GPMachine : Guide d'utilisation

Xavier Devroey Alain Solheid Michaël Marcozzi

Année académique 2011-2012

Table des matières

---

1 Introduction

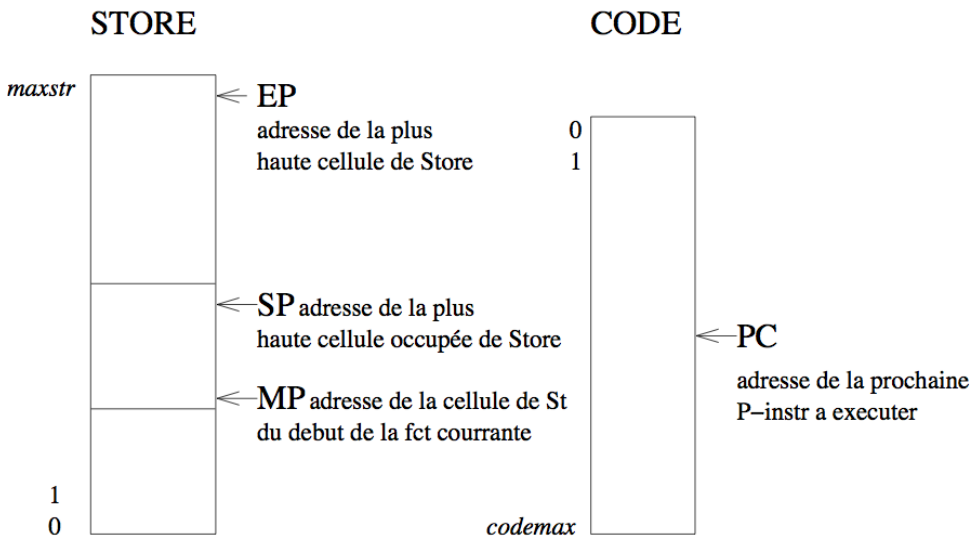


FIGURE 1 – Structure de la GPMachine

La GPMachine possède
- une mémoire de données $STORE$ de longueur $maxstr + 1$
- une mémoire pour les P-instructions de longueur $codemax + 1$
- un registre SP : contient l'adresse de la plus haute cellule occupée de $STORE$
- un registre EP : contient l'adresse de la plus haute cellule de $STORE$
- un registre PC : contient l'adresse de la prochaine P-instruction à exécuter de $CODE$
- un registre MP : contient l'adresse de la cellule de $STORE$ qui correspond au début du bloc d'activation de la fonction courante

Manipule
- entier ("i"), réel ("r"), booléen ("b"), adresse ("a")
- "N" représente un type numérique : $N \in \{i, a\}$
- "S" représente un type scalaire : $S \in \{i, b\}$

– "T" représente un type quelconque : $T \in \{i, a, b\}$

Remarques

– P-Code créé pour la traduction de Pascal
– LSD^{12}utilise une petite partie de P-Code
– on ne présente que la partie de P-Code nécessaire à la traduction de LSD^{12}
– quelques différences existent entre l'implémentation de la P-Machine qui vous est fournie et la définition qui est donnée dans WilHelm et Maurer (registre EP,instruction define,...)

2 Instructions

2.1 P-Instructions pour les expressions

2.1.1 Expressions numériques

P-Instruction	Signification	Condition	Résultat
add N	STORE[SP-1] := STORE[SP-1] $+_N$ STORE[SP] ; SP := SP − 1	(N,N)	(N)
sub N	STORE[SP-1] := STORE[SP-1] $-_N$ STORE[SP] ; SP := SP − 1	(N,N)	(N)
mul N	STORE[SP-1] := STORE[SP-1] $*_N$ STORE[SP] ; SP := SP − 1	(N,N)	(N)
div N	STORE[SP-1] := STORE[SP-1] $/_N$ STORE[SP] ; SP := SP − 1	(N,N)	(N)
neg N	STORE[SP] := $-_N$ STORE[SP]	(N)	(N)

Où (N,N) indique que STORE[SP] et STORE[SP-1] sont de type numérique.

2.1.2 Expressions booléennes

P-Instruction	Signification	Condition	Résultat
or b	STORE[SP-1] := STORE[SP-1] *or* STORE[SP] ; SP := SP − 1	(b,b)	(b)
and b	STORE[SP-1] := STORE[SP-1] *and* STORE[SP] ; SP := SP − 1	(b,b)	(b)
not b	STORE[SP] := *not* STORE[SP]	(b)	(b)
equ S	STORE[SP-1] := STORE[SP-1] $=_S$ STORE[SP] ; SP := SP − 1	(S,S)	(b)
les S	STORE[SP-1] := STORE[SP-1] $<_S$ STORE[SP] ; SP := SP − 1	(S,S)	(b)
leq S	STORE[SP-1] := STORE[SP-1] $\leq_S$ STORE[SP] ; SP := SP − 1	(S,S)	(b)
grt S	STORE[SP-1] := STORE[SP-1] $>_S$ STORE[SP] ; SP := SP − 1	(S,S)	(b)
geq S	STORE[SP-1] := STORE[SP-1] $\geq_S$ STORE[SP] ; SP := SP − 1	(S,S)	(b)

Où les valeurs VRAI et FAUX sont codées respectivement par 1 et 0 dans la P-Machine.

2.2 Lecture et écriture en mémoire

P-Inst	Signification	Condition	Résultat
ldc T c	SP := SP + 1 ; STORE[SP] := c	() Type(c) = T	(T)
lod T d q	SP := SP + 1 ; STORE[SP] := STORE[ad(d,q)]	() Type(STORE[ad(d,q)]=T	(T)
lda T d q	SP := SP + 1 ; STORE[SP] := ad(d,q)	() Type(STORE[ad(d,q)]=T	(a)
ind T	STORE[SP]:=STORE[STORE[SP]]	(a)	(T)
sto T	STORE[STORE[SP-1]]:=STORE[SP] ; SP := SP − 2	(a,T)	()

Où :
- d := différence entre profondeur de l'appel et de la déclaration (profondeur de la fonction courante - profondeur de la fonction dans laquelle la variable est déclarée)
- q := adresse relative de la variable (offset)
- ad(d,q) := base(d,MP)+q
- base(d,MP) := if (d=0) then MP else base(d-1,STORE[MP+1]) fi

2.3 Traduction d'une instruction d'affectation

L'idée pour la traduction d'instructions en PCode est de travailler récursivement. Le tableau suivant donne quelques règles pour la traduction de l'affectation et des opérations d'addition et de soustraction :

