PMachine : Guide d'utilisation

James Ortiz

[INFOB314/IHDCB332] Théorie des langages : Syntaxe et sémantique

Table des matières

---

1 Introduction

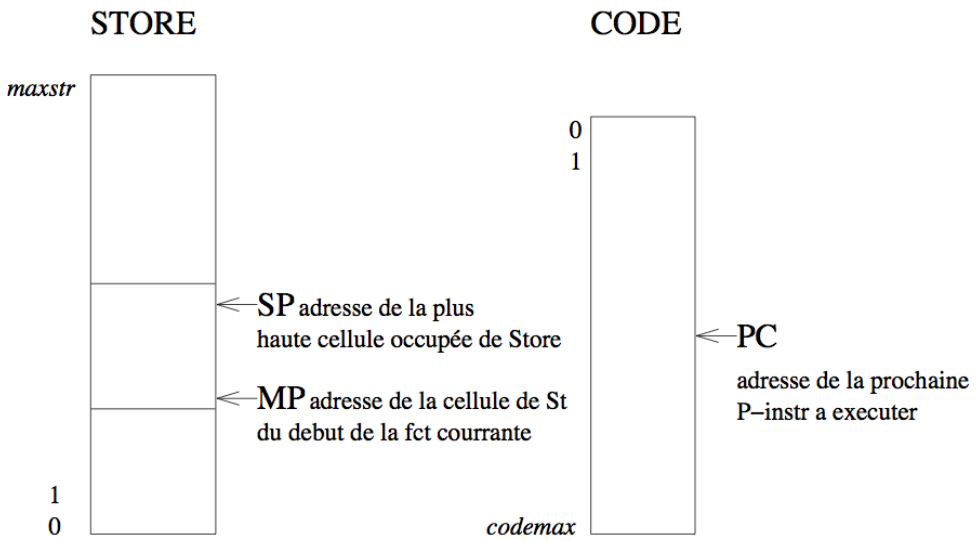


FIGURE 1 – Structure de la GPMachine

La PMachine possède [1] :
— 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 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.

Celle ci manipulera des éléments de type :
— entier ("i"), réel ("r"), booléen ("b"), adresse ("a") ;

1. À noter que cette implémentation ne dispose pas d'un tas (*heap*), les variables sont à stocker au niveau de la pile comme cela sera montré dans ce document.

— "N" représente un type numérique : $N \in \{i, a\}$;
— "S" représente un type scalaire : $S \in \{i, b\}$;
— "T" représente un type quelconque : $T \in \{i, a, b\}$.

Remarquez que le P-Code a été créé pour la traduction de Pascal, le langage $B314$n'utilisera qu'une petite partie du P-Code. Ce document ne présentera que la partie nécessaire à la traduction de programmes $B314$, exécutable sur la PMachine qui vous est donnée.

2 Instructions

2.1 P-Instructions pour les expressions

Les instructions permettant d'effectuer des opérations numériques et booléennes fonctionnent sur le principe de la notation polonaise inversée : il faut d'abord introduire les deux opérandes avant d'effectuer l'opération. Ainsi, pour une opération binaire, l'on va d'abord charge dans la pile la première opérande qui se trouvera en $STORE[SP-1]$, puis la deuxième opérande qui se trouvera $STORE[SP]$, avant d'effectuer l'opération qui consommera les deux valeurs et stockera le résultat au somme de la pile (qui sera alors $STORE[SP-1]$).

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)
mod N	STORE[SP-1] := STORE[SP-1] $\%_N$ STORE[SP] ; SP := SP − 1	(N,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)
grt S	STORE[SP-1] := STORE[SP-1] $>_S$ 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

Les lectures et écritures en mémoire permettent de charger des constantes au sommet de la pile (`ldc`) ; de charger le contenu d'une case (dédiée au stockage de la valeur d'une variable par exemple) de la pile (`lod`) ou son adresse (`lda`) ; de charger, sur base de son adresse se situant au somme de la pile, le contenu d'une case de la pile (`ind`) ; de stocker une valeur dans la pile sur base de son adresse (`sto`) ; de calculer l'adresse d'une case de la pile (représentant une case de tableau par exemple) sur base d'une adresse de base et d'un offset (`ixa`) ; et de supprimer une valeur au sommet de la pile (`pop`).

