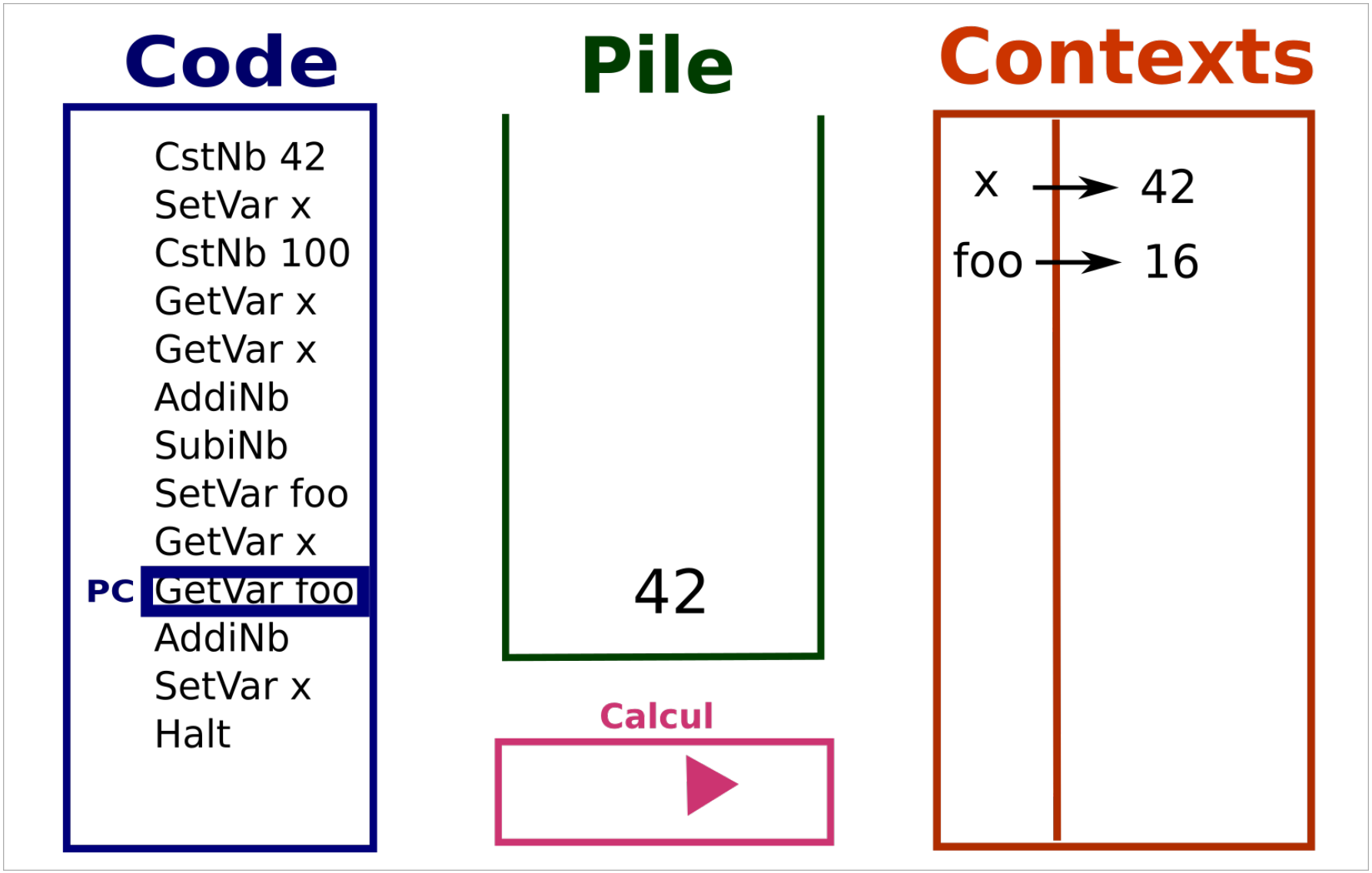
# Example


Code
CstNb 42
SetVar x
CstNb 100
GetVar x
GetVar x
AddiNb
SubiNb
PC SetVar foo
GetVar x
GetVar foo
AddiNb
SetVar x
Halt
Pile
16
Calcul
Contexts
x
42

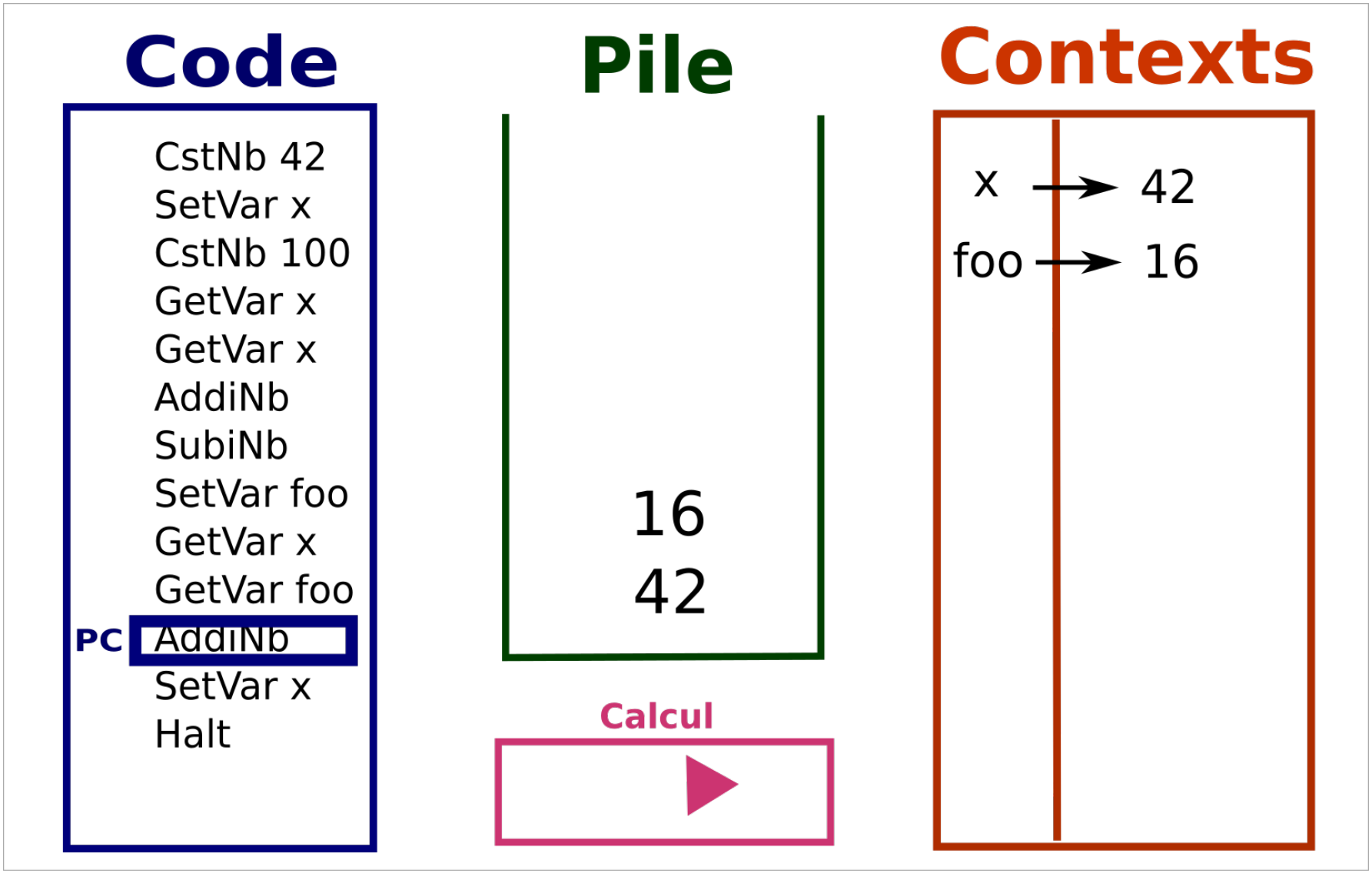
# Example


Code
CstNb 42
SetVar x
CstNb 100
GetVar x
GetVar x
AddiNb
SubiNb
SetVar foo
PC
GetVar x
GetVar foo
AddiNb
SetVar x
Halt
Pile
Calcul
Contexts
x
42
foo
16

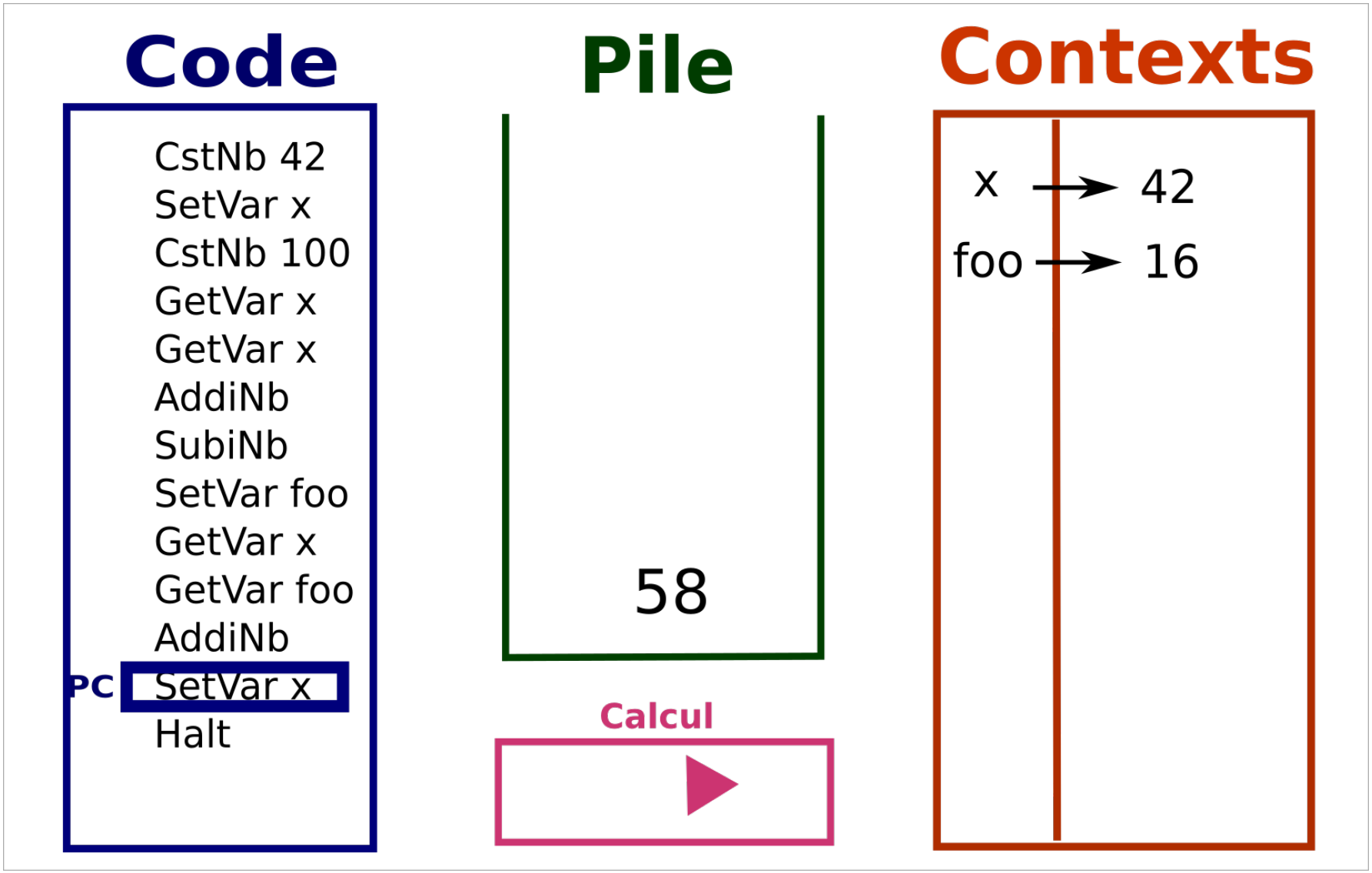
# Example


Code
CstNb 42
SetVar x
CstNb 100
GetVar x
GetVar x
AddiNb
SubiNb
SetVar foo
GetVar x
PC
GetVar foo
AddiNb
SetVar x
Halt
Pile
42
Calcul
Contexts
x
42
foo
16

