Projet Génie Logiciel

Étape C

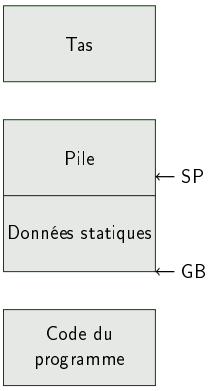
Génération de code

Projet GL

Ensimag
Grenoble INP

3 décembre 2023

Organisation de la mémoire à l'exécution dans IMA



données statiques variables globales, littéraux du programme, tables des méthodes, et valeurs temporaires

pile variables locales, paramètres de méthodes

tas objets alloués dynamiquement (pas utilisé dans le langage sans-objet)

GB (« Global Base ») Registre particulier qui pointe sur le fond de pile

SP (« Stack Pointer ») Pointe sur la dernière case occupée de la pile

Analogie/Différences avec l'assembleur x86 (Pentium)

- zone statique IMA = sections `.data`, `.rodata` et `.bss` en ELF x86.
- SP ≈ %esp (%rsp en 64 bits)
- GB n'a pas d'équivalent en x86

Principes de la génération de code pour expressions

- Évaluation de gauche à droite (sémantique de Deca)
- Calcul du résultat dans un registre banalisé $\geq$ R2, en utilisant d'autres registres pour sauvegarde des résultats de sous-expressions.
- Registres R0 et R1 = registres scratch (modifiables par les appels de méthodes), pas utilisés pour sauvegarder le résultat de sous-expressions.
- **RMAX** autorisé : `X-1` si option `-r X`, ou 15 sinon.
- Si plus de registre disponible, utiliser `PUSH` et `POP` pour sauvegarde des résultats sur la pile (c-à-d. des "temporaires").
  NB : attention à prise en compte dans calcul du `TSTO` (cf. plus loin).

Exemple de l'affectation

```
<codeExp(Assign[ Ident↑symb  e] , n)>
  := <codeExp(e, n)> // calcul de e dans Rn (avec n ∈ 2..MAX)
     STORE Rn, @symb
```


Génération de code naïve pour expressions arithmétiques

Opérande d'une expression atomique | **Mnémonique d'un opérateur binaire**

`<dval(IntLiteral↑n)> := #n`	`<mnemo(Plus)> := ADD`
`<dval(Identifier↑symb)> := @symb`	`<mnemo(Minus)> := SUB`
`<dval(_[ _ _] )> := ⊥`	`<mnemo(Mult)> := MUL`

Code pour calculer *e* dans R*n* (utilisant uniquement R0 et R*n* ... RMAX)

```
<codeExp(e,  n)>
avec <dval(e)>≠⊥
  := LOAD <dval(e)>, Rn

<codeExp(op[ e1  e2] , n)>
avec <dval(e2)>≠⊥
  := <codeExp(e1, n)>
     <mnemo(op)> <dval(e2)>, Rn
```

```
<codeExp(op[ e1  e2] , n)>
avec <dval(e2)>=⊥ et n=MAX
  := <codeExp(e1, n)>
     PUSH Rn ; sauvegarde
     <codeExp(e2, n)>
     LOAD Rn, R0
     POP Rn  ; restauration
     <mnemo(op)> R0, Rn
```

```
<codeExp(op[ e1  e2] , n)> avec <dval(e2)>=⊥ et n<MAX
  := <codeExp(e1, n)>
     <codeExp(e2, n+1)>
     <mnemo(op)> Rn+1, Rn
```


Exercice 1 : évaluation d'expression

On considère le programme suivant :

```
{
    int x = 1;
    int y = 2;
    int z;
    z = 2 * x - 3 * y;
}
```

Question

Dessiner la pile

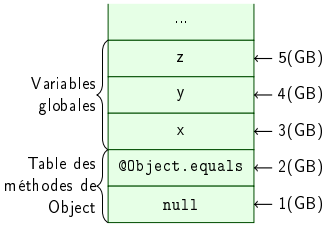
Question

Écrire l'assembleur généré

Exercice 1 : évaluation d'expression

- Source Deca :

```
{
    int x = 1;
    int y = 2;
    int z;
    z = 2 * x - 3 * y;
}
```



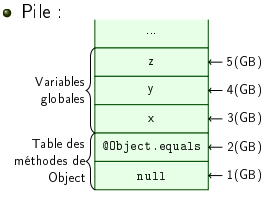
Rem : Table de méthodes de Object pas nécessaire sans objet.