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)	()
ixa v	STORE[SP-1]:= STORE[SP - 1] + STORE[SP] * q ; SP := SP − 1	(a,i)	(a)
pop	SP := SP − 1		

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 P-Instructions de branchement

Les branchements en PCode se font vers un label défini au niveau de code via l'instruction `define`. Le branchement peut être direct (`ujp`) ou se faire uniquement si la valeur au sommet de la pile est fausse (`fjp`).

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

2.4 P-instructions pour les procédures et fonctions

Les instructions suivantes sont utilisées lors de la définition et l'appel de procédures et fonctions. L'appel d'une procédure ou d'une fonction se fait de la manière suivante : réserver l'espace

mémoire utilisé pour le bloc d'appel (contenant entre autre l'adresse de retour) via l'instruction `mst` ; réserver l'espace pour les paramètres et brancher vers le label de la procédure ou de la fonction via l'instruction `cup` ; au début de l'exécution de la procédure ou de la fonction, il est nécessaire de décaler le stack pointer (`SP`) afin de réserver l'espace mémoire nécessaire aux paramètres et aux variables locales via l'instruction `ssp` (qui sera aussi utilisée au début du programme pour les variables globales) ; enfin, au retour de la procédure ou de la fonction, il est nécessaire de restaurer le contexte de l'appelant via l'instruction `retp` (pour les procédures) ou `retf` (pour les fonctions). À noter que l'instruction `retf` aura aussi pour effet, une fois le contexte de l'appelant restauré, de placer au sommet de la pile la valeur de retour de la fonction (cette valeur est stockée à l'adresse 0 du bloc d'appel introduit via l'instruction `mst`).

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

2.5 P-Instructions d'arrêt du programme

Il y a deux manières d'arrêter un programme : sur une erreur (déclenchée par l'instruction `chk`) ou en utilisant l'instruction `stp`. L'instruction `chk` est utilisée pour vérifier les bornes d'un tableau lors de l'exécution et éviter les accès non autorisés en mémoire :

P-Inst	Signification	Condition	Résultat
chk k l	if STORE[SP]<k or STORE[SP]>l then error('value out of range')	(i) Type(k)=Type(l)=(i)	(i)

L'instruction permettant d'arrêter le programme :

P-Inst	Effet
stp	stoppe l'exécution de la PMachine

3 Traduction en P-Code

3.1 Traduction d'une instruction d'affectation

L'idée pour la traduction d'instructions en PCode est de travailler récursivement. La traduction d'un élément se fait donc sur base des traductions de ses sous-parties. À noter que l'on fera la différence entre la génération du PCode pour une expression gauche ou pour une expression droite. 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 $Type(z) = T$

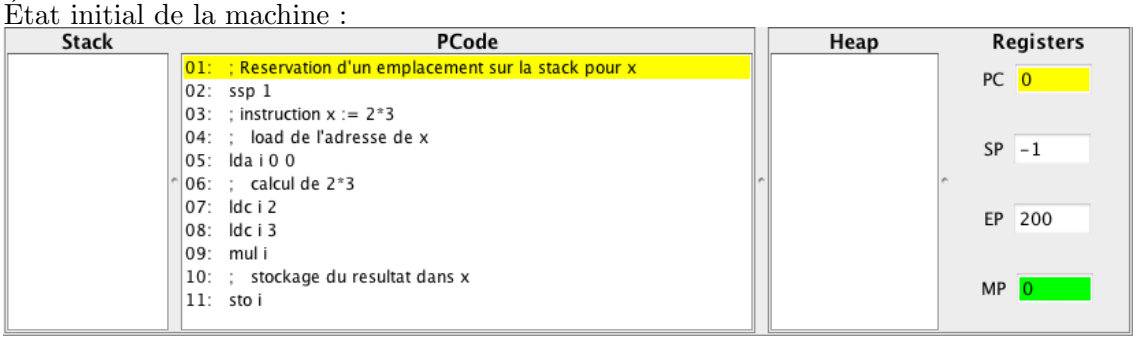
Où $d(z)$ et $q(z)$ sont respectivement la profondeur relative et l'adresse relative de z.