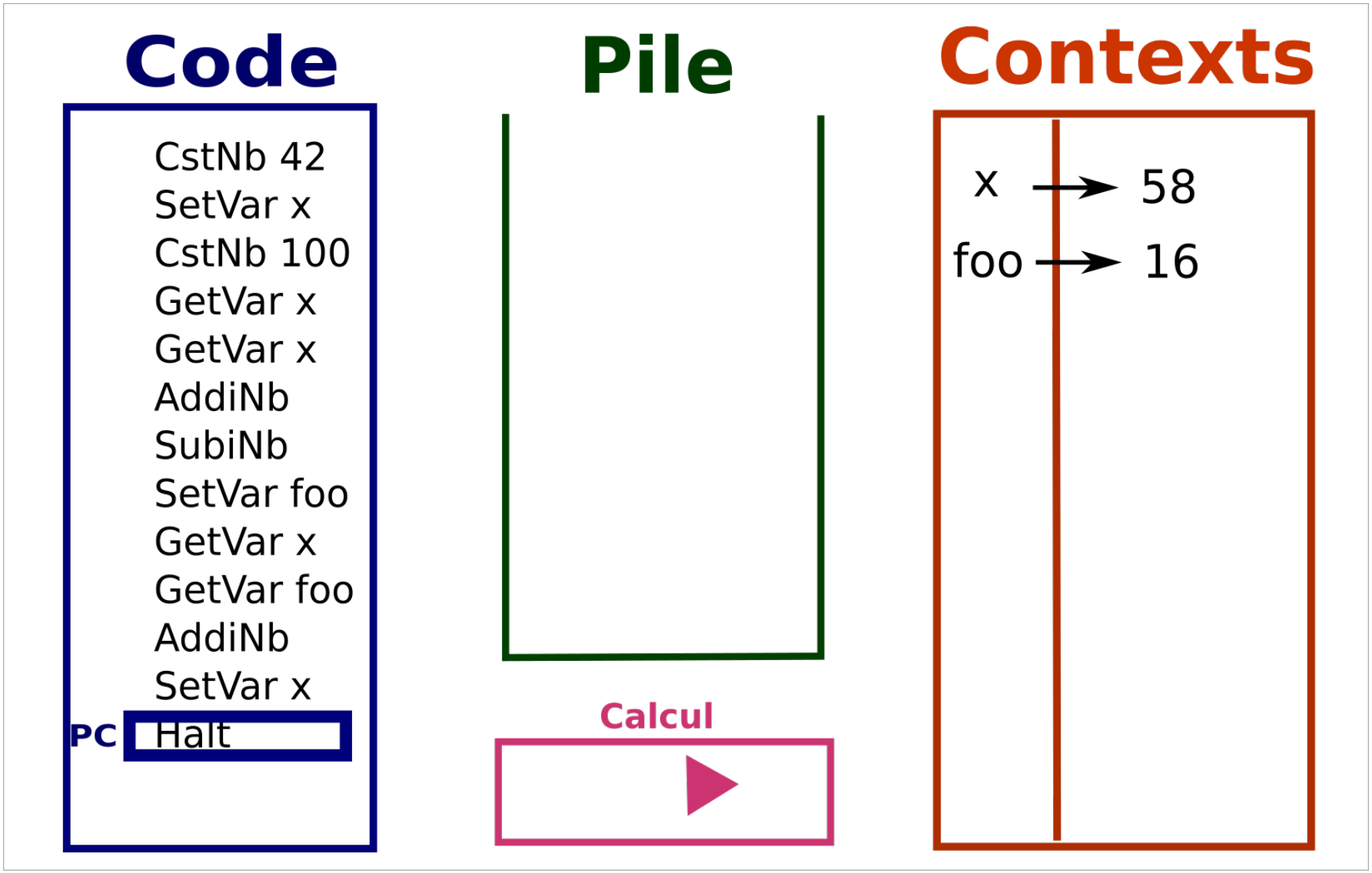
# Example


Code
CstNb 42
SetVar x
CstNb 100
GetVar x
GetVar x
AddiNb
SubiNb
SetVar foo
GetVar x
GetVar foo
PC
AddiNb
SetVar x
Halt
Pile
16
42
Calcul
Contexts
x
42
foo
16

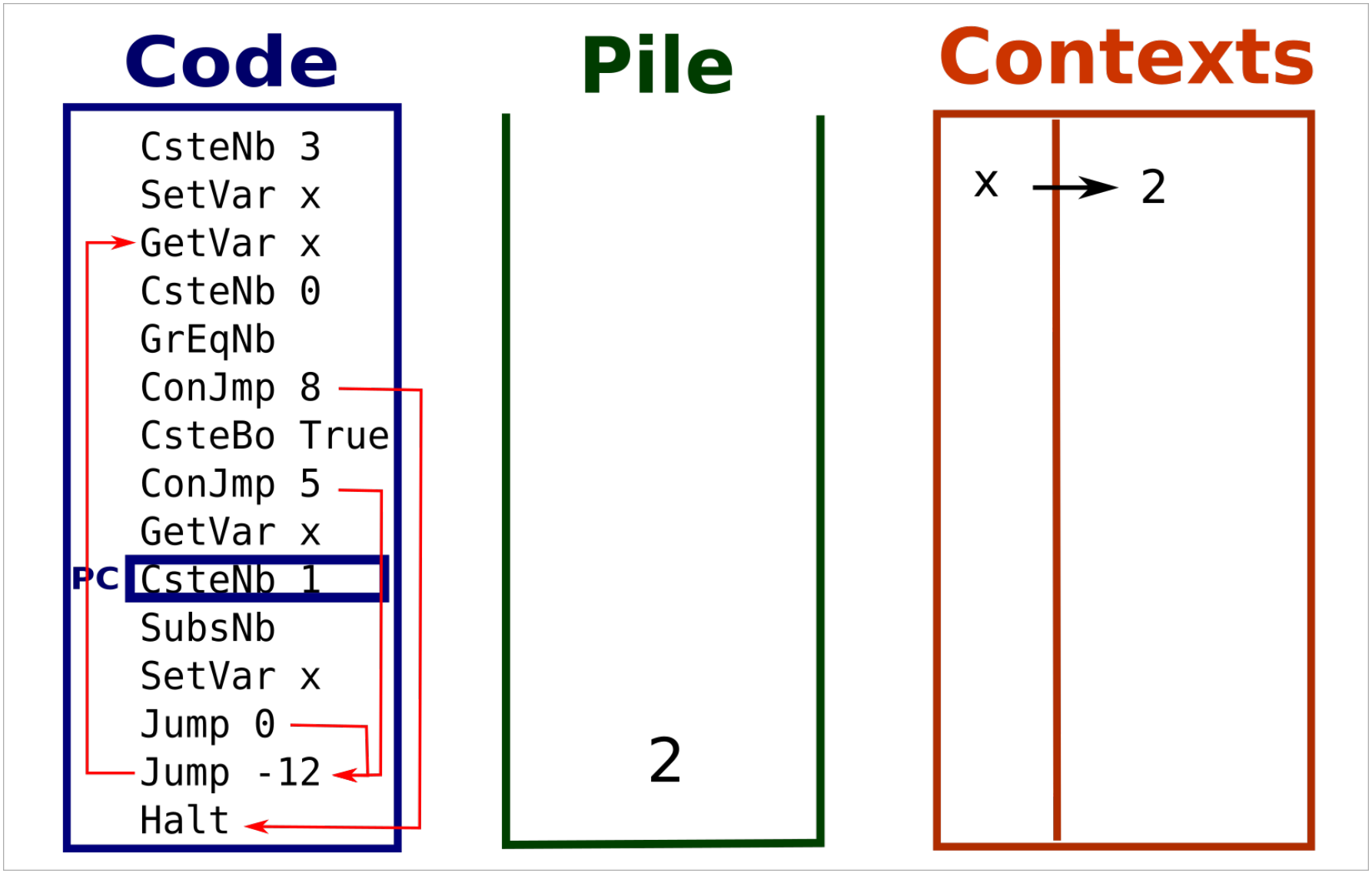
# Example


Code
CstNb 42
SetVar x
CstNb 100
GetVar x
GetVar x
AddiNb
SubiNb
SetVar foo
GetVar x
GetVar foo
AddiNb
PC
SetVar x
Halt
Pile
58
Calcul
Contexts
x
42
foo
16

# Example


Code
CstNb 42
SetVar x
CstNb 100
GetVar x
GetVar x
AddiNb
SubiNb
SetVar foo
GetVar x
GetVar foo
AddiNb
SetVar x
PC
Halt
Pile
Calcul
Contexts
x
58
foo
16

# Jumps : moving in the code



# Jumps : moving in the code

| Instruction | meaning | semantics | before | after |
|---|---|---|---|---|
| **Jump** offset | always jumps | `PC := PC + off + 1;` | `stk` | `stk` |
| **ConJmp** offset | may jump depending on stack's summit | if `Pop`<br>then `PC := PC + 1;`<br>else `PC := PC + off + 1;` | `b:stk` | `stk` |

## Offsets

Relative number of lines with respect to the next line

## Condition

**ConJmp** jumps only is the top of the stack is `False`

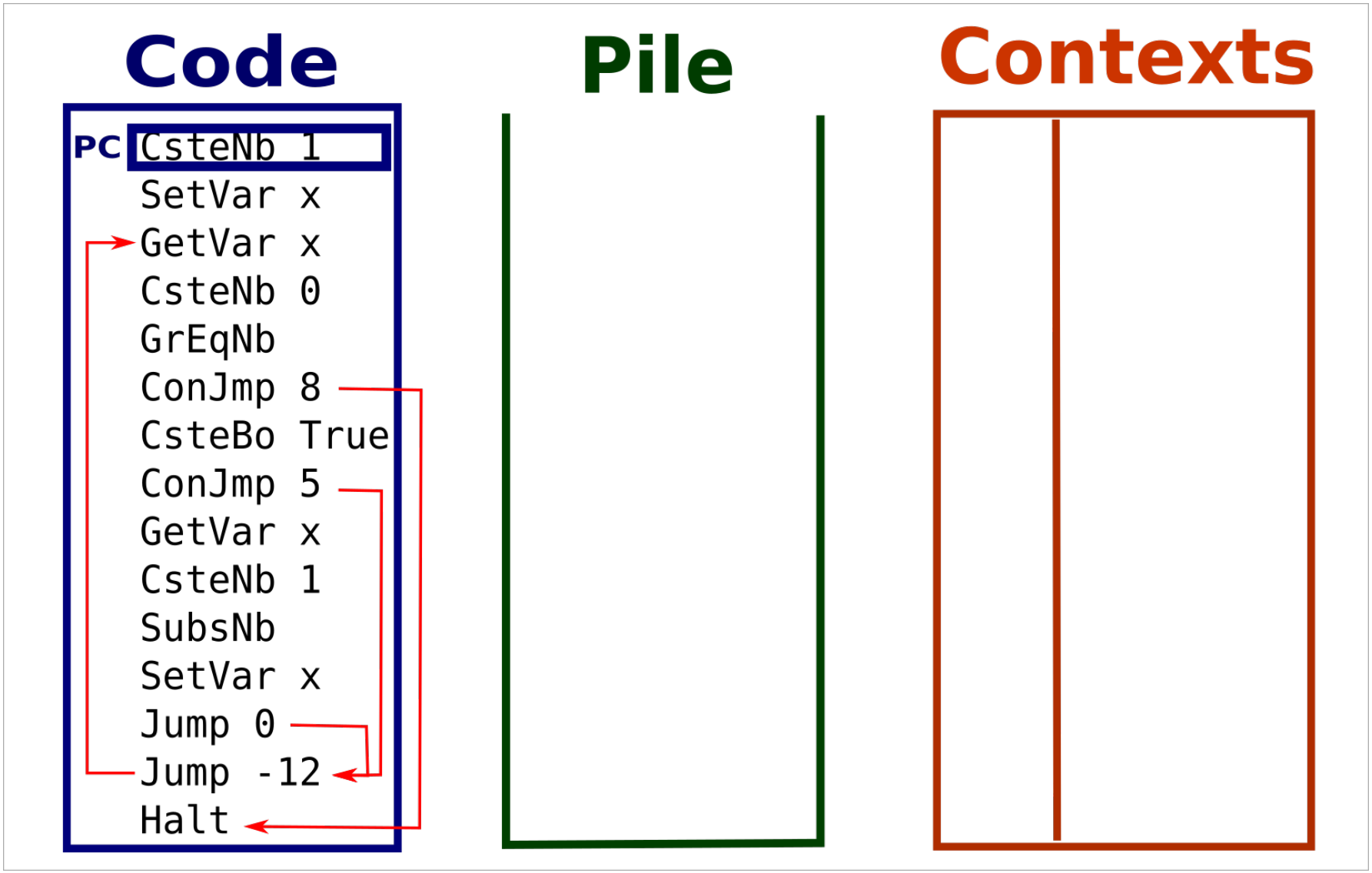
## In drawings

For readability, we draw the offsets as arrows.