Fonction		Condition
$PCode(z := e) =$	$PCode_G(z)$; $PCode_D(e)$; *sto T*	$Type(z) = Type(e) = T$
$PCode_D(e_1 + e_2) =$	$PCode_D(e_1)$; $PCode_D(e_2)$; *add N*	$Type(e_1) = Type(e_2) = N$
$PCode_D(e_1 * e_2) =$	$PCode_D(e_1)$; $PCode_D(e_2)$; *mul N*	$Type(e_1) = Type(e_2) = N$
$PCode_D(c) =$	*ldc T c*	c constante et $Type(c) = T$
$PCode_G(z) =$	*lda T d(z) q(z)*	z variable et $Type(z) = T$
$PCode_D(z) =$	$PCode_G(z)$; *ind T*	z variable et $Type(z) = T$

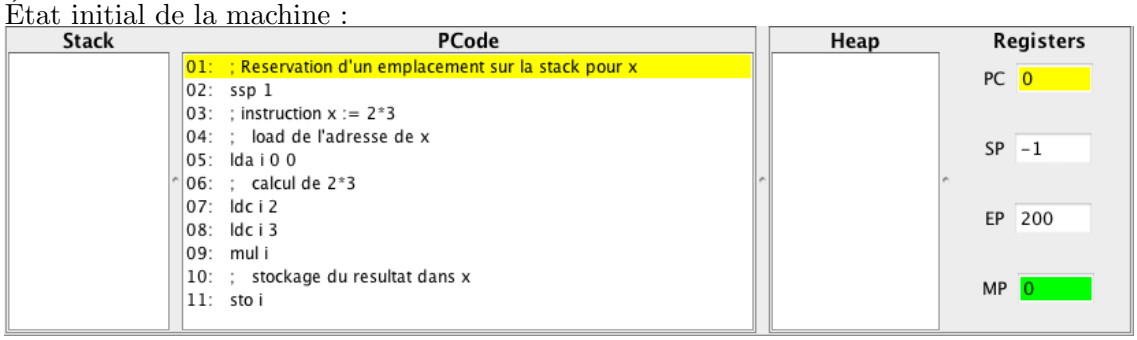
Où $d(z)$ et $q(z)$ sont respectivement la profondeur relative et l'adresse relative de z.

Exemples L'instruction `x:=2*3` avec x de type integer sera traduite de la manière suivante :

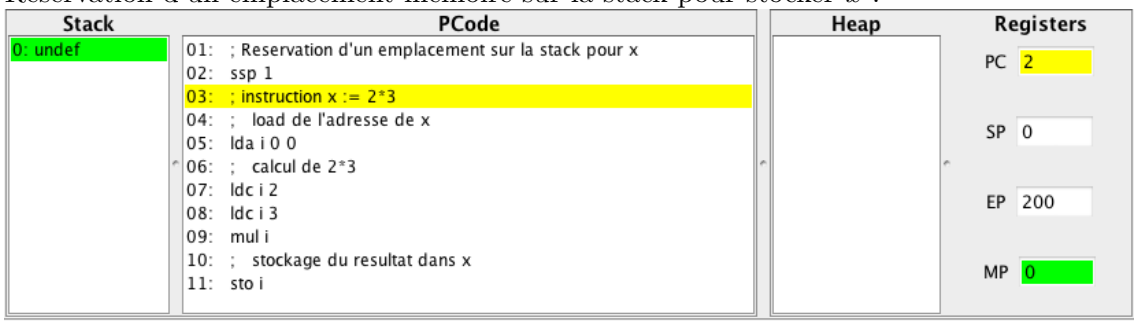
$$
\begin{aligned}
&PCode(x := 2 * 3)\\
&\quad = PCode_G(x);\ PCode_D(2 * 3);\ sto\ i\\
&\quad = lda\ i\ d(x)\ q(x);\ PCode_D(2 * 3);\ sto\ i\\
&\quad \vdots\\
&\quad = lda\ i\ d(x)\ q(x);\ ldc\ i\ 2;\ ldc\ i\ 3;\ mul\ i;\ sto\ i
\end{aligned}
$$

Si l'on considère que la variable x est située à l'adresse 0 ($q(x) = 0$) du contexte courant ($d(x) = 0$), on obtient dans la PMachine :

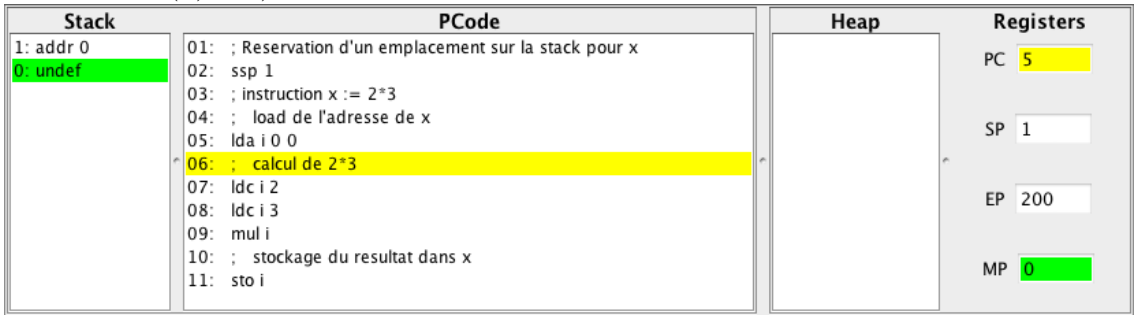
1. État initial de la machine :



2. Réservation d'un emplacement mémoire sur la stack pour stocker x :

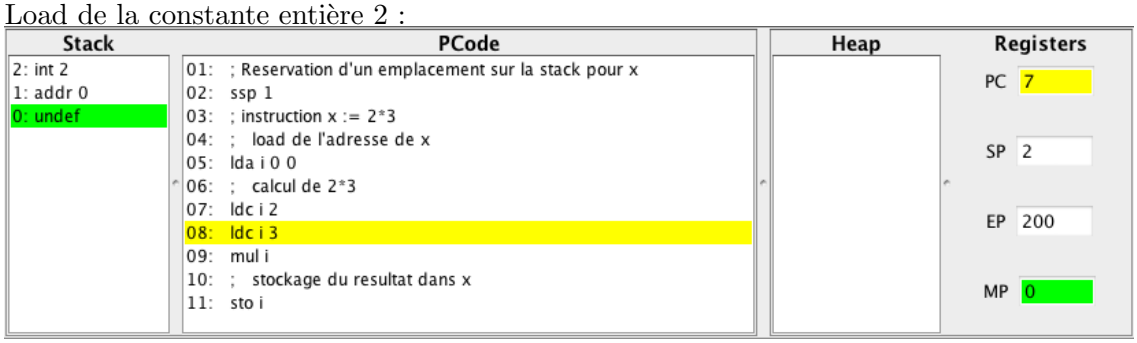


3. Load de l'adresse de x situé en position 0 ($q(x) = 0$) du contexte courant (la différence de profondeur $d(x) = 0$) :