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

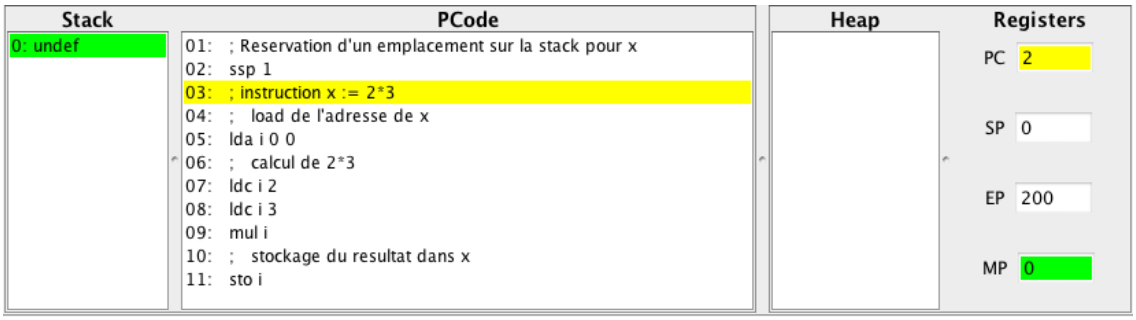
$$
\begin{aligned}
&PCode(x := 2 * 3)\\
&\quad = PCode_G(x);\ PCode_D(2 * 3);\ sto\ i\\
&\quad = lda\ i\ d(x)\ q(x);\ PCode_D(2 * 3);\ sto\ i\\
&\quad \vdots\\
&\quad = lda\ i\ d(x)\ q(x);\ ldc\ i\ 2;\ ldc\ i\ 3;\ mul\ i;\ sto\ i
\end{aligned}
$$

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 :

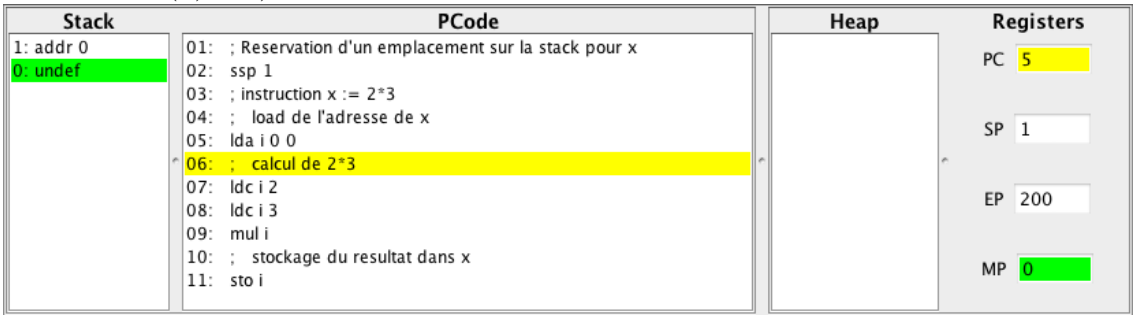
1. État initial de la machine :



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

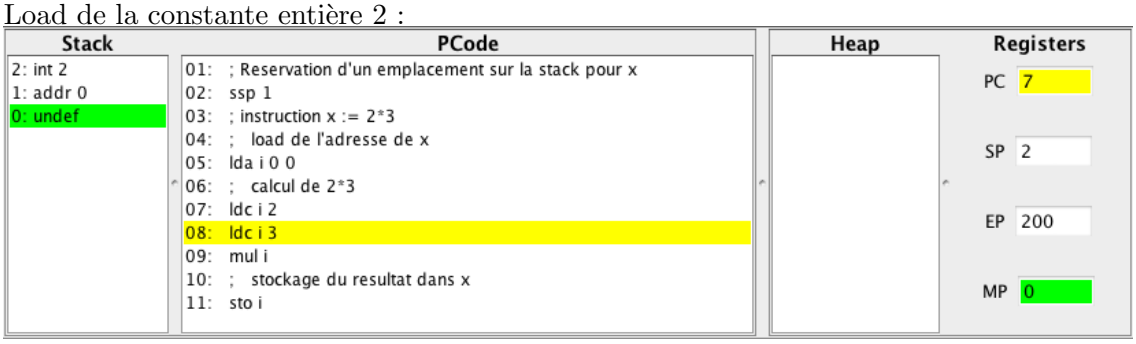


3. Load de l'adresse de x situé en position 0 ($q(x) = 0$) du contexte courant (la différence de profondeur $d(x) = 0$) :