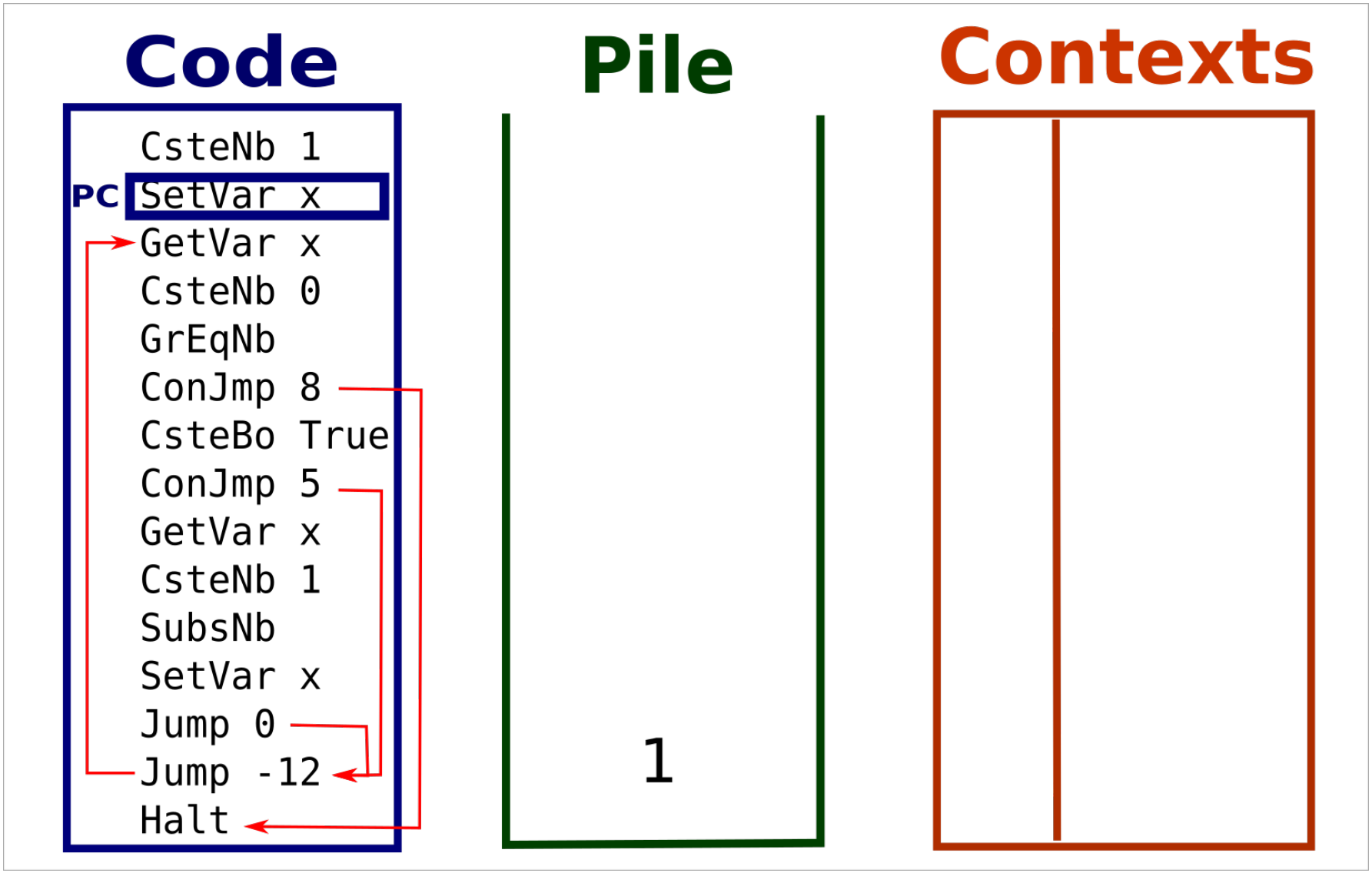
## Vocabulary

**PC :** Pointer to the actual instruction
**Offset :** relative reference to another instruction (cf jumps)
**Label :** static pointer to an instruction (see next slides)