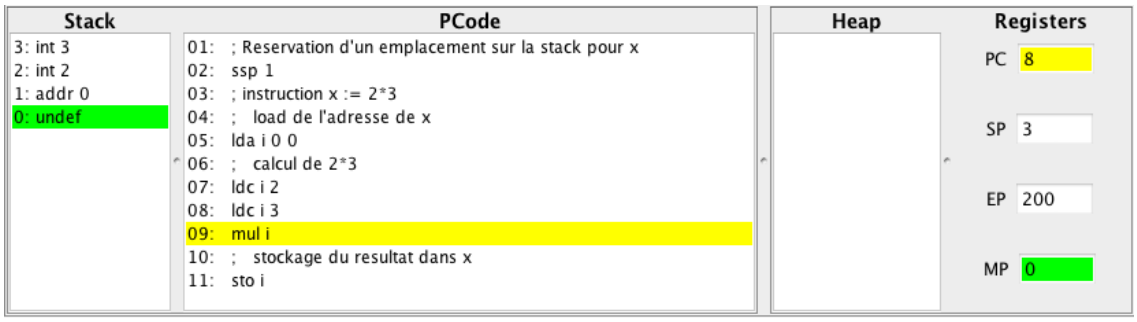


À noter que si l'on avait voulu accéder depuis une fonction $fct()$ à une variable a déclarée dans la fonction $fctPere()$ parente de $fct()$, la différence de profondeur $d(a)$ aurait été de 1. De même, si a avait été déclarée dans $fctGrandPere()$, $d(a)$ vaudrait 2, etc.

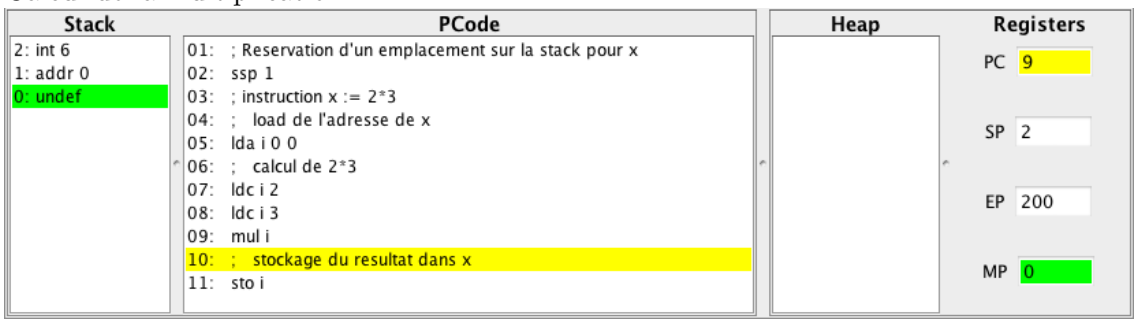
4. Load de la constante entière 2 :



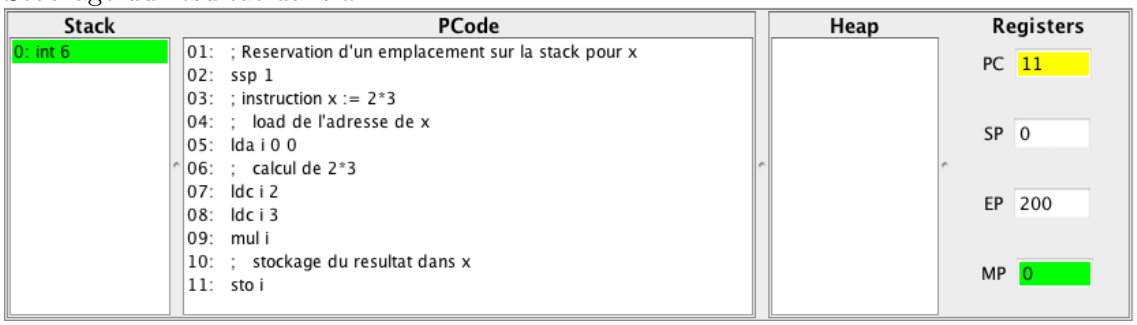
5. Load de la constante entière 3 :



6. Calcul de la multiplication :



7. Stockage du résultat dans x :



De la même manière, l'instruction `y:=3*x+4` avec x et y de type integer sera traduite de la manière suivante :

$$
\begin{aligned}
&PCode(y := 3 * x + 4)\\
&\quad = PCode_G(y);\ PCode_D(3 * x + 4);\ sto\ i\\
&\quad = lda\ i\ d(y)\ q(y);\ PCode_D(3 * x + 4);\ sto\ i\\
&\quad \vdots\\
&\quad = lda\ i\ d(y)\ q(y);\ ldc\ i\ 3;\ lda\ i\ d(x)\ q(x);\ ind\ i;\ mul\ i;\ ldc\ i\ 4;\ add\ i;\ sto\ i
\end{aligned}
$$

2.4 P-Instructions de branchement

P-Inst	Effet	Commentaire
define @k		définition d'une étiquette
ujp @k	PC := adresse(@k)	branchement inconditionnel
fjp @k	if STORE[SP] = false then PC := adresse(@k) SP := SP - 1	branchement conditionnel

L'évaluation paresseuse des instructions *and* et *or* peut par exemple s'implémenter en utilisant le branchement conditionnel :

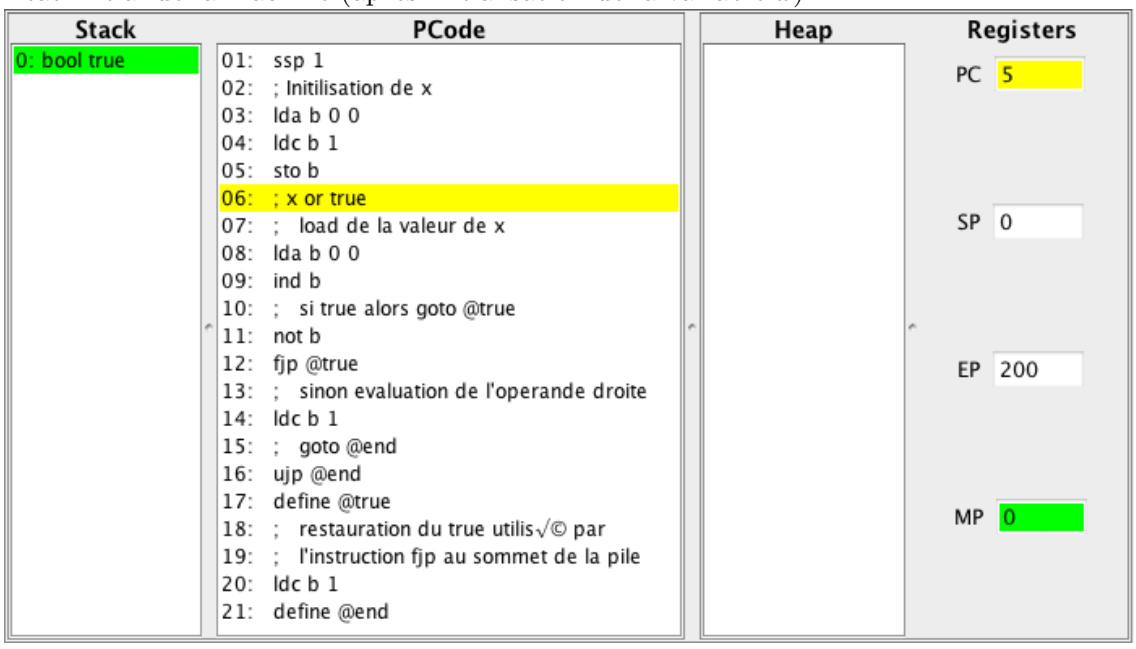
$$
\begin{aligned}
&PCode(E_1 \ or \ E_2) \\
&\quad = PCode_D(E_1); \\
&\quad\ \ not; \\
&\quad\ \ fjp \ @true; \\
&\quad\ \ PCode_D(E_2); \\
&\quad\ \ ujp \ @end; \\
&\quad\ \ define \ @true; \\
&\quad\ \ ldc \ b \ 1; \\
&\quad\ \ define \ @end;
\end{aligned}
$$