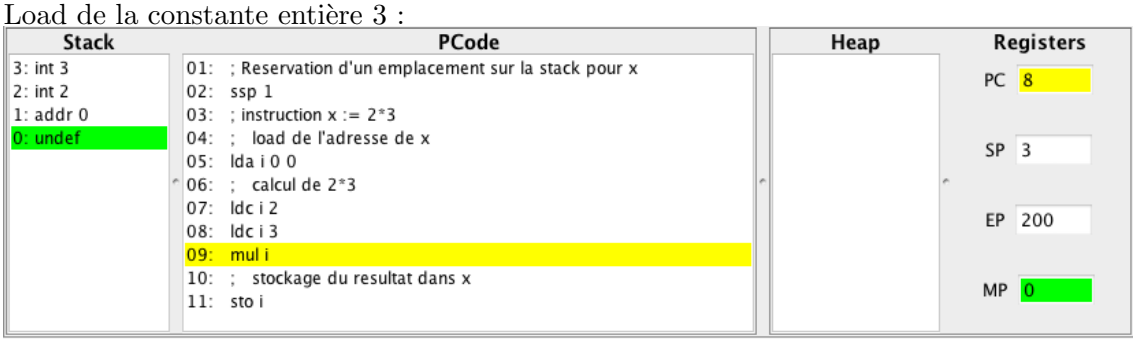


À noter que si l'on avait voulu accéder depuis une fonction $fct()$ à une variable a déclarée dans la fonction $fctPere()$ parente de $fct()$, la différence de profondeur $d(a)$ aurait été de 1. De même, si a avait été déclarée dans $fctGrandPere()$, $d(a)$ vaudrait 2, etc.

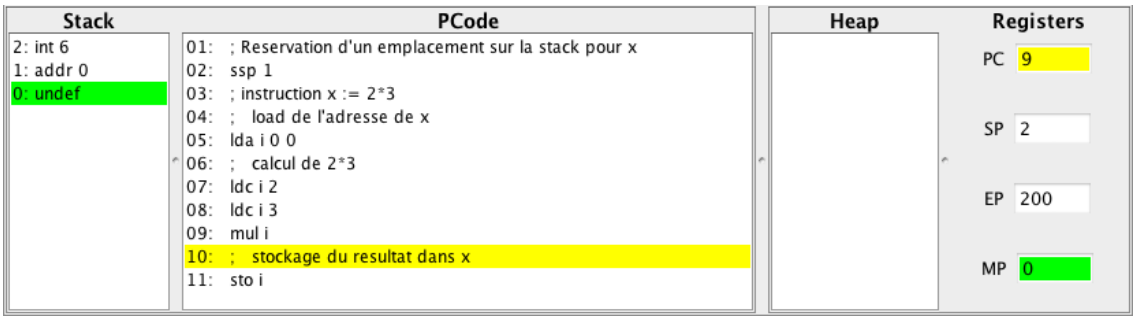
4. Load de la constante entière 2 :



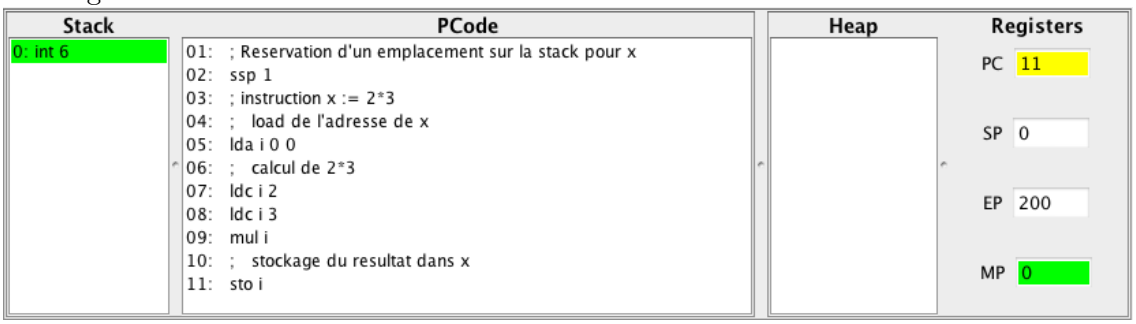
5. Load de la constante entière 3 :



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 :