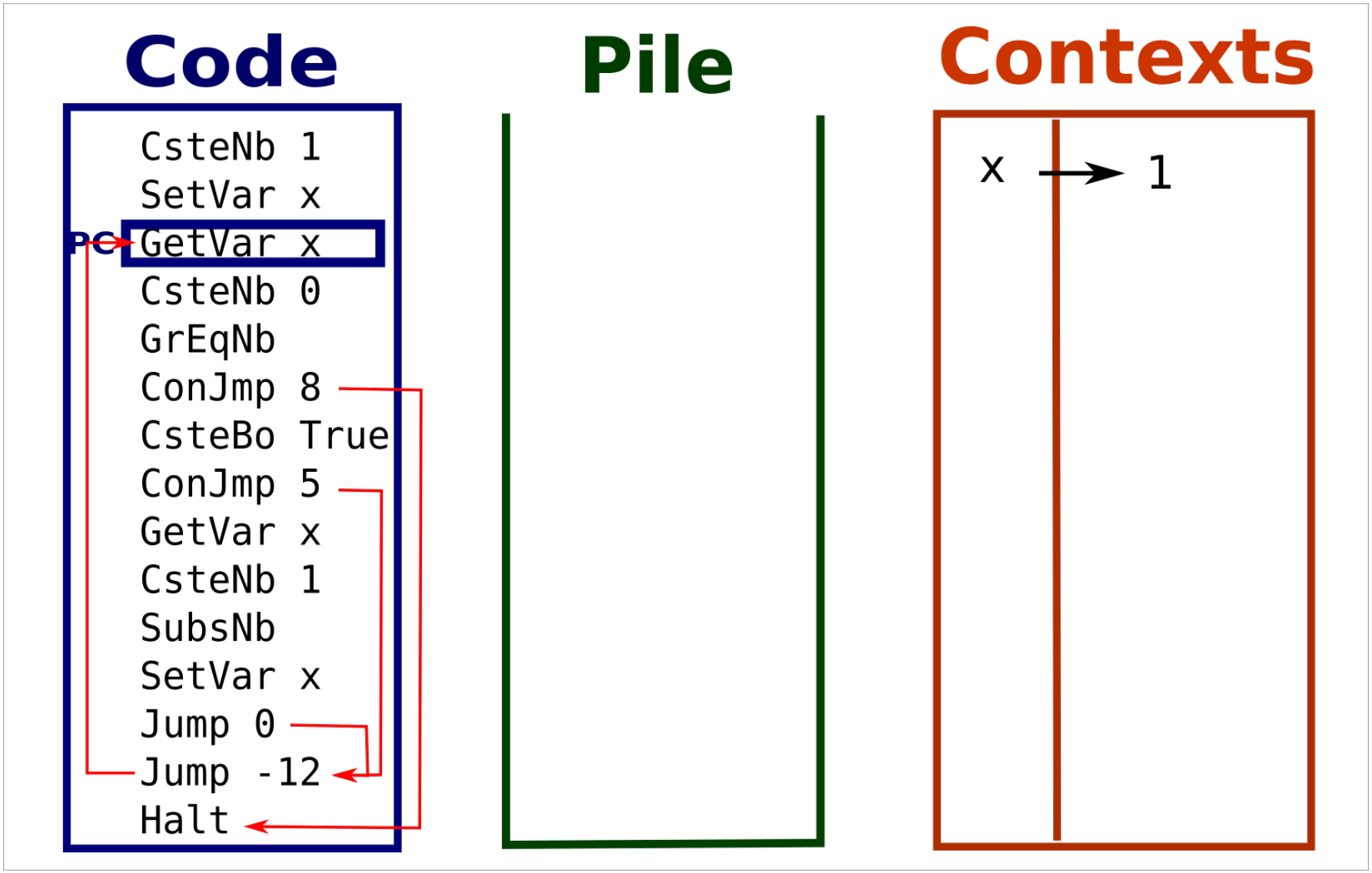
# Example en action



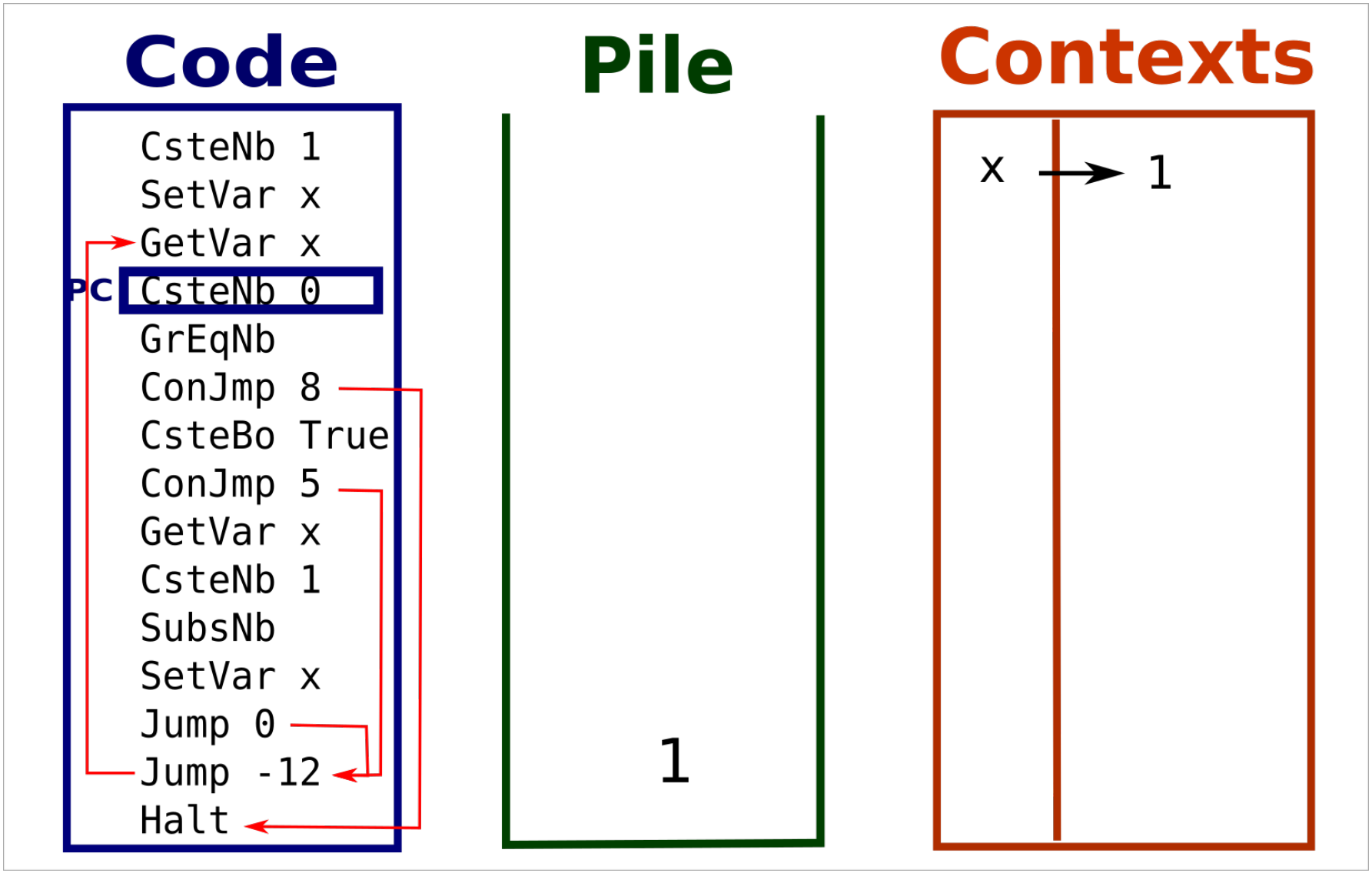
# Example en action



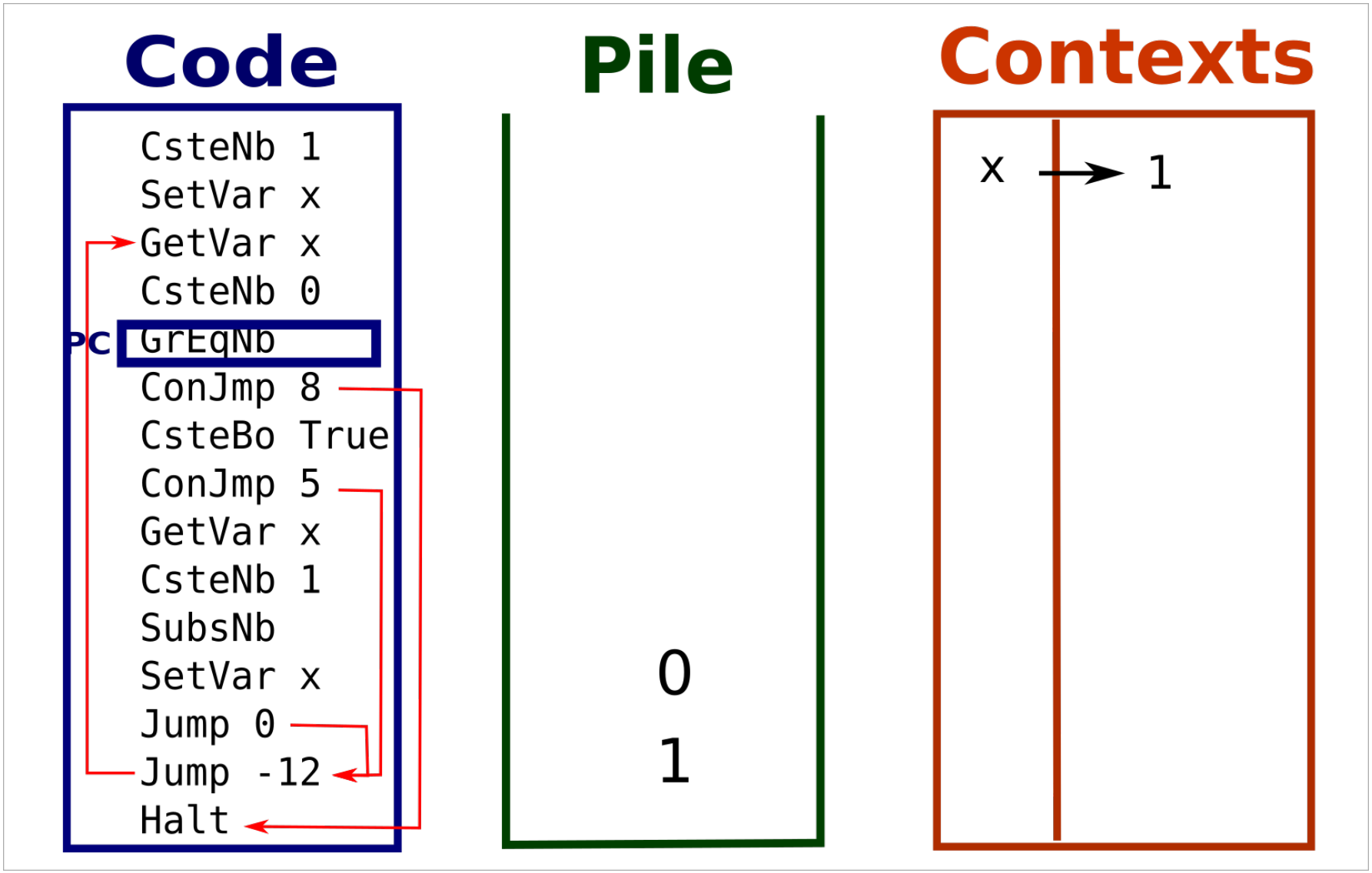
# Example en action



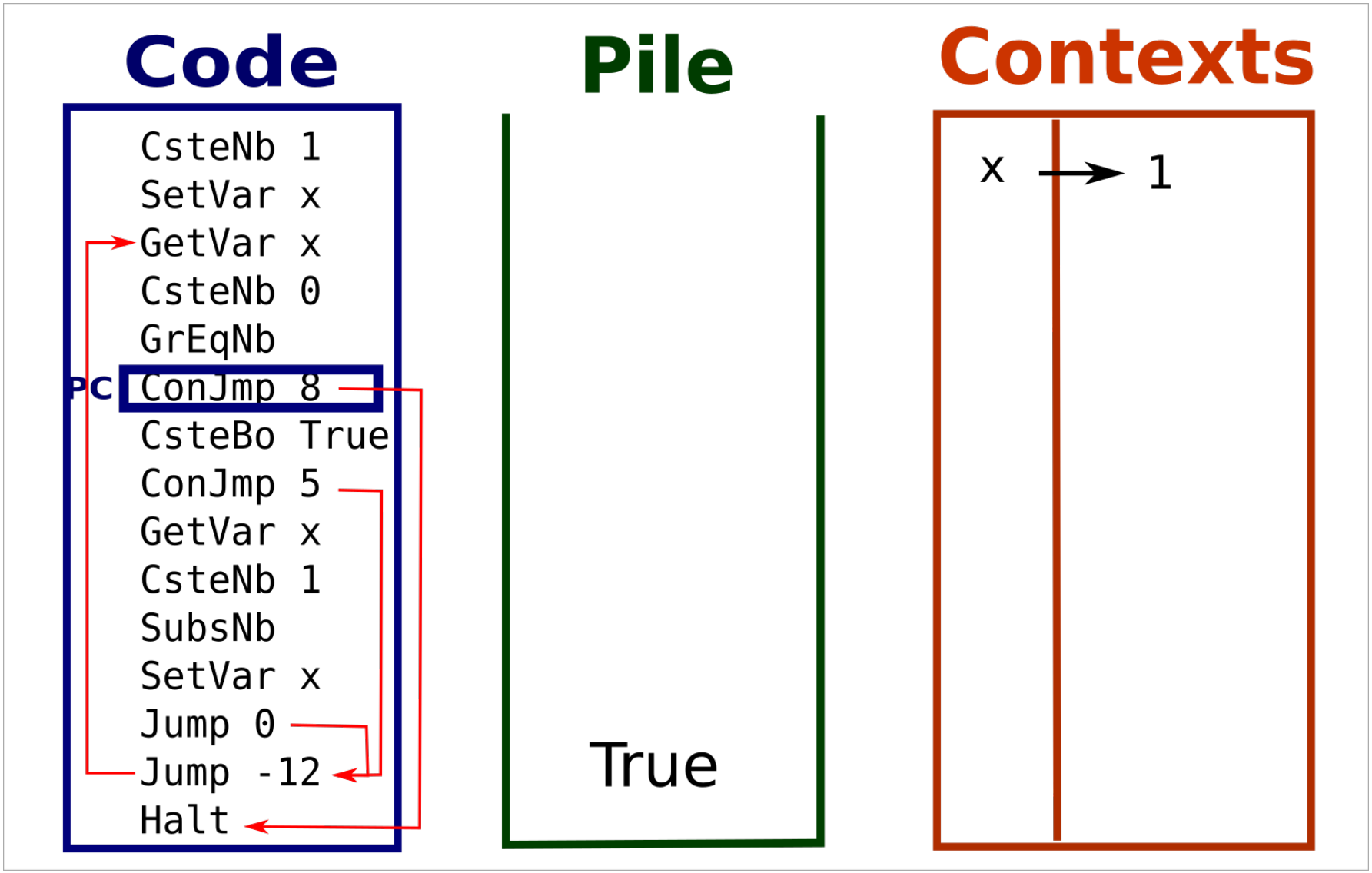
# Example en action



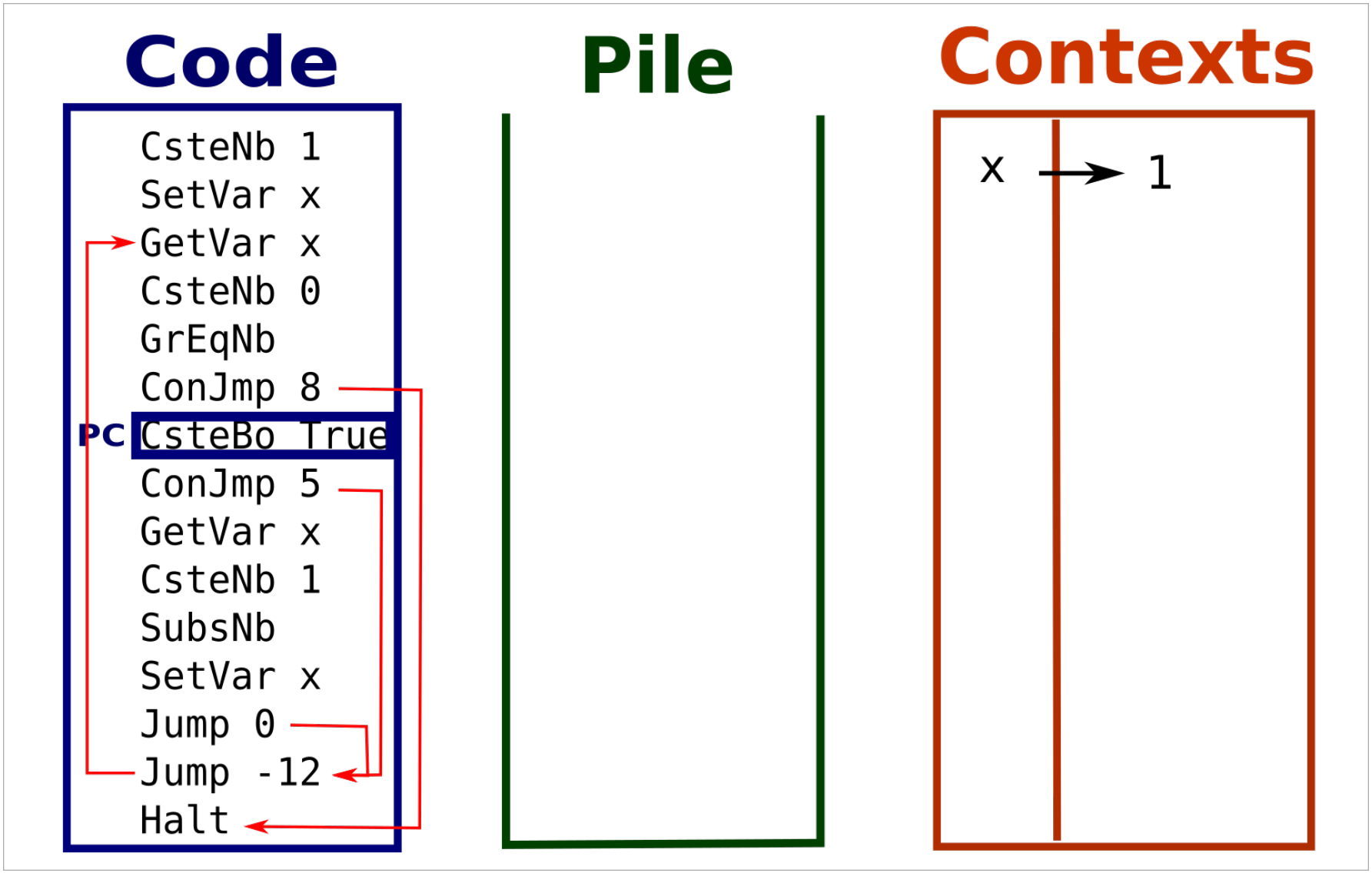
# Example en action



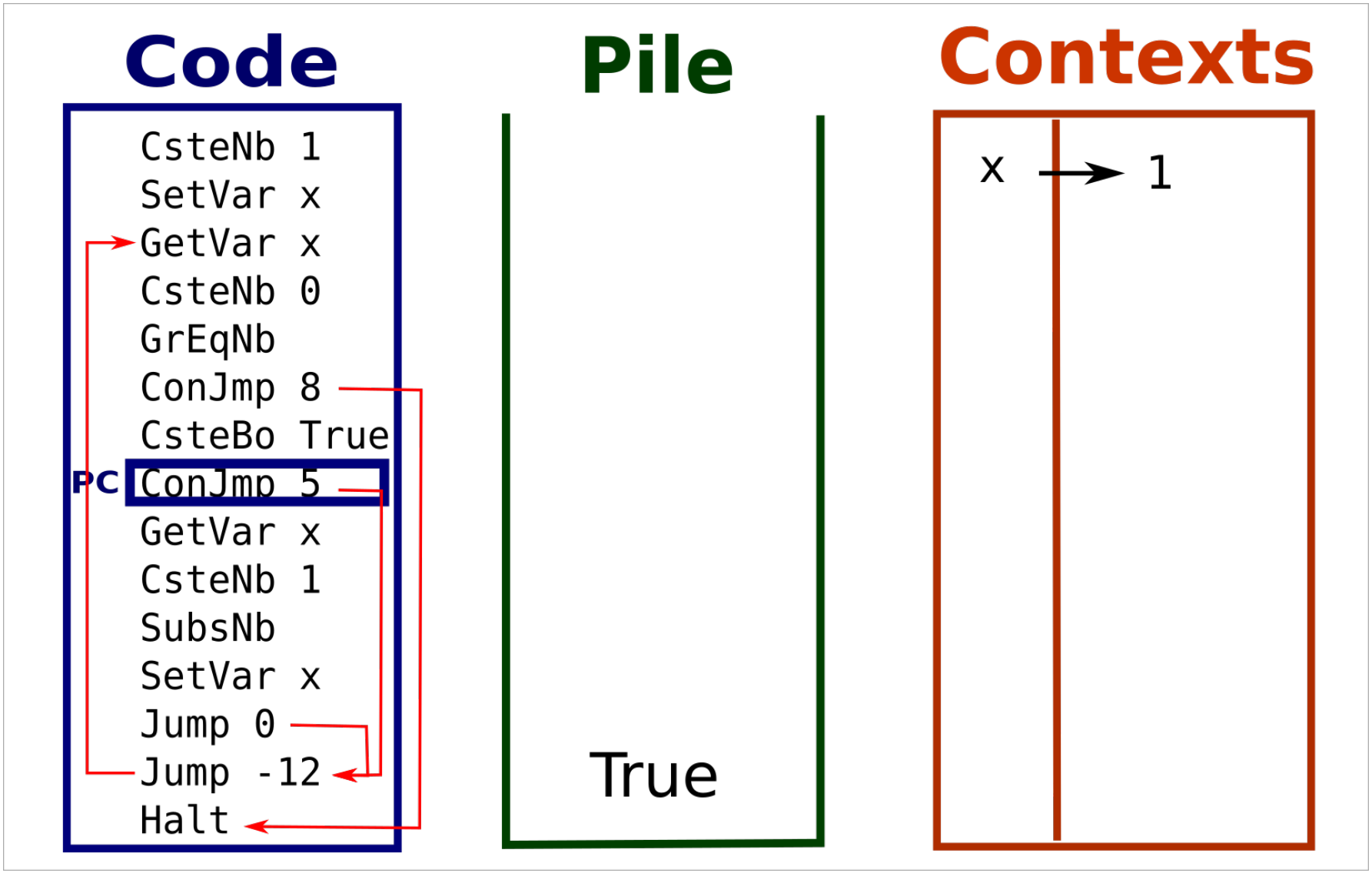
# Example en action



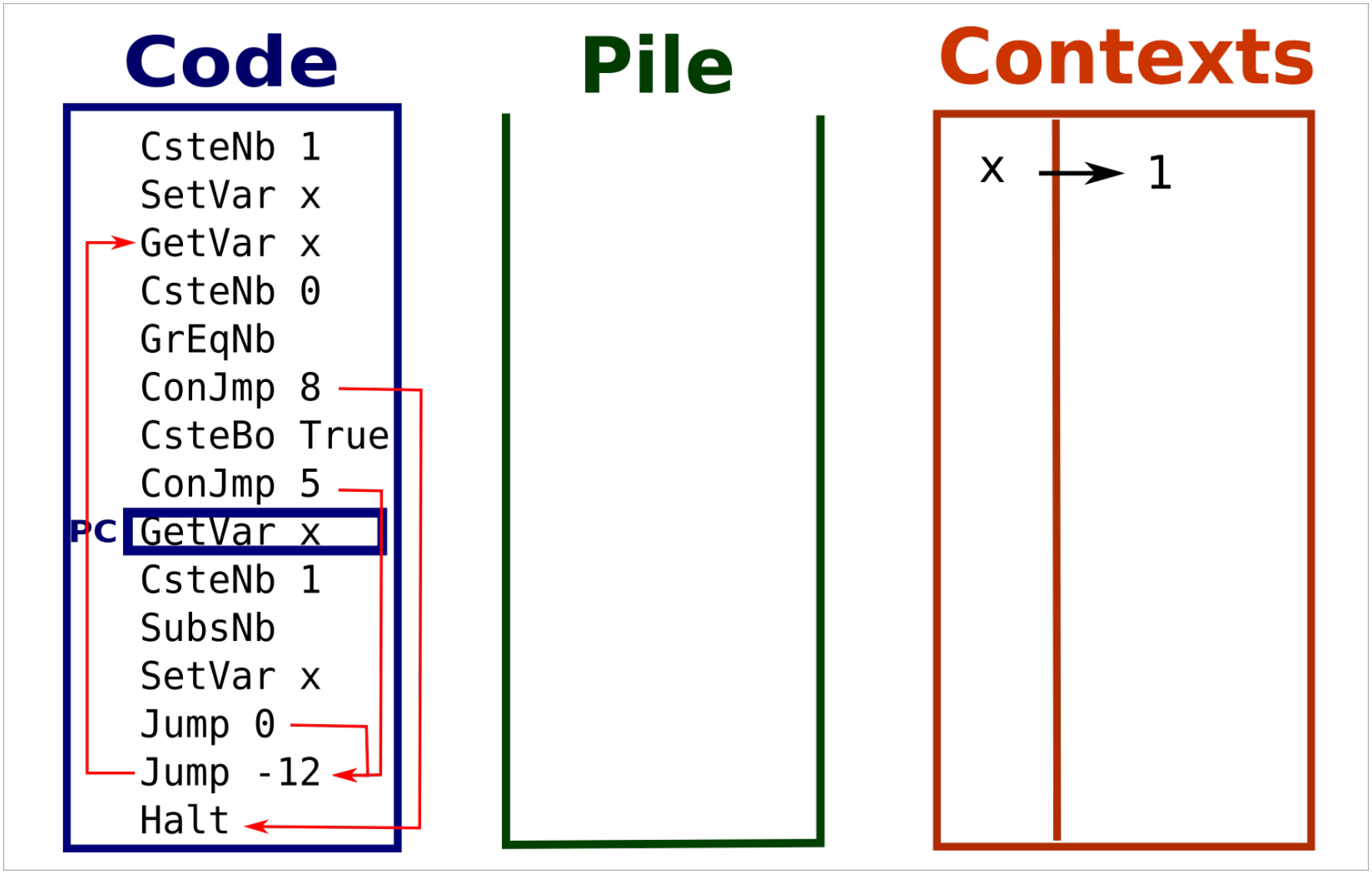
# Example en action



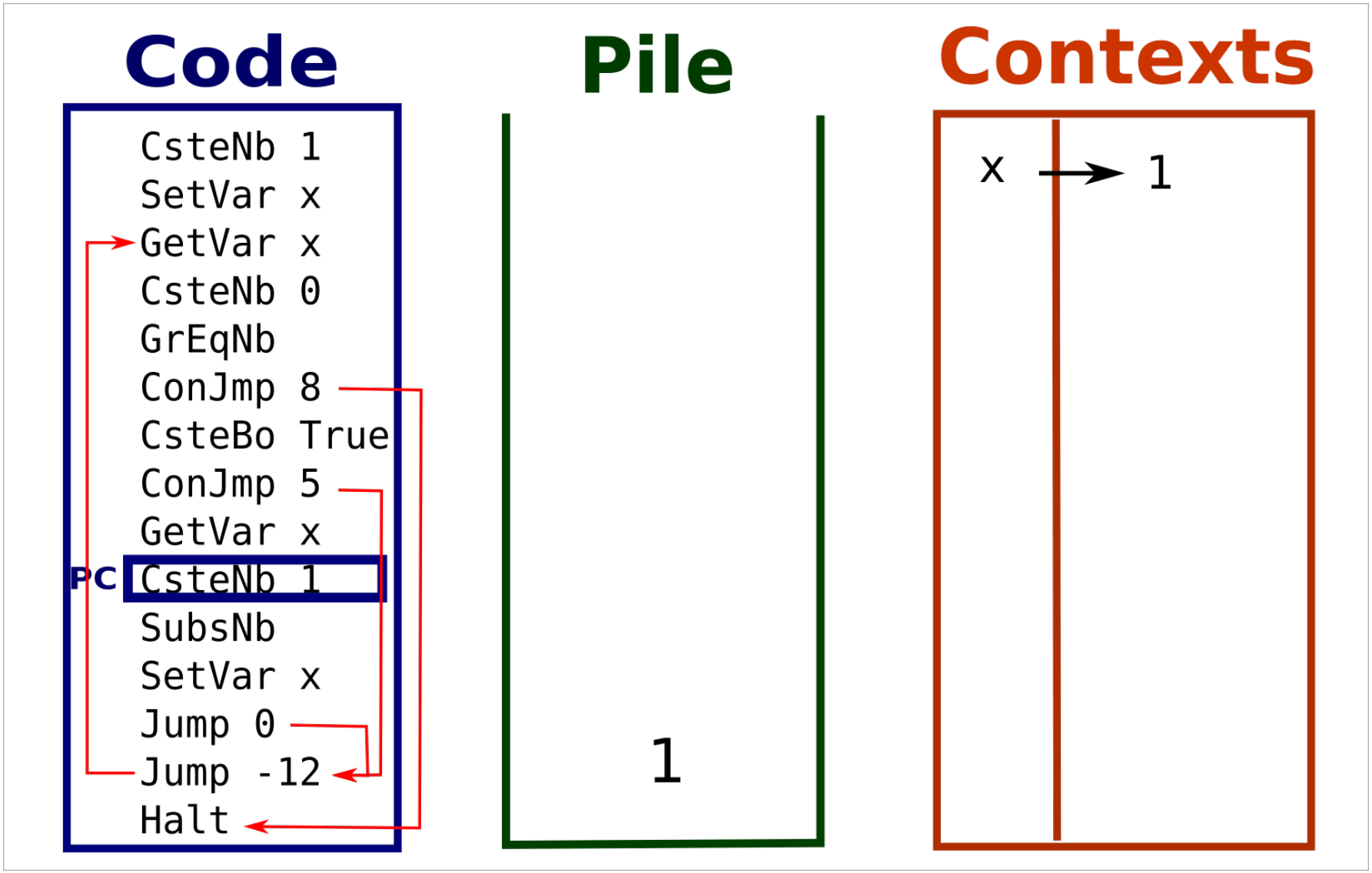
# Example en action



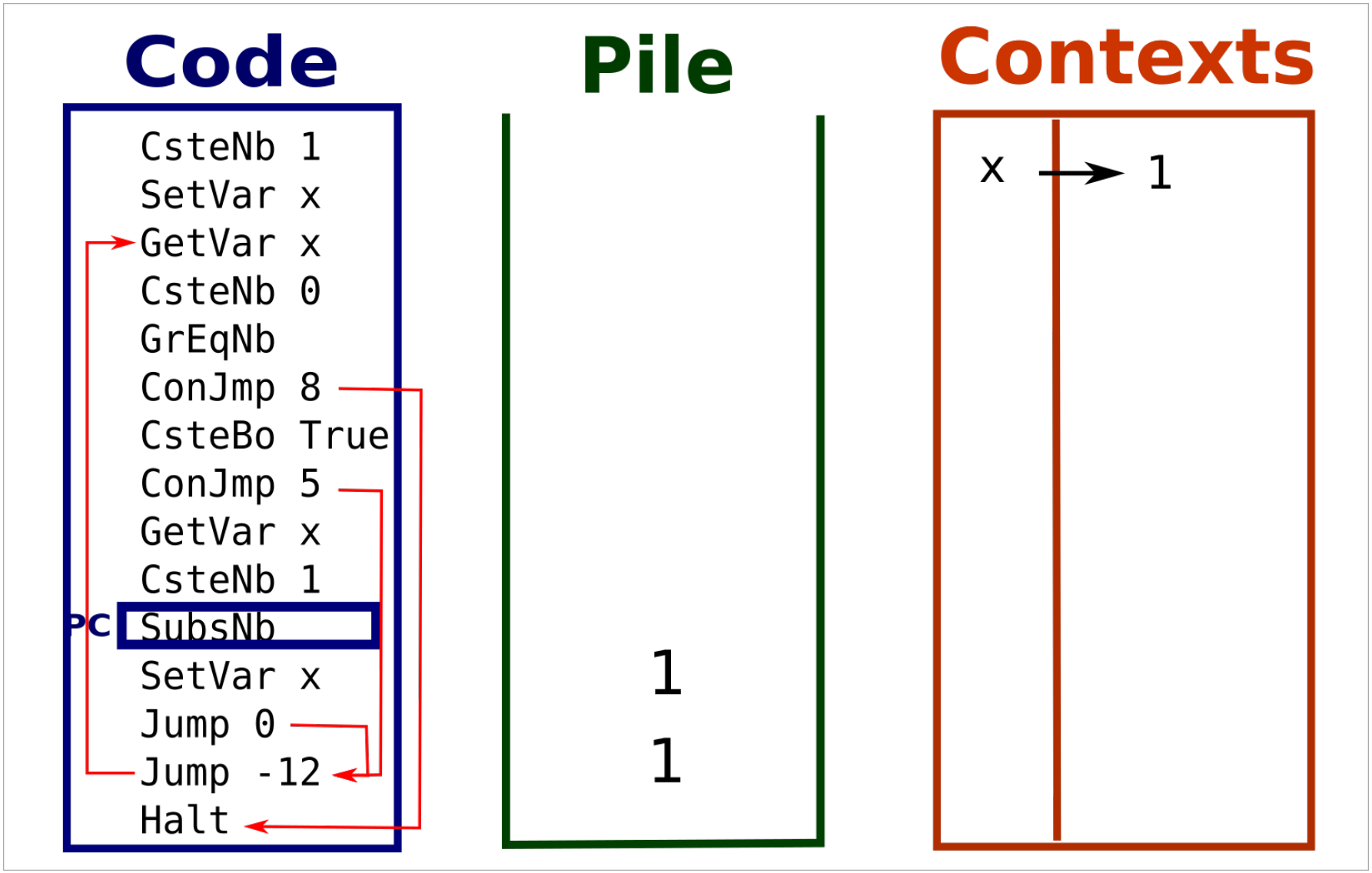
# Example en action



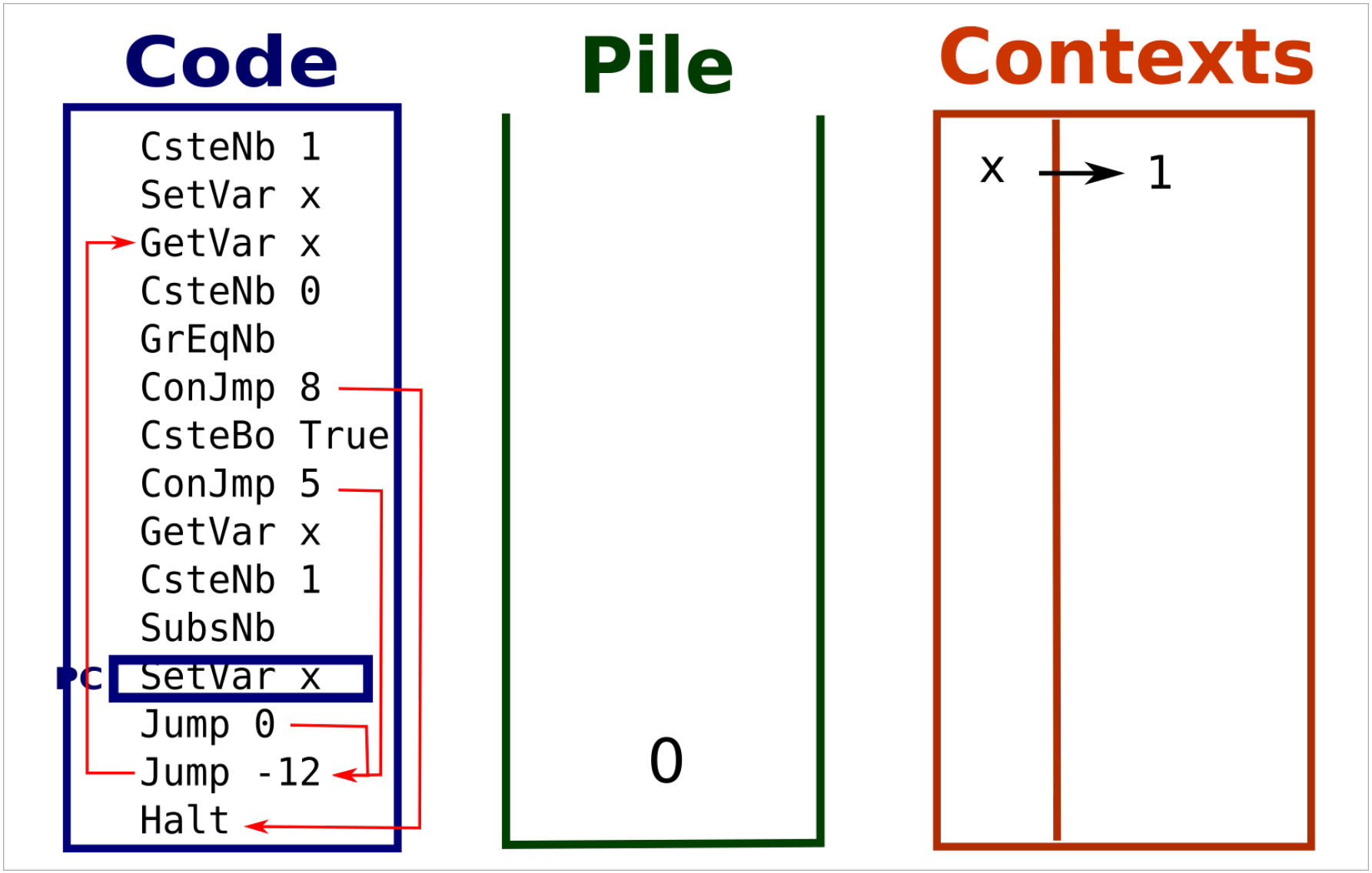
# Example en action



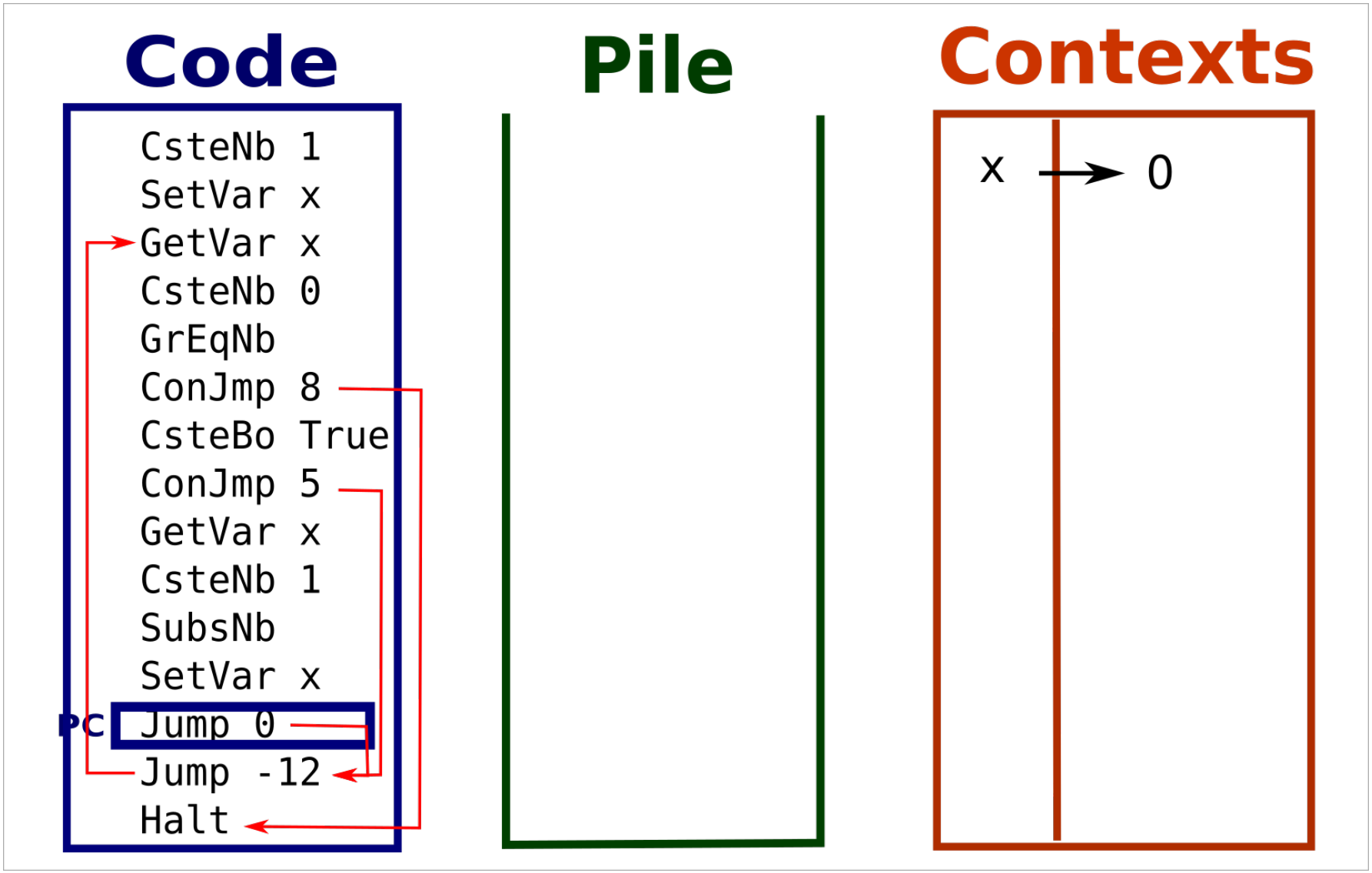
# Example en action



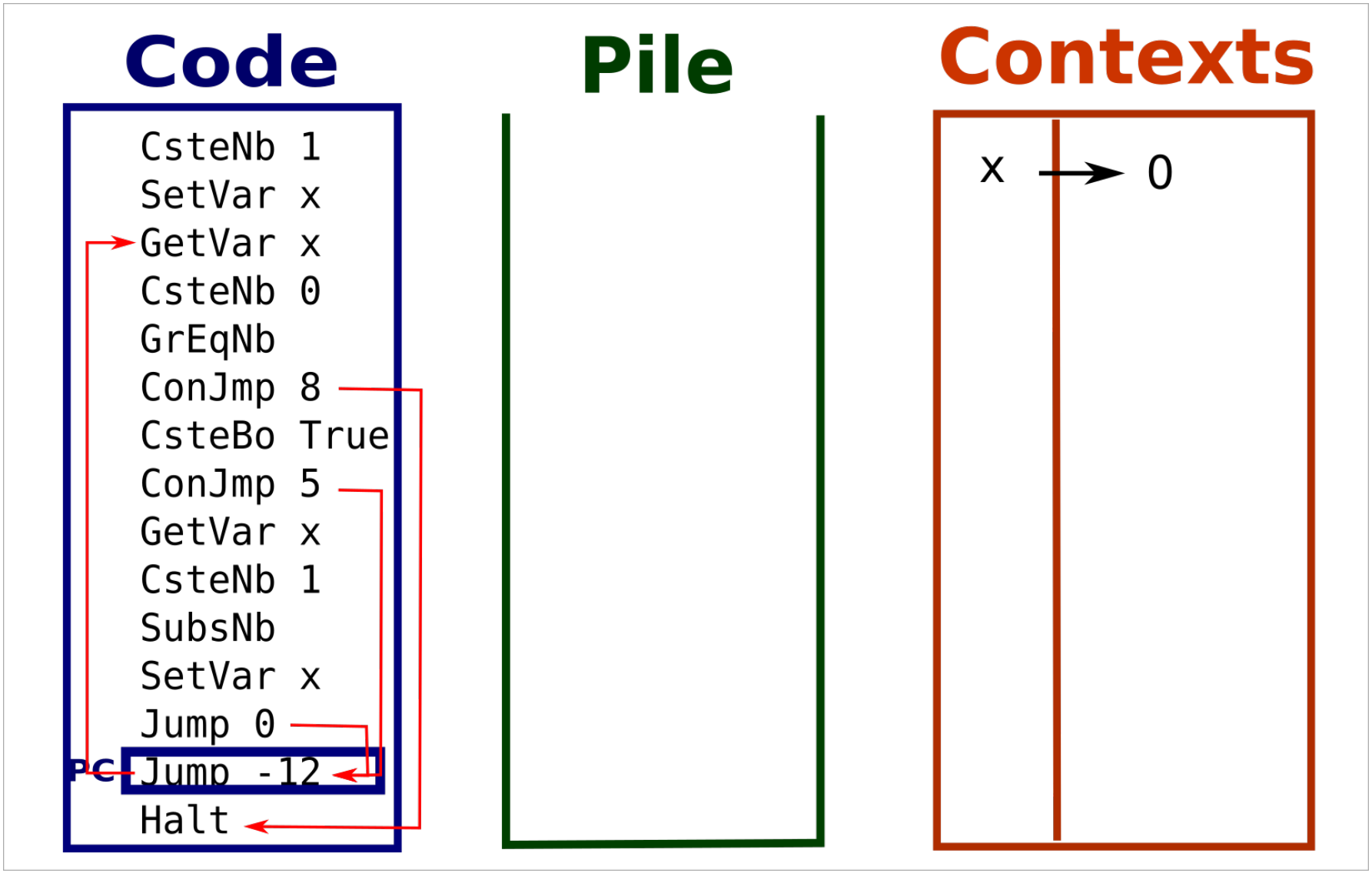
# Example en action



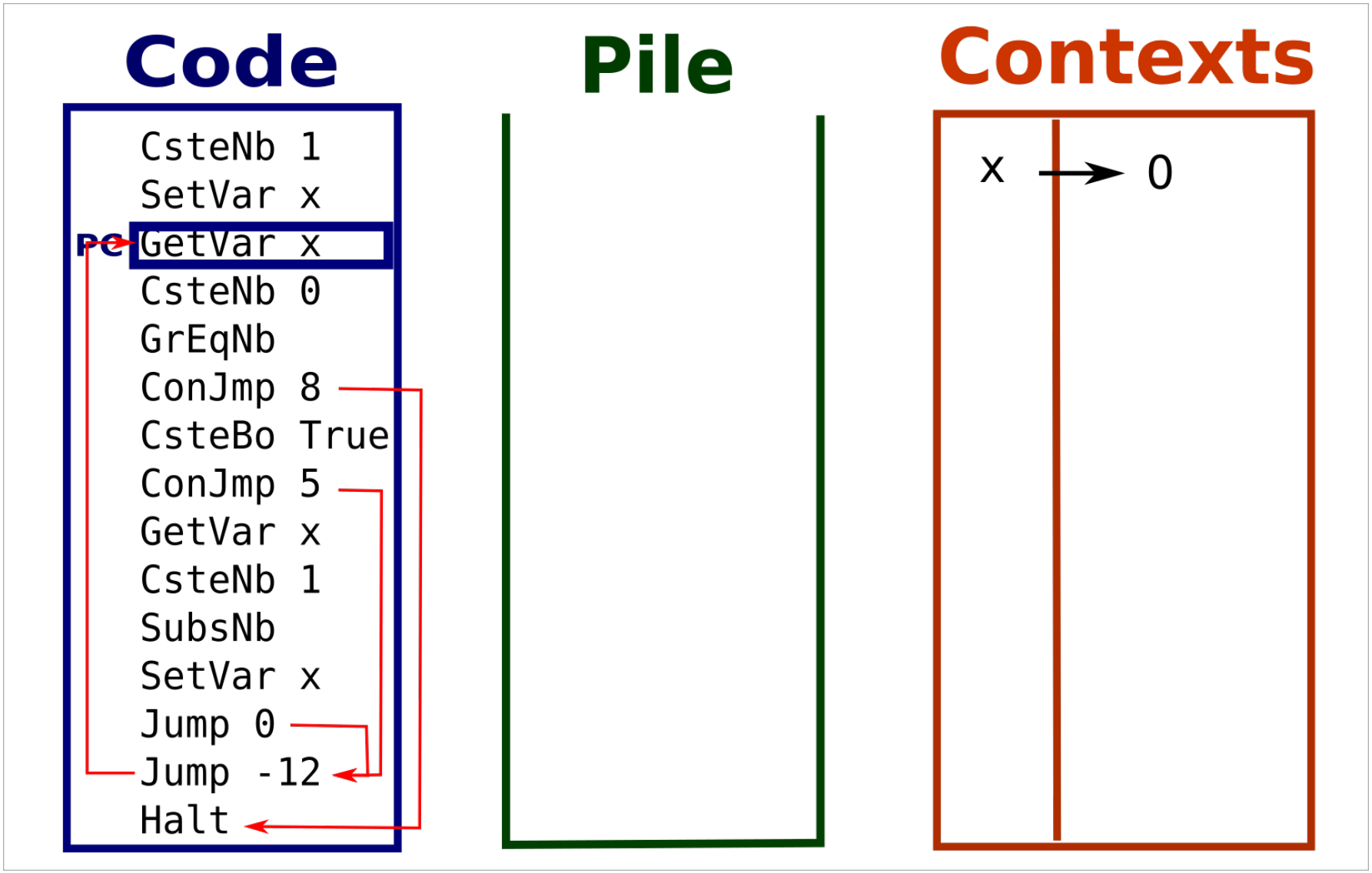
# Example en action



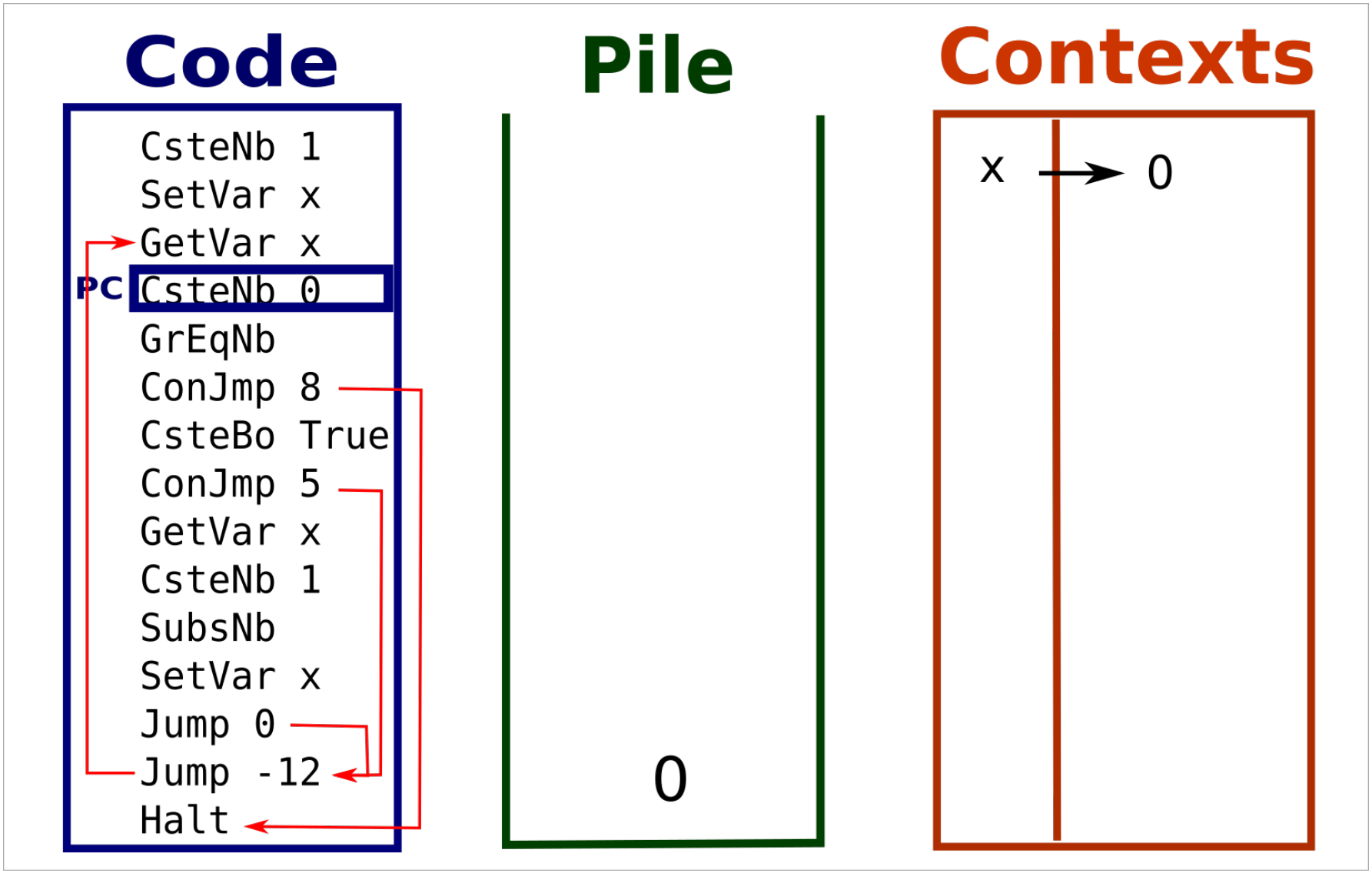
# Example en action



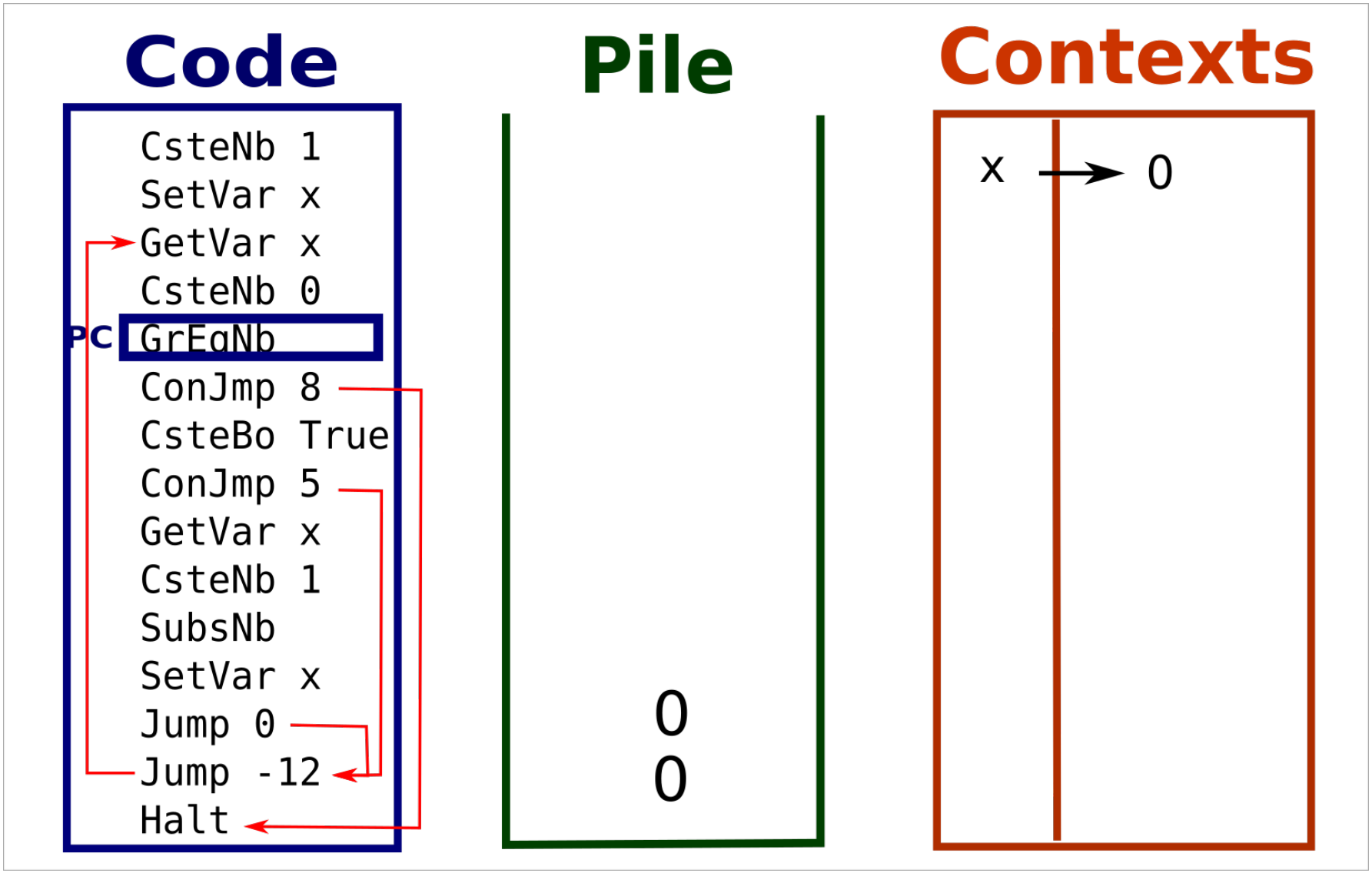
# Example en action



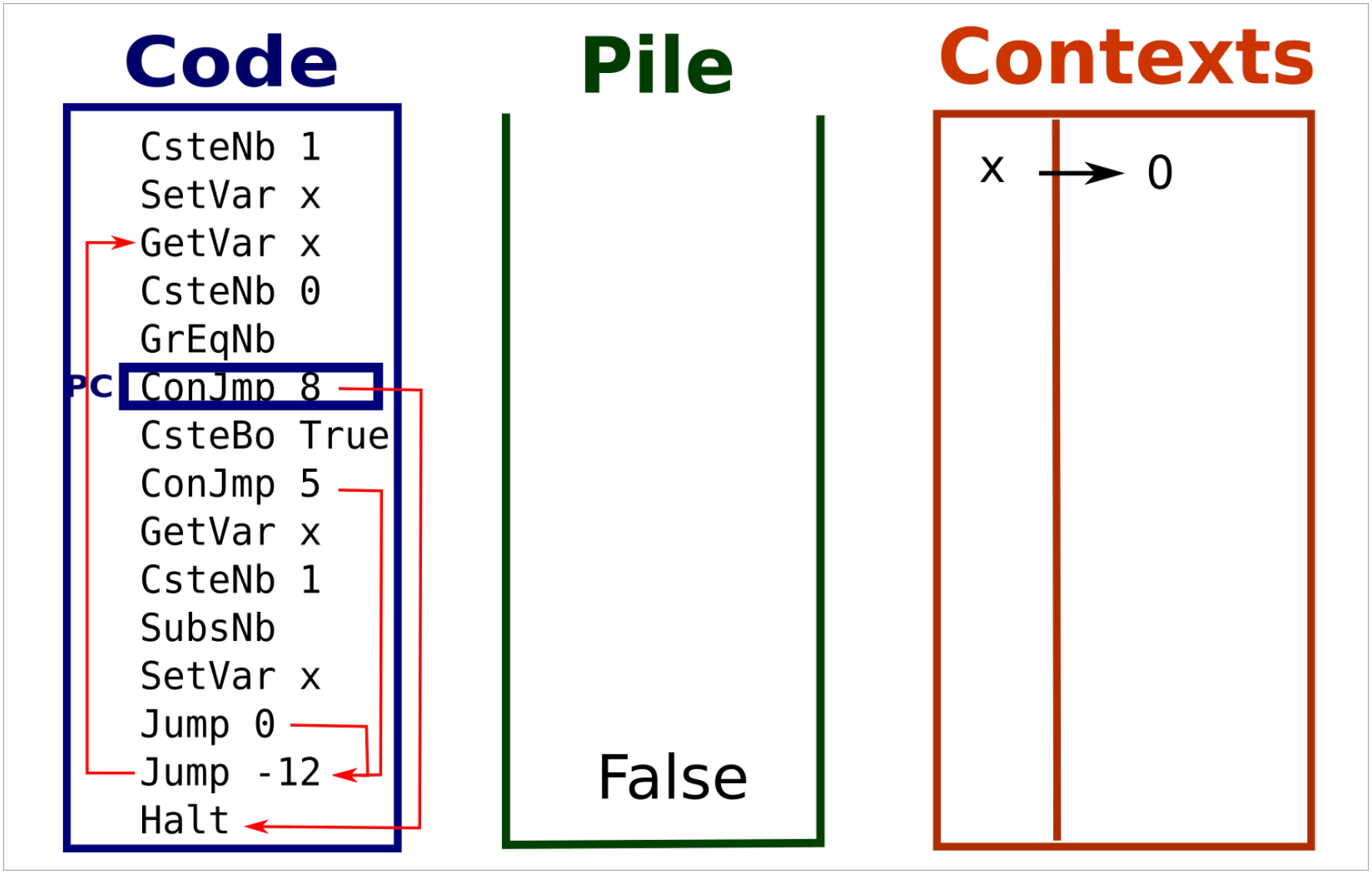
# Example en action



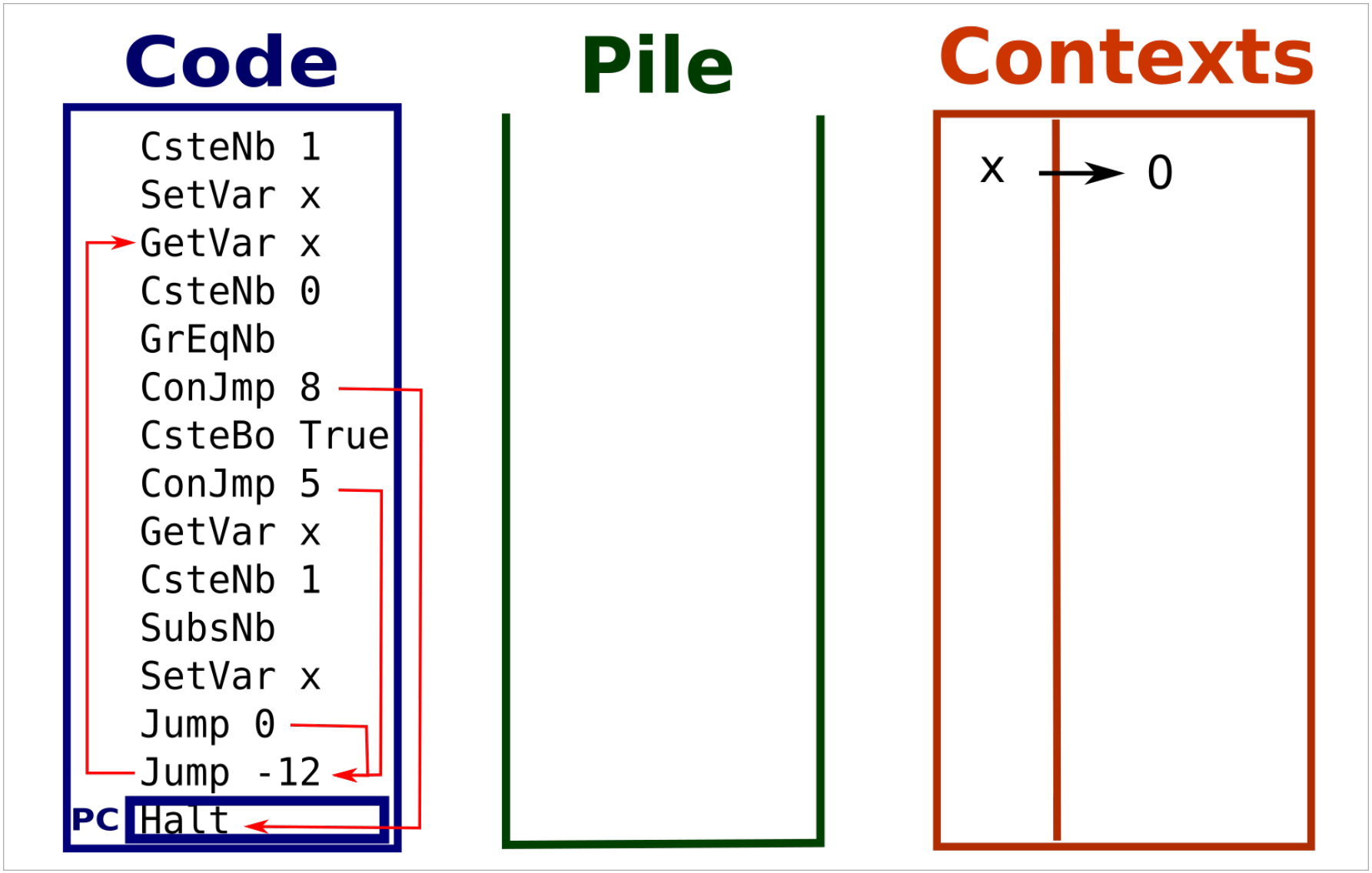