La traduction de l'instruction `x or false` devient donc en PCode :

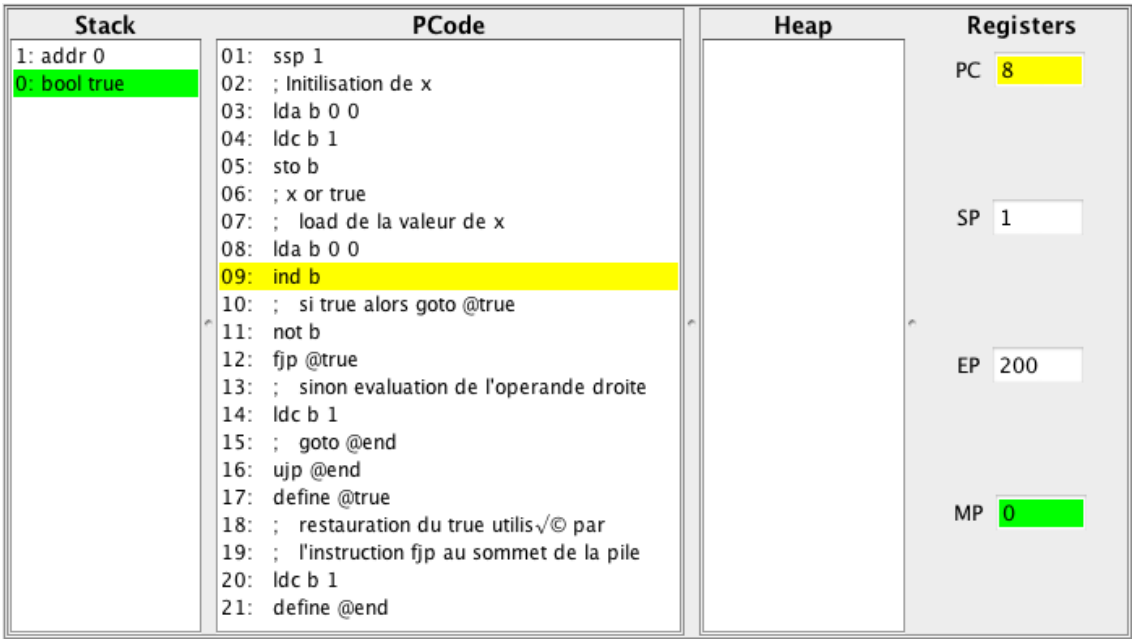
$$
\begin{aligned}
&PCode(x or false) \\
&\quad = \ PCode_D(x); \ not; \ fjp \ @true; \ PCode_D(true); \\
&\quad\ \ ujp \ @end; \ define \ @true; \ ldc \ b \ 1; \ define \ @end \\
&\quad = \ lda \ i \ d(x) \ q(x); \ ind \ b; \ not \ b; \ fjp \ @true; \\
&\quad\ \ ldc \ b \ 1; \ ujp \ @end; \ define \ @true; \ ldc \ b \ 1; \ define \ @end
\end{aligned}
$$

Si on exécute ce code dans la GPMachine, on obtient le résultat suivant :

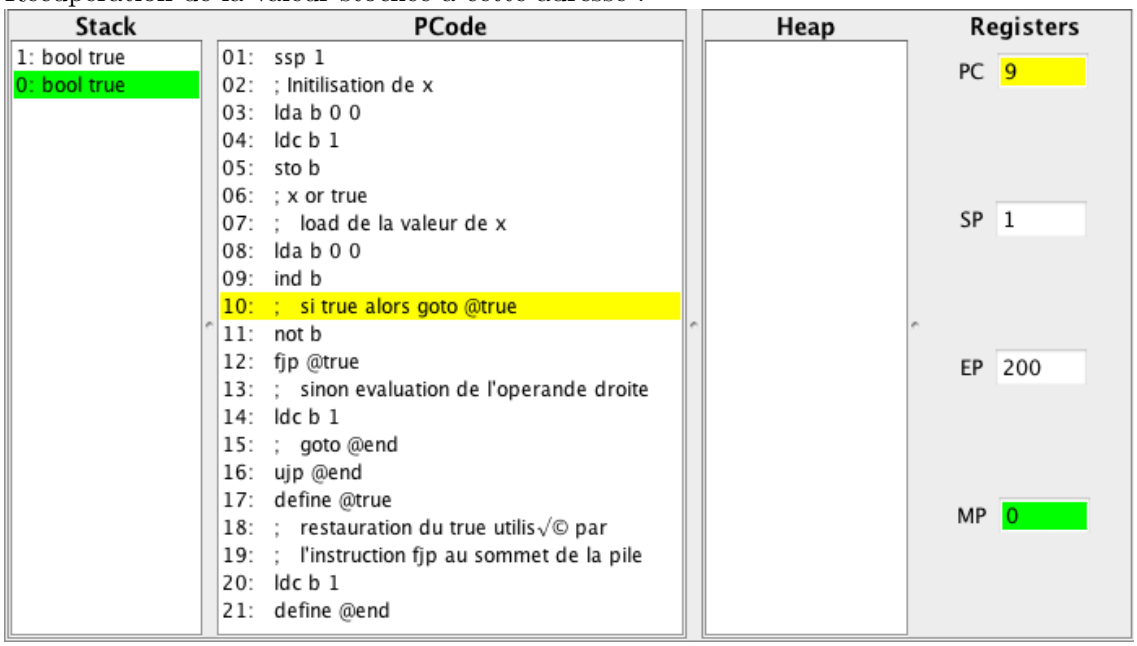
1. État initial de la machine (après initialisation de la variable x) :