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

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

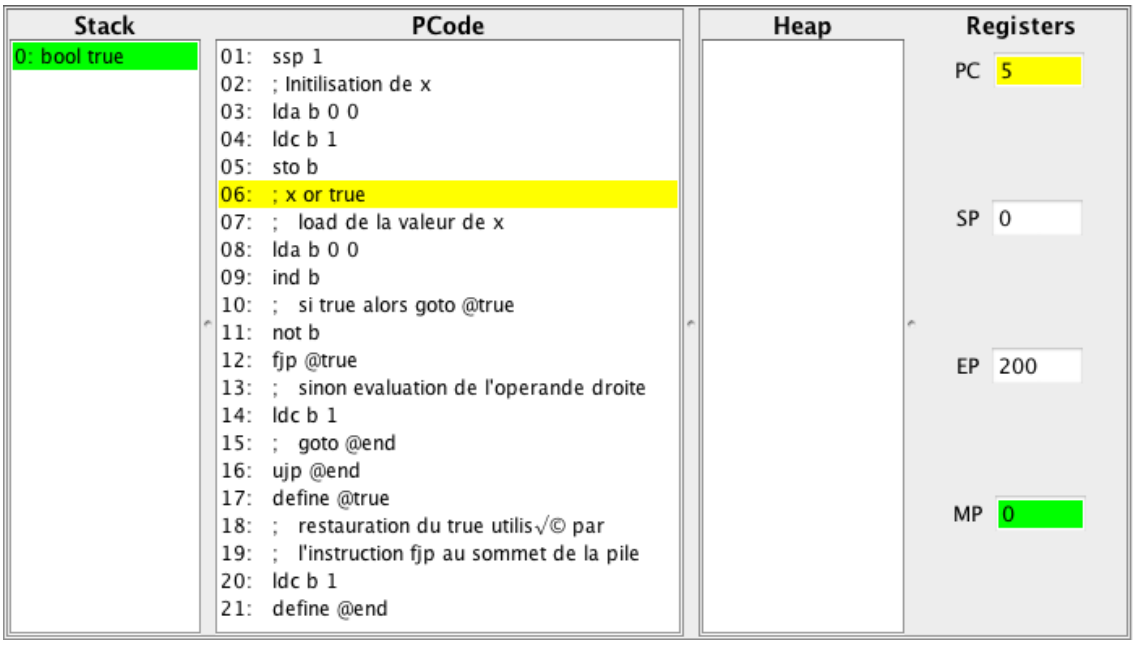
$$
\begin{aligned}
&PCode(E_1\ or\ E_2)\\
&\quad = PCode_D(E_1);\\
&\quad\ \ not\\
&\quad\ \ fjp\ @true\\
&\quad\ \ PCode_D(E_2)\\
&\quad\ \ ujp\ @end\\
&\quad\ \ define\ @true\\
&\quad\ \ ldc\ b\ 1\\
&\quad\ \ define\ @end
\end{aligned}
$$

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

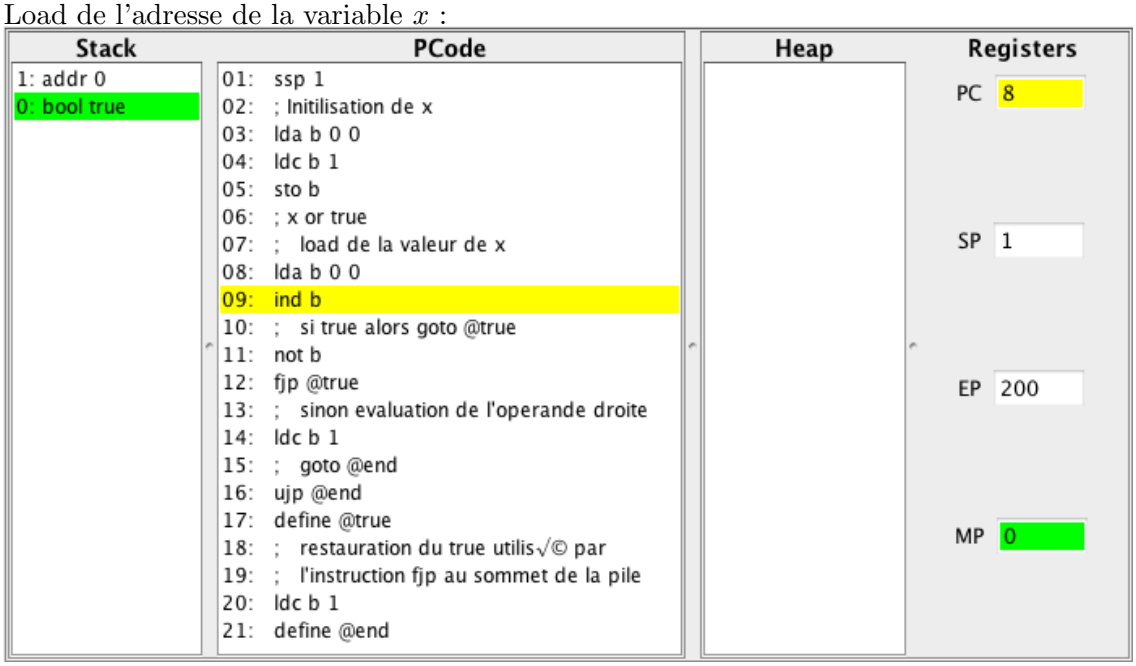
$$
\begin{aligned}
&PCode(xorfalse)\\
&\quad = PCode_D(x);\ not;\ fjp\ @true;\ PCode_D(true);\\
&\quad\ \ ujp\ @end;\ define\ @true;\ ldc\ b\ 1;\ define\ @end\\
&\quad = lda\ i\ d(x)\ q(x);\ ind\ b;\ not\ b;\ fjp\ @true;\\
&\quad\ \ ldc\ b\ 1;\ ujp\ @end;\ define\ @true;\ ldc\ b\ 1;\ define\ @end
\end{aligned}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