# Example en action



# Example en action



# Example en action



# Labels are authorised

| Instruction | meaning | semantics | before | after |
|---|---|---|---|---|
| **Jump** label | always jump | PC := position(label); | stk | stk |
| **ConJmp** label | may jump depending on stack's submit | if Pop<br>then PC := PC + 1;<br>else PC := position(label); | b:stk | stk |

## Label are swapped in the machine

The assembly is assembled before execution :
labels become offsets.

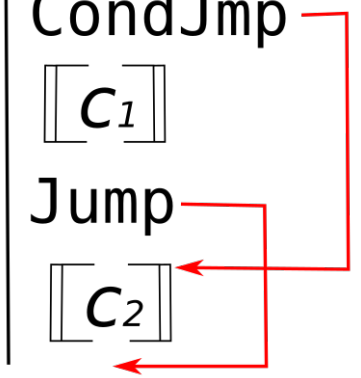
```
        CsteBo true
        SetVar x
loop:   GetVar x
        CsteNb 0
        GrStNb
        ConJmp fin
        CsteBo false
        SetVar x
        Jump   loop
fin:    GetVar x
```

becomes

```
CsteBo true
SetVar x
GetVar x
CsteNb 0
GrStNb
ConJmp 3
CsteBo false
SetVar x
Jump   -7
GetVar x
```

# Jumps : used for conditionals

$$[\![\textbf{if ( } e \textbf{ ) } c_1 \textbf{ else } c_2]\!] = \begin{array}{|l} [\![e]\!] \\ \texttt{CondJmp} \\ [\![c_1]\!] \\ \texttt{Jump} \\ [\![c_2]\!] \end{array}$$



