

GPMachine : Guide d'utilisation*

Xavier Devroey

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

Table des matières

1	Introduction	1
2	Instructions	3
2.1	P-Instructions pour les expressions	3
2.1.1	Expressions numériques	3
2.1.2	Expressions booléennes	3
2.2	Lecture et écriture en mémoire	4
2.3	Traduction d'une instruction d'affectation	4
2.4	P-Instructions de branchement	6
2.5	Autres traductions	10
2.6	Traduction d'un programme	11
2.7	P-instructions pour les procédures et fonctions	11
2.8	Traduction des procédures, fonctions et paramètres	11
2.8.1	PCode pour la déclaration d'une procédure	11
2.8.2	PCode pour un appel de procédure	12
2.8.3	PCode pour la déclaration d'une fonction	12
2.8.4	PCode pour un appel de fonction	12
2.8.5	Bloc d'appel de procédure/fonctions	13
2.8.6	Exemple d'appel de fonction	13
2.9	Utilisation du tas (heap)	14
2.9.1	Exemple d'utilisation du tas	14

1 Introduction

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*

*Ce document a été élaboré en partie à partir des notes de Hubert Toussaint.

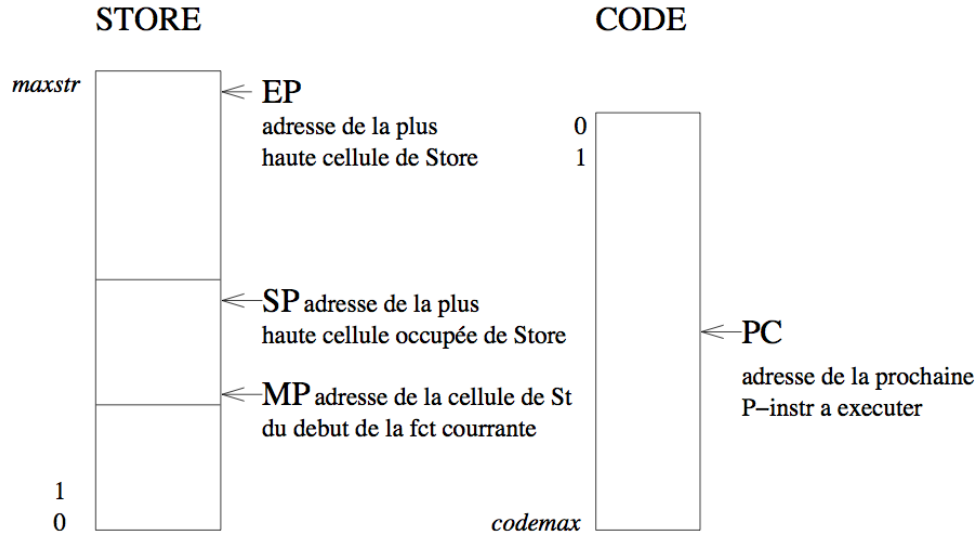


FIGURE 1 – Structure de la GPMachine

- 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	$\text{STORE}[\text{SP}-1] := \text{STORE}[\text{SP}-1] +_N \text{STORE}[\text{SP}] ;$ $\text{SP} := \text{SP} - 1$	(N,N)	(N)
sub N	$\text{STORE}[\text{SP}-1] := \text{STORE}[\text{SP}-1] -_N \text{STORE}[\text{SP}] ;$ $\text{SP} := \text{SP} - 1$	(N,N)	(N)
mul N	$\text{STORE}[\text{SP}-1] := \text{STORE}[\text{SP}-1] *_N \text{STORE}[\text{SP}] ;$ $\text{SP} := \text{SP} - 1$	(N,N)	(N)
div N	$\text{STORE}[\text{SP}-1] := \text{STORE}[\text{SP}-1] /_N \text{STORE}[\text{SP}] ;$ $\text{SP} := \text{SP} - 1$	(N,N)	(N)
neg N	$\text{STORE}[\text{SP}] := -_N \text{STORE}[\text{SP}]$	(N)	(N)

Où (N,N) indique que $\text{STORE}[\text{SP}]$ et $\text{STORE}[\text{SP}-1]$ sont de type numérique.