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

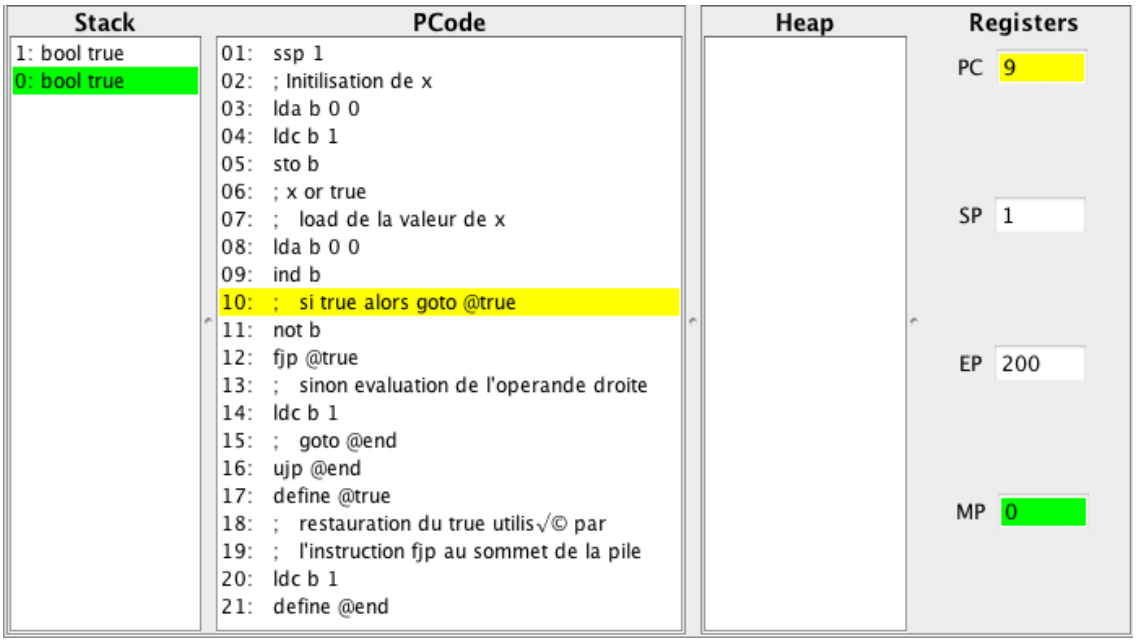
1. État initial de la machine (après initialisation de la variable x) :