## How to read this

Execute (the code of) $e$,
if a `True` is found then execute $c_1$ and go to the end of the if_then_else,
else jump the code of $c_1$ to execute $c_2$.

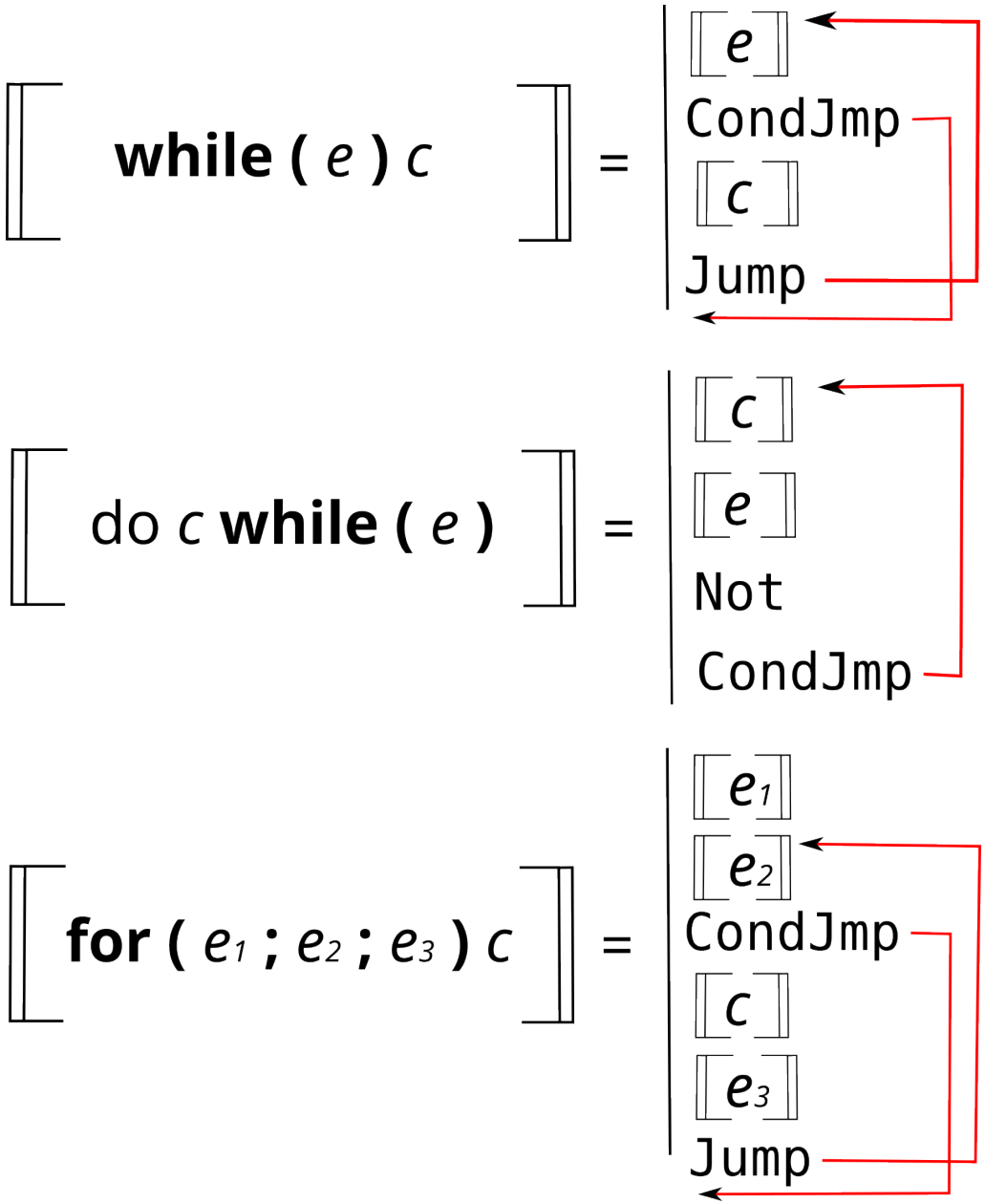
## Interpretation ⟦code⟧

the "default" assembly created from code

## Offsets ?

One also has to compute offsets of generate fresh labels...

# Jumps : utilised for loops



# By the way, what about && and || ?

# By the way, what about && and || ?

$$[\![ e_1 \;\&\&\; e_2 ]\!] = \begin{array}{|l} [\![ e_1 ]\!] \\ \texttt{CondJmp} \\ [\![ e_2 ]\!] \\ \texttt{Jump} \\ \texttt{CstBo False} \end{array}$$

$$[\![ e_1 \;||\; e_2 ]\!] = \begin{array}{|l} [\![ e_1 ]\!] \\ \texttt{CondJmp} \\ \texttt{CstBo True} \\ \texttt{Jump} \\ [\![ e_2 ]\!] \end{array}$$

# How to compute offsets ?

One has to (pre)compute the size of the assembly generated by skipped commands. Several solutions :

## On the fly

Before writing loops or conditionals, precompute it.
Advantage : only compute useful code.

## Semantic analysis

Add, in each node of the AST, the size of the expected generated code.