2.1.2 Expressions booléennes

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

$PCode(x := 2 * 3)$
 $= PCode_G(x); PCode_D(2 * 3); sto i$
 $= lda i d(x) q(x); PCode_D(2 * 3); sto i$
 \vdots
 $= lda i d(x) q(x); ldc i 2; ldc i 3; mul i; sto i$

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 :

Stack	PCode	Heap	Registers
	01: ; Reservation d'un emplacement sur la stack pour x		PC 0
	02: ssp 1		SP -1
	03: ; instruction x := 2*3		EP 200
	04: ; load de l'adresse de x		MP 0
	05: lda i 0 0		
	06: ; calcul de 2*3		
	07: ldc i 2		
	08: ldc i 3		
	09: mul i		
	10: ; stockage du resultat dans x		
	11: sto i		

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

Stack	PCode	Heap	Registers
0: undef	01: ; Reservation d'un emplacement sur la stack pour x		PC 2
	02: ssp 1		SP 0
	03: ; instruction x := 2*3		EP 200
	04: ; load de l'adresse de x		MP 0
	05: lda i 0 0		
	06: ; calcul de 2*3		
	07: ldc i 2		
	08: ldc i 3		
	09: mul i		
	10: ; stockage du resultat dans x		
	11: sto i		

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

Stack	PCode	Heap	Registers
1: addr 0	01: ; Reservation d'un emplacement sur la stack pour x		PC 5
0: undef	02: ssp 1		SP 1
	03: ; instruction x := 2*3		EP 200
	04: ; load de l'adresse de x		MP 0
	05: lda i 0 0		
	06: ; calcul de 2*3		
	07: ldc i 2		
	08: ldc i 3		
	09: mul i		
	10: ; stockage du resultat dans x		
	11: sto i		

À no-

ter 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 :

Stack	PCode	Heap	Registers
2: int 2	01: ; Reservation d'un emplacement sur la stack pour x		PC 7
1: addr 0	02: ssp 1		SP 2
0: undef	03: ; instruction x := 2*3		EP 200
	04: ; load de l'adresse de x		MP 0
	05: lda i 0 0		
	06: ; calcul de 2*3		
	07: ldc i 2		
	08: ldc i 3		
	09: mul i		
	10: ; stockage du resultat dans x		
	11: sto i		

5. Load de la constante entière 3 :

Stack	PCode	Heap	Registers
3: int 3 2: int 2 1: addr 0 0: undef	01: ; Reservation d'un emplacement sur la stack pour x 02: ssp 1 03: ; instruction x := 2*3 04: ; load de l'adresse de x 05: lda i 0 0 06: ; calcul de 2*3 07: ldc i 2 08: ldc i 3 09: mul i 10: ; stockage du resultat dans x 11: sto i		PC 8 SP 3 EP 200 MP 0

6. Calcul de la multiplication :

Stack	PCode	Heap	Registers
2: int 6 1: addr 0 0: undef	01: ; Reservation d'un emplacement sur la stack pour x 02: ssp 1 03: ; instruction x := 2*3 04: ; load de l'adresse de x 05: lda i 0 0 06: ; calcul de 2*3 07: ldc i 2 08: ldc i 3 09: mul i 10: ; stockage du resultat dans x 11: sto i		PC 9 SP 2 EP 200 MP 0

7. Stockage du résultat dans x :

Stack	PCode	Heap	Registers
0: int 6	01: ; Reservation d'un emplacement sur la stack pour x 02: ssp 1 03: ; instruction x := 2*3 04: ; load de l'adresse de x 05: lda i 0 0 06: ; calcul de 2*3 07: ldc i 2 08: ldc i 3 09: mul i 10: ; stockage du resultat dans x 11: sto i		PC 11 SP 0 EP 200 MP 0