Exercice 1 : évaluation d'expression

- Source Deca :

```
{
    int x = 1;
    int y = 2;
    int z;
    z = 2 * x - 3 * y;
}
```

- Pile :



- Code généré (**RMAX**≥R3) :

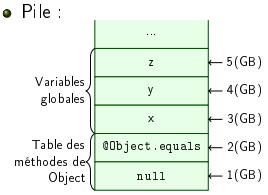
```
; int x = 1
LOAD #1, R2
STORE R2, 3(GB)
; int y = 2
LOAD #2, R2
STORE R2, 4(GB)
; z = 2 * x - 3 * y
LOAD #2, R2
MUL 3(GB), R2
LOAD #3, R3
MUL 4(GB), R3
SUB R3, R2
STORE R2, 5(GB)
```

Exercice 1 : évaluation d'expression

- Source Deca :

```
{
    int x = 1;
    int y = 2;
    int z;
    z = 2 * x - 3 * y;
}
```

- Pile :



- Code généré (**RMAX**=R2) :

```
; int x = 1
LOAD #1, R2
STORE R2, 3(GB)
; int y = 2
LOAD #2, R2
STORE R2, 4(GB)
; z = 2 * x - 3 * y
LOAD #2, R2
MUL 3(GB), R2
PUSH R2    ; sauvegarde
LOAD #3, R2
MUL 4(GB), R2
LOAD R2, R0
POP R2     ; restauration
SUB R0, R2
STORE R2, 5(GB)
```


Documentation

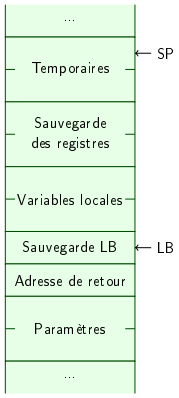
- Sémantique de Deca : II-[Semantique]
- Machine abstraite : II-[MachineAbstraite]
- Conventions de liaison : II-[ConventionsLiaison]
- Algorithmes de génération de code : IV-[Gencode]
- Outil fourni : `ima` (interprète de la machine abstraite), plus un metteur au point (voir IV-[Ima])
- À faire : I-[Consignes] (étape C)

Algorithmes et implantation de la génération de code

IV-[Gencode]

- Génération de code en deux passes :
  - Passe 1 : construction de la table des méthodes de chaque classe
    - Pas utile pour le langage sans-objet.
    - cf. IV-[Exemple]
  - Passe 2 : génération de code pour
    - le programme principal
    - chaque classe : initialisation des objets et codage des méthodes
    - messages d'erreurs pour les erreurs à l'exécution
- Parcours d'arbre en utilisant le patron « interprète », basé sur la grammaire d'arbres.

Bloc d'activation d'un appel de procédure



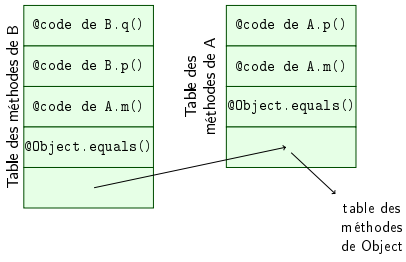
SP Pointeur de Pile, qui pointe sur le sommet de pile. (analogie avec l'assembleur x86 : SP ≈ %esp, ou %rip en 64 bits)

LB base locale, qui permet d'accéder aux paramètres, variables locales et temporaires dans la pile. (analogie avec l'assembleur x86 : LB ≈ %ebp, ou %rbp en 64 bits)