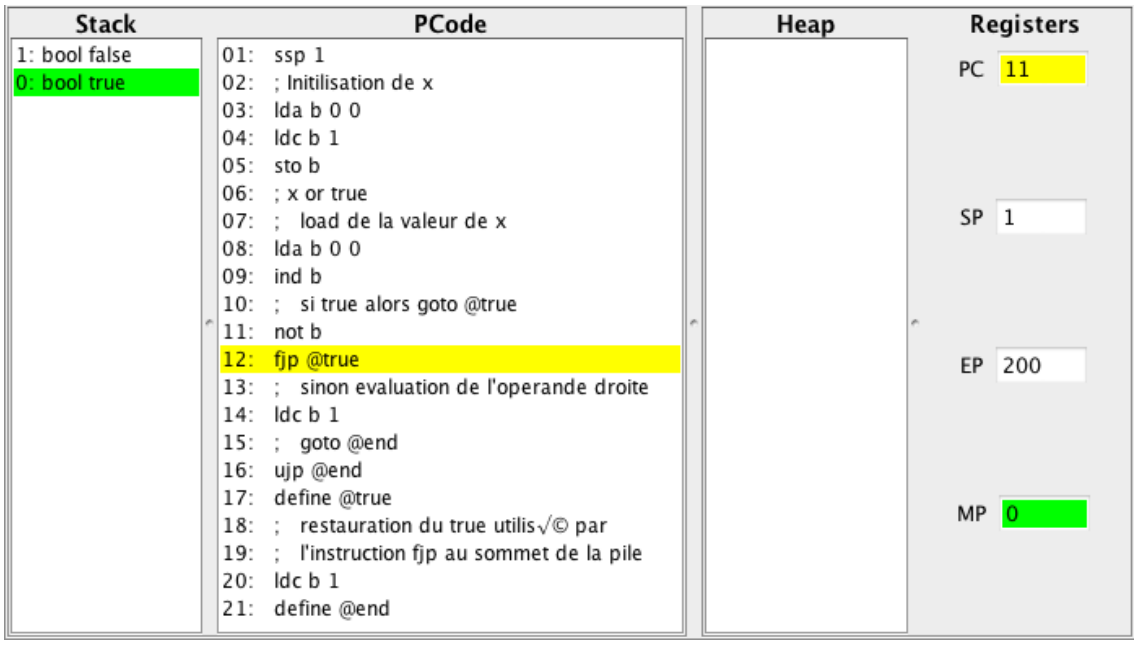
2. Load de l'adresse de la variable x :



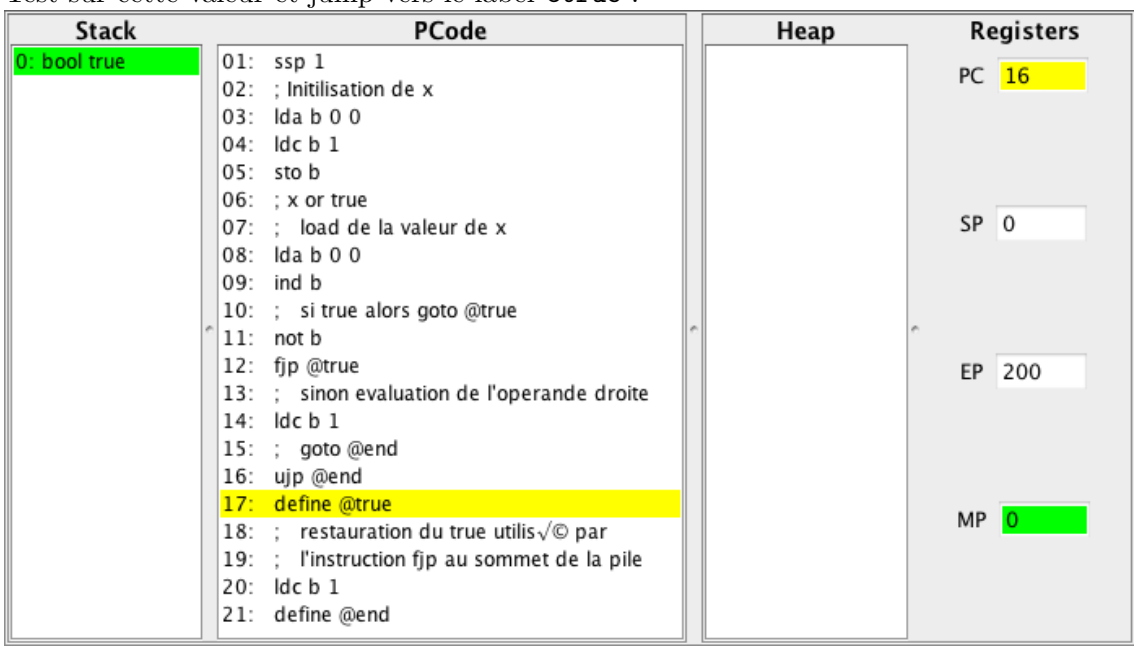
3. Récupération de la valeur stockée à cette adresse :



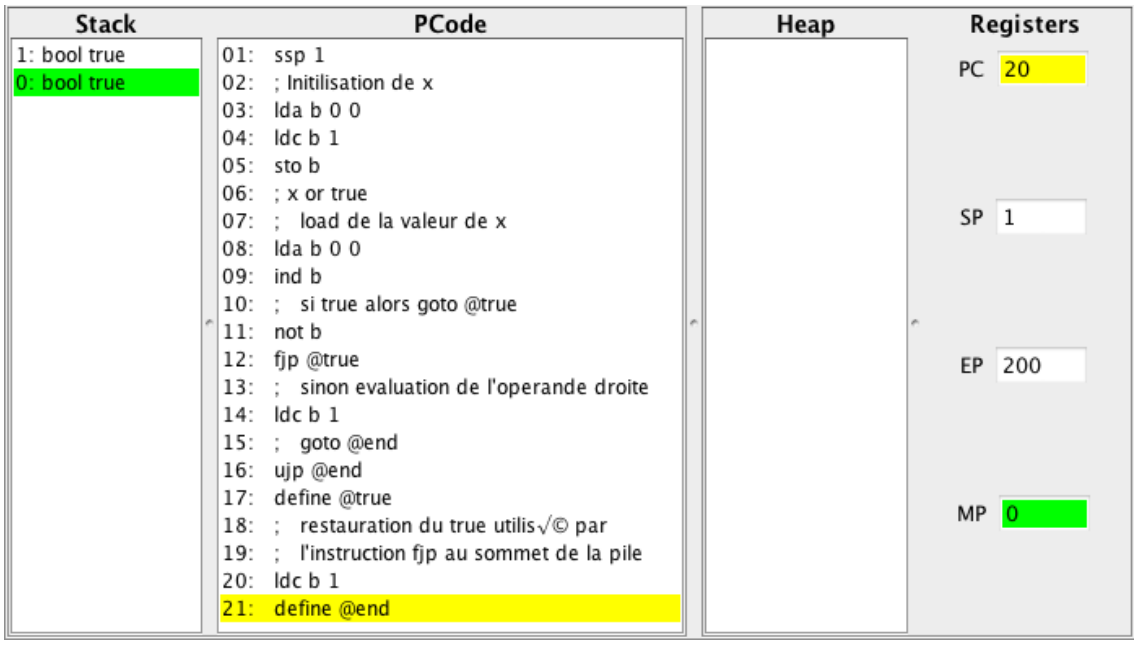
4. Négation du résultat de l'évaluation de l'expression gauche :