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

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

2.4 P-Instructions de branchement

P-Inst	Effet	Commentaire
define @k		définition d'une étiquette
ujp @k	PC := adresse(@k)	branchement incondtionnel
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 :

```

PCode( $E_1$  or  $E_2$ )
= PCodeD( $E_1$ );
  not
  fjp @true
  PCodeD( $E_2$ )
  ujp @end
  define @true
  ldc b 1
  define @end

```

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

```

PCode(xorfalse)
= PCodeD(x); not; fjp @true; PCodeD(true);
  ujp @end; define @true; ldc b 1; define @end
= lda i d(x) q(x); ind b; not b; fjp @true;
  ldc b 1; ujp @end; define @true; ldc b 1; define @end

```

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*) :

Stack	PCode	Heap	Registers
0: bool true	01: ssp 1		PC 5
	02: ; Initialisation de x		
	03: lda b 0 0		
	04: ldc b 1		
	05: sto b		
	06: ; x or true		
	07: ; load de la valeur de x		SP 0
	08: lda b 0 0		
	09: ind b		
	10: ; si true alors goto @true		
	11: not b		
	12: fjp @true		EP 200
	13: ; sinon evaluation de l'operande droite		
	14: ldc b 1		
	15: ; goto @end		
	16: ujp @end		
	17: define @true		
	18: ; restauration du true utilis√© par		
	19: ; l'instruction fjp au sommet de la pile		
	20: ldc b 1		
	21: define @end		MP 0

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

Stack	PCode	Heap	Registers
1: addr 0	01: ssp 1		PC 8
0: bool true	02: ; Initialisation de x		
	03: lda b 0 0		
	04: ldc b 1		
	05: sto b		
	06: ; x or true		
	07: ; load de la valeur de x		SP 1
	08: lda b 0 0		
	09: ind b		
	10: ; si true alors goto @true		
	11: not b		
	12: fjp @true		EP 200
	13: ; sinon evaluation de l'operande droite		
	14: ldc b 1		
	15: ; goto @end		
	16: ujp @end		
	17: define @true		
	18: ; restauration du true utilis√© par		
	19: ; l'instruction fjp au sommet de la pile		
	20: ldc b 1		
	21: define @end		MP 0

3. Récupération de la valeur stockée à cette adresse :

Stack	PCode	Heap	Registers
1: bool true	01: ssp 1		PC 9
0: bool true	02: ; Initialisation de x		
	03: lda b 0 0		
	04: ldc b 1		
	05: sto b		
	06: ; x or true		
	07: ; load de la valeur de x		SP 1
	08: lda b 0 0		
	09: ind b		
	10: ; si true alors goto @true		
	11: not b		
	12: fjp @true		EP 200
	13: ; sinon evaluation de l'operande droite		
	14: ldc b 1		
	15: ; goto @end		
	16: ujp @end		
	17: define @true		
	18: ; restauration du true utilis√© par		
	19: ; l'instruction fjp au sommet de la pile		
	20: ldc b 1		
	21: define @end		MP 0

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

Stack	PCode	Heap	Registers
1: bool false	01: ssp 1		PC 11
0: bool true	02: ; Initialisation de x		
	03: lda b 0 0		
	04: ldc b 1		
	05: sto b		
	06: ; x or true		
	07: ; load de la valeur de x		SP 1
	08: lda b 0 0		
	09: ind b		
	10: ; si true alors goto @true		
	11: not b		
	12: fjp @true		EP 200
	13: ; sinon evaluation de l'operande droite		
	14: ldc b 1		
	15: ; goto @end		
	16: ujp @end		
	17: define @true		MP 0
	18: ; restauration du true utilis√© par		
	19: ; l'instruction fjp au sommet de la pile		
	20: ldc b 1		
	21: define @end		