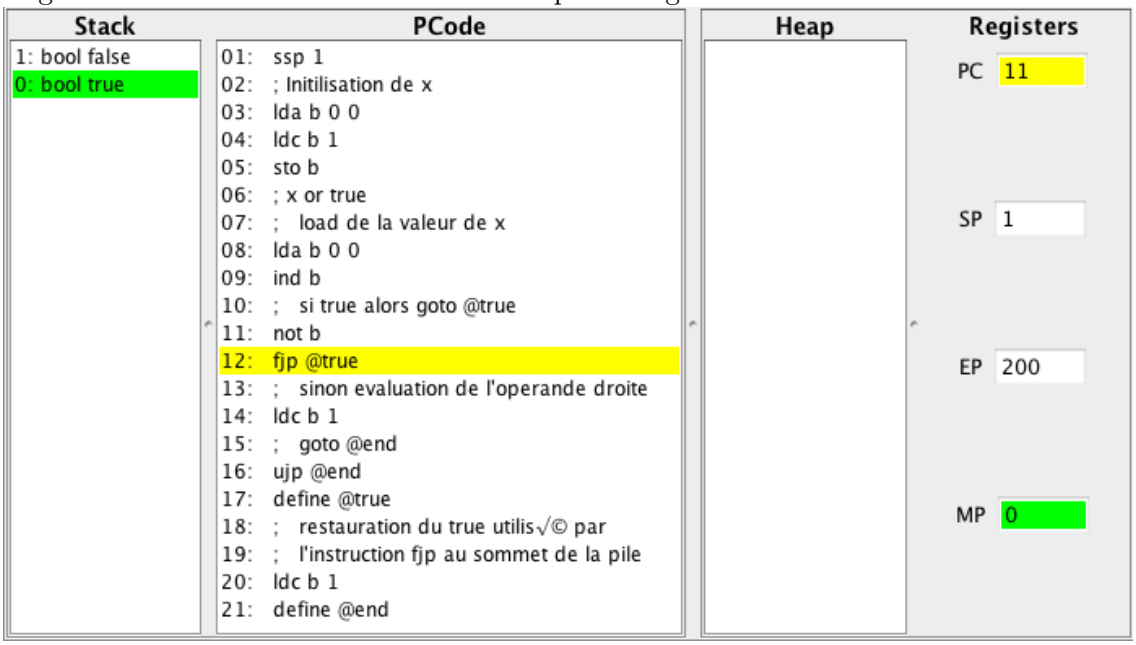
2. Load de l'adresse de la variable x :



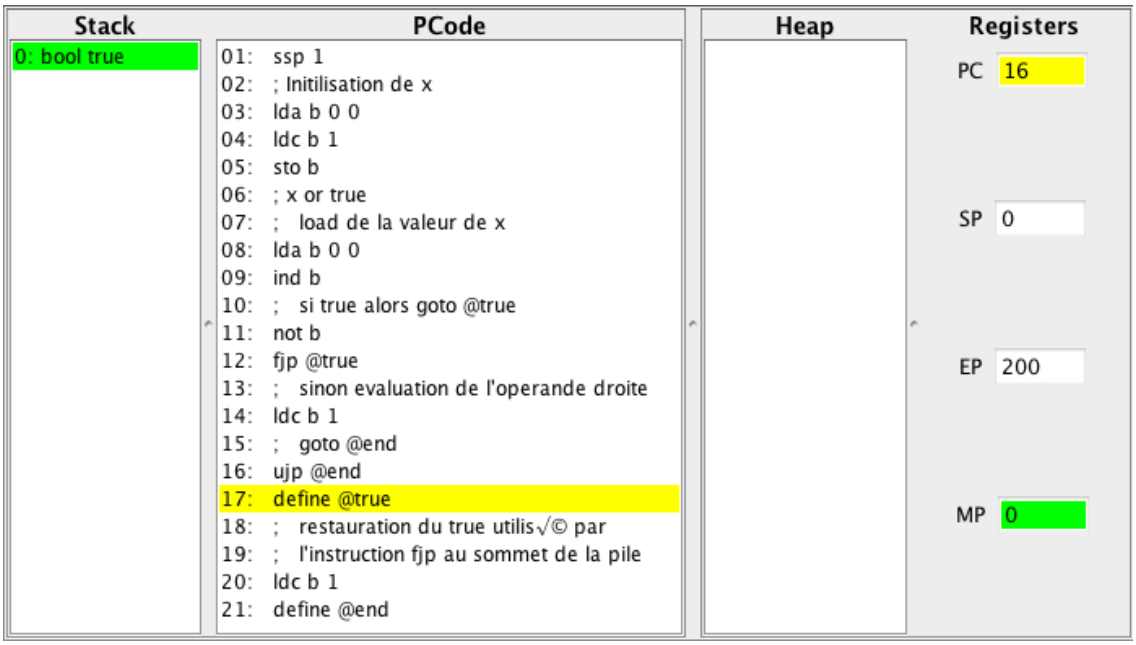
3. Récupération de la valeur stockée à cette adresse :