Table des méthodes

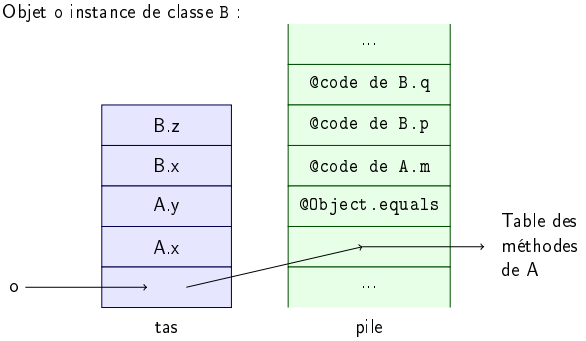
```
class A {
   int x;
   int y;
   void m() { }
   void p() { }
}

class B extends A {
   int x;
   int z;
   void p() { }
   void q() { }
}
```



Codage d'un objet

Objet o instance de classe B :



Exercice 2 : programme d'exemple

```
class Point2D {
   int x; // Abscisse
   int y; // Ordonnee
   // Deplace ce point
   // de a en diagonale.
   void diag(int a) {
      x = x + a;
      y = y + a;
   }
}

class Point3D extends Point2D {
   int z; // Hauteur
   // On redefinit la methode diag,
   // pour tenir compte de z
   void diag(int a) {
      x = x + a;
      y = y + a;
      z = z + a;
   }
}
```

```
{
   Point2D p1, p2;
   Point3D p3;

   p1 = new Point2D();
   p1.diag(1);

   p3 = new Point3D();
   p3.diag(2);

   p2 = p3;
   p2.diag(3);

   println("p3.z = ", p3.z);
}
```

a. Qu'affiche le programme ?
b. Dessiner l'état de la pile et du tas
c. Ecrire le code généré pour ce programme Deca

Exercice 2(a) : Sémantique du programme

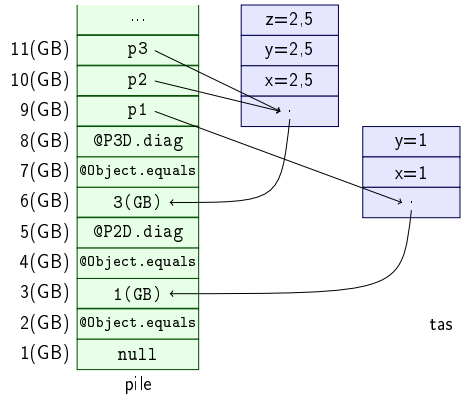
```
{
   Point2D p1, p2;
   Point3D p3;

   p1 = new Point2D(); // p1 initialise a zero :
                       // p1.x = 0; p1.y = 0
   p1.diag(1);         // p1.x = 1; p1.y = 1;

   p3 = new Point3D(); // p3 initialise a zero :
                       // p3.x = 0; p3.y = 0; p3.z = 0
   p3.diag(2);         // p3.x = 2; p3.y = 2; p3.z = 2

   p2 = p3;    // p2 et p3 representent le meme point
   p2.diag(3); // Liaison dynamique : Appel de Point3D.diag
               // p2.x = 5; p2.y = 5; p2.z = 5
   println("p3.z = ", p3.z); // p3.z = 5
}
```

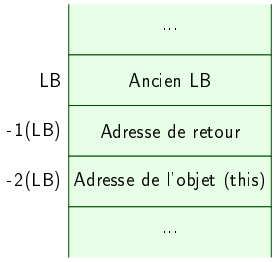
Exercice 2(b) : état de la pile et du tas



Construction des tables de méthodes

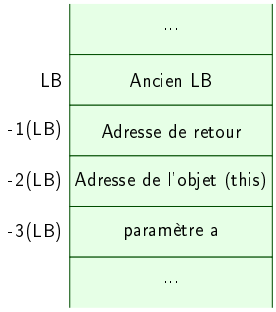
```
    ADDSP #11 (8 pour tables des methodes + 3 variables globales)
; Construction de la table des methodes de Object
    LOAD #null, R0
    STORE R0, 1 (GB)
    LOAD code.Object.equals, R0
    STORE R0, 2 (GB)
; Construction de la table des methodes de Point2D
    LEA 1 (GB), R0
    STORE R0, 3 (GB)
    LOAD code.Object.equals, R0
    STORE R0, 4 (GB)
    LOAD code.Point2D.diag, R0
    STORE R0, 5 (GB)
; Construction de la table des methodes de Point3D
    LEA 3 (GB), R0
    STORE R0, 6 (GB)
    LOAD code.Object.equals, R0
    STORE R0, 7 (GB)
    LOAD code.Point3D.diag, R0
    STORE R0, 8 (GB)
```