5. Test sur cette valeur et jump vers le label `@true` :



6. Restauration de la valeur *true* consommée par l'instruction `fjp` au sommet de la pile :



2.5 Autres traductions

Fonction		Condition
$PCode(read\ x) =$	$PCode_G(x);$ $read;$	$Type(x) = N$
$PCode(write\ x) =$	$PCode_D(x); prin$ $prin;$	$Type(x) = N$
$PCode(if\ E\ then\ I_1\ else\ I_2\ fi) =$	$PCode_D(E);$ $fjp\ @else;$ $PCode(I_1);$ $ujp\ @fi;$ $define\ @else;$ $PCode(I_2);$ $define\ @fi$	$Type(E) = b$
$PCode(while\ E\ do\ I\ od) =$	$define\ @while;$ $PCode_D(E);$ $fjp\ @od;$ $PCode(I);$ $ujp\ @while;$ $define\ @od$	$Type(E) = b$

2.6 Traduction d'un programme

Fonction		Condition
$PCode(Program) =$	$ssp\ s;$ $ujp\ @begin;$ $PCode(ProcDeclList);$ $define\ @begin;$ $PCode(InstList);$ stp	

(où ssp s effectue $SP := MP + s - 1$ c-à-d réserve la place dans STORE pour les variables)

2.7 P-instructions pour les procédures et fonctions

Les instructions suivantes sont utilisées lors de la définition et l'appel de procédures et fonctions.

P-Instruction	Signification	Commentaire
mst d	STORE[SP+2] := base(d,MP) ; STORE[SP+3] := MP ; SP :=SP+5	où d = différence profondeur appel/déclaration prédécesseur dynamique réserve l'espace sur la pile pour le bloc d'appel
cup p @fct	MP :=SP-(p+4) STORE[MP+4] :=PC PC :=adresse(@k)	réserve l'espace pour les paramètres où p = nombre de paramètres sauver l'adresse de retour aller à @fct
ssp s	SP :=MP+s-1	s = 5 + nombre de cellules mémoire pour les paramètres et les variables de la fonction/procédure
retp	SP :=MP-1 PC :=STORE[MP+4] MP :=STORE[MP+2]	libère l'espace occupé aller à l'instruction qui suit l'appel registre MP à jour
retf	SP :=MP-1 SP :=SP+1 STORE[SP] :=STORE[MP] PC :=STORE[MP+4] MP :=STORE[MP+2]	libère l'espace occupé réserve une case pour la valeur de retour stocke la valeur de retour en haut de la pile aller à l'instruction qui suit l'appel registre MP à jour

Avec base(d,MP) := if (d=0) then MP else base(d-1,STORE[MP+1]) fi

2.8 Traduction des procédures, fonctions et paramètres

2.8.1 PCode pour la déclaration d'une procédure

Fonction	
$PCode(ProcDecl) =$	$define$ @$proc$; $ssp\ s$; ujp @$procBody$; $PCode(ProcDeclList)$; $define$ @$procBody$; $PCode(InstList)$; $retp$

2.8.2 PCode pour un appel de procédure

Fonction	
$PCode(proc(e_1, e_2, ..., e_n)) =$	$mst\ d;$ $PCode_A(e_1);$ $PCode_A(e_2);$ ...; $PCode_A(e_n);$ $cup\ p\ @proc$

avec

$PCode_A(e) = PCode_G(e)$ si e est un paramètre passé par adresse

$PCode_A(e) = PCode_D(e)$ si e est un paramètre passé par valeur

$PCode_G$ pour les paramètres

$PCode_G(x) = lda\ T\ d(x)\ q(x)$ si x est une variable locale ou globale de type T

$PCode_G(x) = lda\ T\ d(x)\ q(x)$ si x est un paramètre passé par valeur de type T

$PCode_G(x) = lod\ a\ d(x)\ q(x)$ si x est un paramètre passé par adresse

2.8.3 PCode pour la déclaration d'une fonction

Fonction	
$PCode(ProcDecl) =$	$define\ @fct;$ $ssp\ s;$ $ujp\ @fctBody;$ $PCode(FctDeclList);$ $define\ @fctBody;$ $PCode(InstList);$ $retf$

PCode pour une l'instruction return Contrairement aux procédures, les fonctions renvoient une valeur à l'appelant. En P-Code, la valeur est placée au sommet de la pile une fois de retour dans le contexte de l'appelant. La valeur retournée est celle située à l'adresse 0 dans le contexte de la fonction appelée. Une instruction `return` sera donc traduite en PCode de la manière suivante :

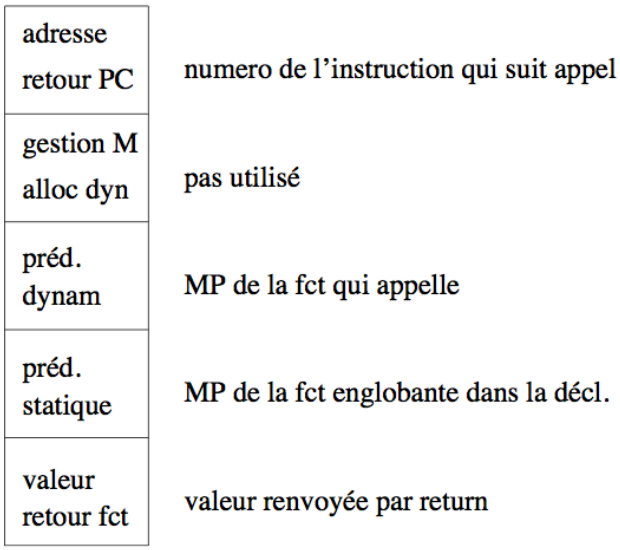
Fonction		Condition
$PCode(return\ e) =$	$lda\ T\ 0\ 0;$ $PCode_D(e);$ $sto\ T;$ $retf;$	$Type(e) = T$

2.8.4 PCode pour un appel de fonction

Fonction	
$PCode(fct(e_1, e_2, ..., e_n)) =$	$mst\ d;$ $PCode_A(e_1);$ $PCode_A(e_2);$ ...; $PCode_A(e_n);$ $cup\ p\ @fct$

2.8.5 Bloc d'appel de procédure/fonctions

La structure du bloc d'appel d'une procédure ou d'une fonction est présenté dans la figure suivante. Cet espace est réservé sur la pile au moment de l'exécution d'une instruction `mst`.