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

Stack	PCode	Heap	Registers
0: bool true	01: ssp 1		PC 16
	02: ; Initialisation de x		
	03: lda b 0 0		
	04: ldc b 1		
	05: sto b		
	06: ; x or true		
	07: ; load de la valeur de x		SP 0
	08: lda b 0 0		
	09: ind b		
	10: ; si true alors goto @true		
	11: not b		
	12: fjp @true		
	13: ; sinon evaluation de l'operande droite		EP 200
	14: ldc b 1		
	15: ; goto @end		
	16: ujp @end		
	17: define @true		MP 0
	18: ; restauration du true utilis√© par		
	19: ; l'instruction fjp au sommet de la pile		
	20: ldc b 1		
	21: define @end		

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

Stack	PCode	Heap	Registers
1: bool true	01: ssp 1		PC 20
0: bool true	02: ; Initialisation de x		
	03: lda b 0 0		
	04: ldc b 1		
	05: sto b		
	06: ; x or true		
	07: ; load de la valeur de x		SP 1
	08: lda b 0 0		
	09: ind b		
	10: ; si true alors goto @true		
	11: not b		
	12: fjp @true		EP 200
	13: ; sinon evaluation de l'operande droite		
	14: ldc b 1		
	15: ; goto @end		
	16: ujp @end		
	17: define @true		
	18: ; restauration du true utilis√© par		
	19: ; l'instruction fjp au sommet de la pile		
	20: ldc b 1		MP 0
	21: define @end		

2.5 Autres traductions

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

2.6 Traduction d'un programme

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

(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 $base(d,MP) := \text{if } (d=0) \text{ then } MP \text{ else } base(d-1,STORE[MP+1]) \text{ fi}$

2.8 Traduction des procédures, fonctions et paramètres

2.8.1 PCode pour la déclaration d'une procédure

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

2.8.2 PCode pour un appel de procédure

Fonction
$PCode(proc(e_1, e_2, \dots, e_n)) =$ $ \begin{array}{l} mst\ d \\ PCode_A(e_1) \\ PCode_A(e_2) \\ \dots \\ PCode_A(e_n) \\ cup\ p\ @proc \end{array} $

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
$PCode(ProcDecl) =$ $ \begin{array}{l} define\ @fct \\ ssp\ s \\ ujp\ @fctBody \\ PCode(FctDeclList) \\ define\ @fctBody \\ PCode(InstList) \\ retf \end{array} $

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
$PCode(return\ e) =$ $ \begin{array}{l} lda\ T\ 0\ 0 \\ PCode_D(e) \\ sto\ T \\ retf \end{array} $	$Type(e) = T$

2.8.4 PCode pour un appel de fonction

Fonction
$PCode(fct(e_1, e_2, \dots, e_n)) =$ $ \begin{array}{l} mst\ d \\ PCode_A(e_1) \\ PCode_A(e_2) \\ \dots \\ PCode_A(e_n) \\ cup\ p\ @fct \end{array} $

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**.

adresse retour PC	numero de l'instruction qui suit appel
gestion M alloc dyn	pas utilisé
préd. dynam	MP de la fct qui appelle
préd. statique	MP de la fct englobante dans la décl.
valeur retour fct	valeur renvoyée par return

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] ≤ 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.

Stack	PCode	Heap	Registers
2: int 4	01: ssp 1		PC 4
1: addr 0	02: ; Reservation de 4 blocs dans le heap		
0: undef	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		SP 2
	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		EP 200
	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		MP 0
	27: ind a		
	28: ldc a 3		
	29: add a		
	30: ldc i 4		
	31: sto i		
	32: stp		

2. Exécution de l'instruction **new** :

Stack	PCode	Heap	Registers
0: addr 197	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	200: undef 199: undef 198: undef 197: undef	PC 5 SP 0 EP 196 MP 0

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

Stack	PCode	Heap	Registers
0: addr 197	01: ssp 1	200: int 4	PC 31
	02: ; Reservation de 4 blocs dans le heap	199: int 3	
	03: lda a 0 0	198: int 2	
	04: ldc i 4	197: int 1	
	05: new		
	06: ; bloc[0] := 1		
	07: lda a 0 0		
	08: ind a		
	09: ldc i 1		SP 0
	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		EP 196
	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		MP 0