Advantage : linear time.

Along the project, you may will need the "size" of many things for many offsets (in functions, exceptions, classes...),
thus **the second solution tends to be better**.

## If you choose to use labels

Be careful, a label can only appear once in your code. Thus you need to generate fresh labels each time !

Hint: a label can ends with a number...

# Memory (des)allocation: system issue

## Allocating memory

Physical machines : in the stack (if not full) or via the system (virtual memory)
Virtual/abstract machines : idem but hidden/abstracted...

## Freeing memory

Physical machines : by hand
Virtual/abstract machines : via an embedded GC (generally that of the language the VM was written in)

# Context management

## What is a context ?

This is a dictionary that maps the name of the current variables to their value

## On virtual machines

Two choices :

- Hash table : efficient,
- Chained list : allow shadowing and copy.

In practice :

- Combination of both (but mainly the first).

## On physical machines

Idem + static binding (hash table $\rightarrow$ Vector/Stack).
Encoding in the stacks at statically determined distances.

# Context management

## What is a context ?

This is a dictionary that maps the name of the current variables to their value

## On virtual machines

Two choices :

- Hash table : efficient,
- Chained list : allow shadowing and copy.

In practice :

- Combination of both (but mainly the first).

## On physical machines

Idem + static binding (hash table $\rightarrow$ Vector/Stack).
Encoding in the stacks at statically determined distances.

# Miscellaneous : Meta-programming

## Meta-operation in interpreted languages

Manipulating the internal structure of the virtual machine during the execution.

## Examples

- Java : dynamic creation of classes,
- JS : code introspection of a function (not in the project)
- Python : dynamic manipulation of contexts.

Allow for magical solutions, sometime very efficient, but **compiling it is basically impossible**, and it is a source of **safety and security issues**.