2.8.6 Exemple d'appel de fonction

L'exemple suivant présente un appel de fonction avec passage de paramètres et le renvoi d'une valeur au programme appelant. Sa syntaxe correspond à celle définie dans le cadre du TP compilateur LSD^{12}.

```
program a;
function main():void;
var
        x int;
        function addTo(a : int, b : int):int;
                var
                begin
                        return a + b;
                end;
begin
        x := 2;
        x := addTo(x, 3);
end;
end;
```

La traduction en PCode devient alors :

```
; ************* Start program *************
ssp 1
ujp @begin
; ------- Start function addTo -------
define @addTo
ssp 7
```

```
lda i 0 0
lod i 0 5
lod i 0 6
add i
sto i
retf
; ********************
; Start program :
; ********************
define @begin
lda i 0 0
ldc i 2
sto i
lda i 0 0
mst 0
lod i 0 0
ldc i 3
cup 2 @addTo
sto i
stp
```

2.9 Utilisation du tas (heap)

P-Inst	Signification	Condition	Résultat

new	if (EP-STORE[SP] $\leq$ SP)
	then error("Heap Overflow")
	else
	EP :=EP-STORE[SP];
	STORE[SP] := EP+1;
	fi;
		(i)	(a)

Attention, il n'y a pas d'instruction *free* disponible en P-Code.

2.9.1 Exemple d'utilisation du tas

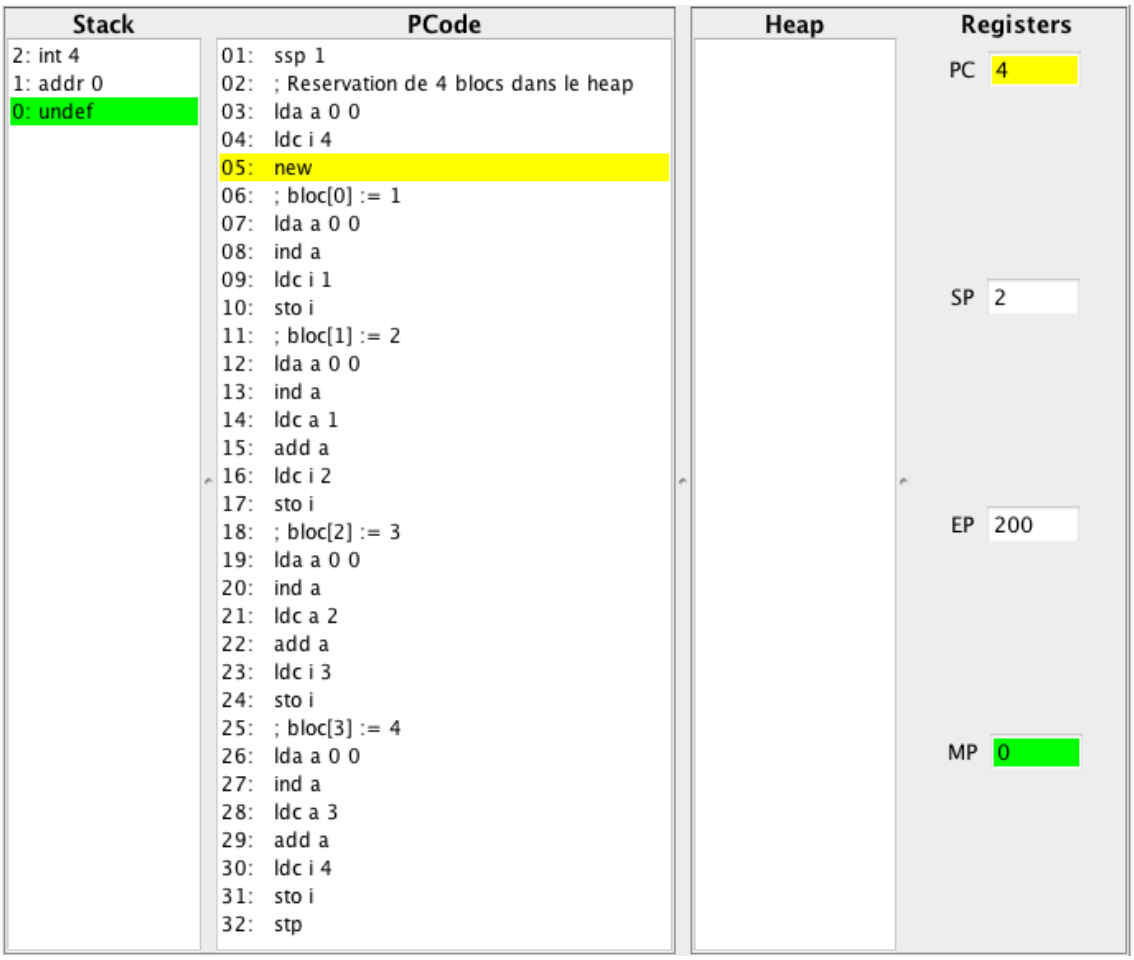
Le PCode suivant réserve 4 emplacements sur le *heap* et les initialise avec les valeurs 1, 2, 3 et 4.

```
ssp 1
; Reservation de 4 blocs dans le heap
lda a 0 0
ldc i 4
new
; bloc[0] := 1
lda a 0 0
ind a
ldc i 1
sto i
; bloc[1] := 2
```