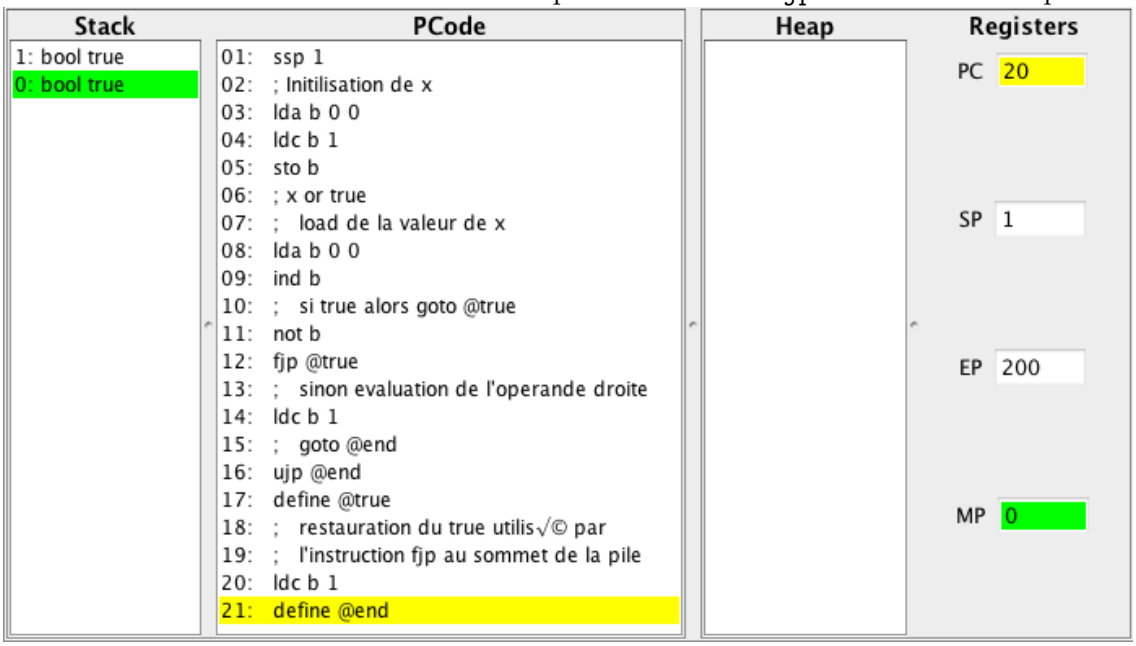
4. Négation du résultat de l'évaluation de l'expression gauche :



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 :



3.2 Autres traductions

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

3.3 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)

3.4 Traduction des procédures, fonctions et paramètres

3.4.1 PCode pour la déclaration d'une procédure

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

3.4.2 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

3.4.3 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 :

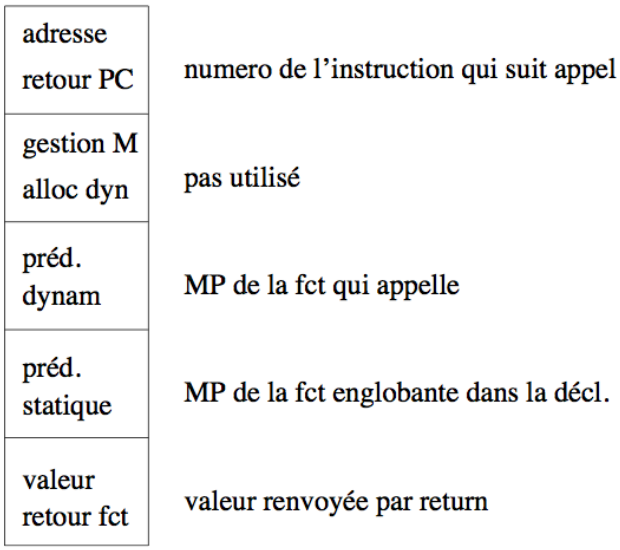
Fonction		Condition
$PCode(return\ e) =$	$lda\ T\ 0\ 0$ $PCode_D(e)$ $sto\ T$ $retf$	$Type(e) = T$

3.4.4 PCode pour un appel de fonction

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

3.4.5 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`.



3.4.6 Exemple d'appel de fonction

L'exemple suivant (en Pascal) présente un appel de fonction avec passage de paramètres et le renvoi d'une valeur au programme appelant.

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