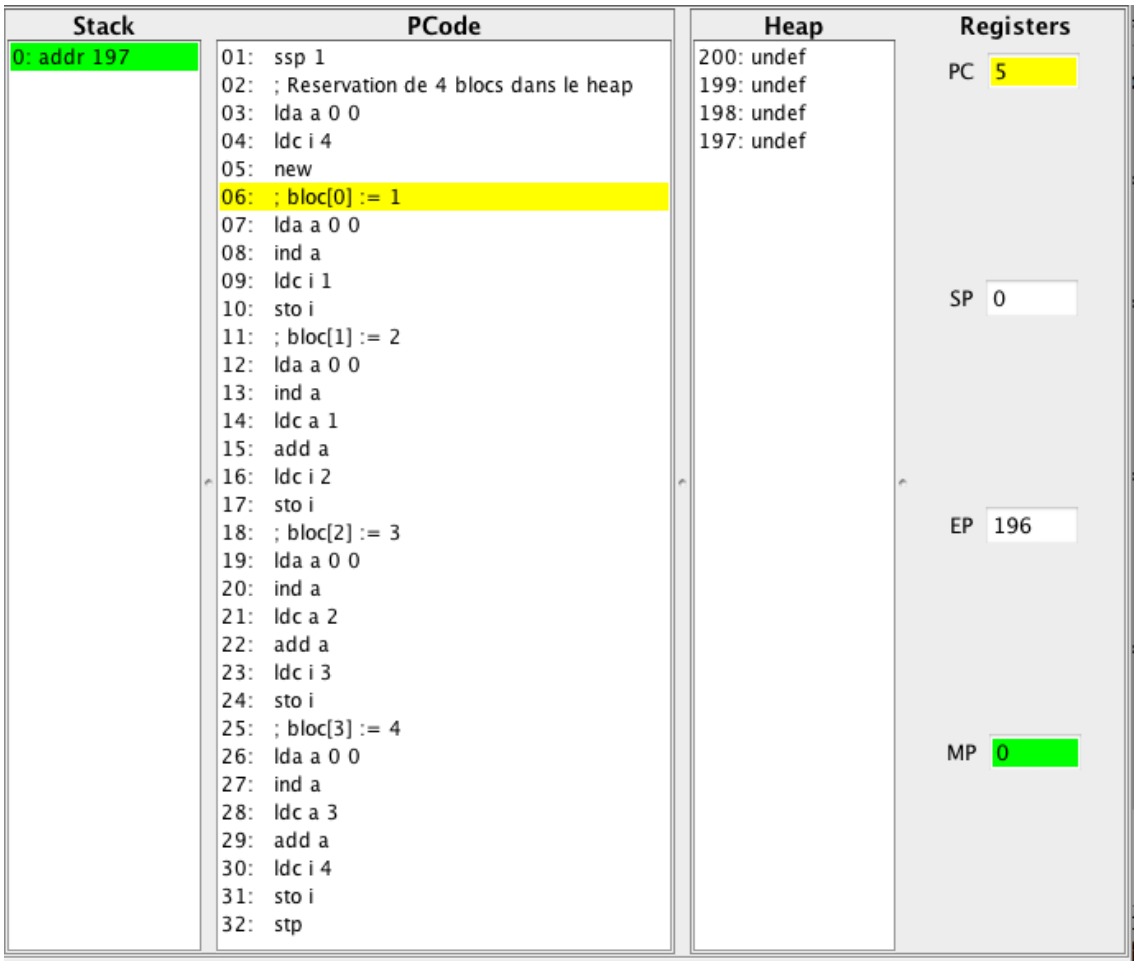
```
lda a 0 0
ind a
ldc a 1
add a
ldc i 2
sto i
; bloc[2] := 3
lda a 0 0
ind a
ldc a 2
add a
ldc i 3
sto i
; bloc[3] := 4
lda a 0 0
ind a
ldc a 3
add a
ldc i 4
sto i
stp
```

Si on exécute ce code dans la GPMachine, on obtient le résultat suivant :

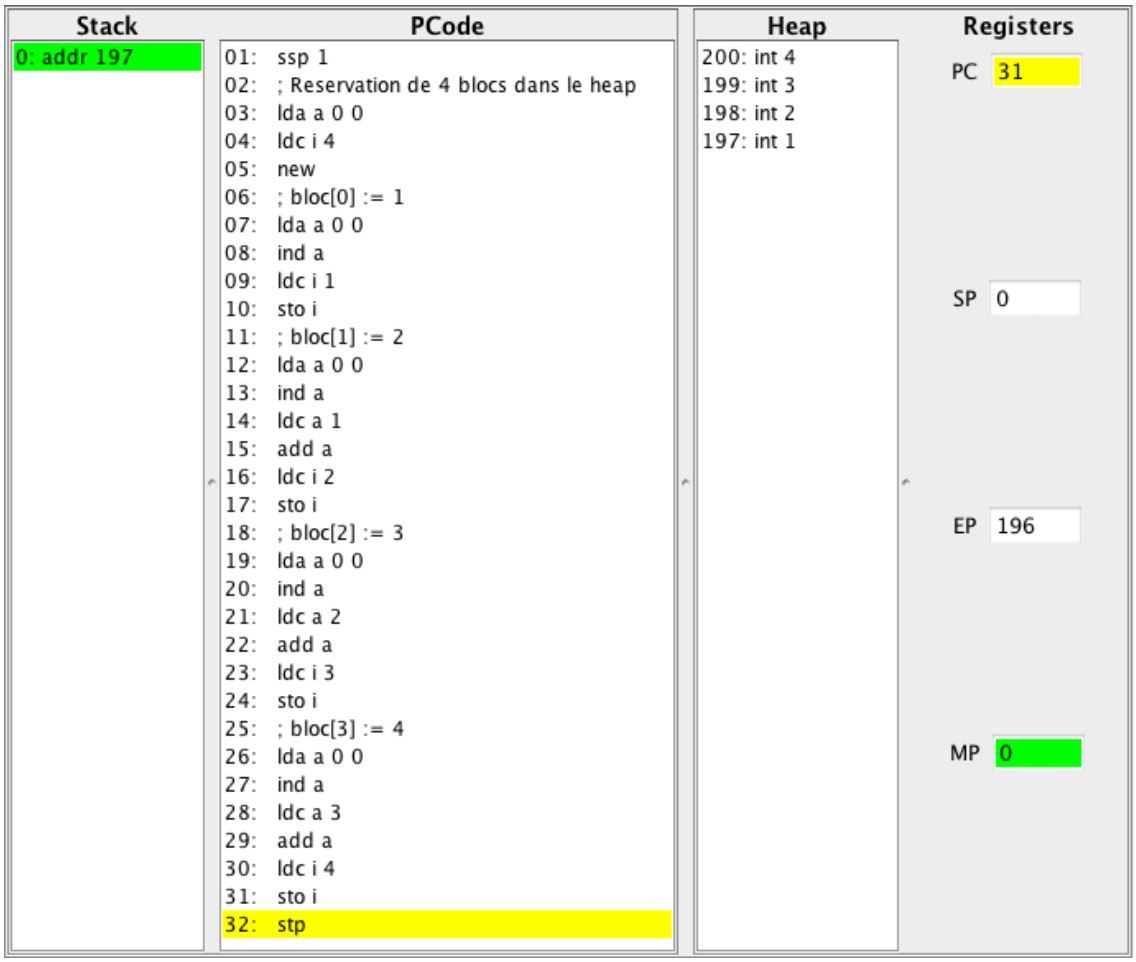
1. Passage des valeurs nécessaires à l'instruction `new` au sommet de la pile. `STORE[1]` contient l'adresse de la cellule mémoire dans laquelle l'adresse du premier bloc réservé sera placée. `STORE[2]` contient le nombre de cellules mémoires à réserver.



2. Exécution de l'instruction `new` :



3. Initialisation des cellules mémoires réservées sur le *heap* :


Stack
0: addr 197
PCode
01: ssp 1
02: ; Reservation de 4 blocs dans le heap
03: lda a 0 0
04: ldc i 4
05: new
06: ; bloc[0] := 1
07: lda a 0 0
08: ind a
09: ldc i 1
10: sto i
11: ; bloc[1] := 2
12: lda a 0 0
13: ind a
14: ldc a 1
15: add a
16: ldc i 2
17: sto i
18: ; bloc[2] := 3
19: lda a 0 0
20: ind a
21: ldc a 2
22: add a
23: ldc i 3
24: sto i
25: ; bloc[3] := 4
26: lda a 0 0
27: ind a
28: ldc a 3
29: add a
30: ldc i 4
31: sto i
32: stp
Heap
200: int 4
199: int 3
198: int 2
197: int 1
Registers
PC 31
SP 0
EP 196
MP 0