Initialisation à 0 des champs de Point2D



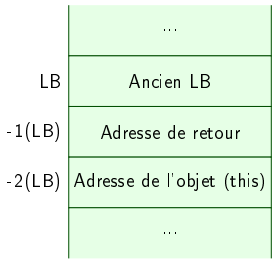

```
; Initialisation des
; champs de Point2D
init.Point2D :
   LOAD #0, R0
   LOAD -2(LB), R1
   STORE R0, 1(R1)
   LOAD #0, R0
   LOAD -2(LB), R1
   STORE R0, 2(R1)
   RTS
```


Code de la méthode diag de Point2D




```
; Code de la methode diag
; dans la classe Point2D
code.Point2D.diag :
; Sauvegarde des registres
    PUSH R2
    PUSH R3
; x = x + a
    LOAD -2 (LB), R2
    LOAD -2 (LB), R3
    LOAD 1 (R3), R3
    ADD -3 (LB), R3
    STORE R3, 1 (R2)
; y = y + a
    LOAD -2 (LB), R2
    LOAD -2 (LB), R3
    LOAD 2 (R3), R3
    ADD -3 (LB), R3
    STORE R3, 2 (R2)
; Restauration des registres
    POP R3
    POP R2
    RTS
```


Initialisation à 0 des champs de Point3D

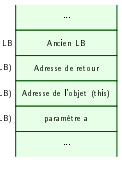



```
; Initialisation des
; champs de Point3D
init.Point3D :
     LOAD -2 (LB), R0
     PUSH R0
     BSR init.Point2D
     SUBSP #1
     LOAD #0, R0
     LOAD -2 (LB), R1
     STORE R0, 3 (R1)
     RTS
```


Code de la méthode diag de Point3D

```
; Code de la methode diag
; dans la classe Point3D
code.Point3D.diag :
; Sauvegarde des registres
     PUSH R2
     PUSH R3
; x = x + a
     LOAD -2 (LB), R2
     LOAD -2 (LB), R3
     LOAD 1 (R3), R3
     ADD -3 (LB), R3
     STORE R3, 1 (R2)
; y = y + a
     LOAD -2 (LB), R2
     LOAD -2 (LB), R3
     LOAD 2 (R3), R3
     ADD -3 (LB), R3
     STORE R3, 2 (R2)
```

```
; z = z + a
     LOAD -2 (LB), R2
     LOAD -2 (LB), R3
     LOAD 3 (R3), R3
     ADD -3 (LB), R3
     STORE R3, 3 (R2)
     POP R3
     POP R2
     RTS
```



Code du programme principal (1)

```
; p1 = new Point2D();
; (allocation + initialisation)
     NEW #3, R2
     BOV tas_plein
     LEA 3 (GB), R0
     STORE R0, 0 (R2)
     PUSH R2
     BSR init.Point2D
     POP R2
     STORE R2, 9 (GB)
```

```
; p1.diag(1);
     ADDSP #2
     ; empile p1
     LOAD 9 (GB), R2
     STORE R2, 0 (SP)
     ; empile 1
     LOAD #1, R2
     STORE R2, -1 (SP)
     ; appel de méthode
     LOAD 0 (SP), R2
     ; objet null dans
     ; appel de methode ?
     CMP #null, R2
     BEQ dereferencement_null
     ; adresse de la
     ; méthode diag de p1.
     LOAD 0 (R2), R2
     BSR 2 (R2)
     SUBSP #2
```


Code du programme principal (2)

```
; p3 = new Point3D() ;
     NEW #4, R2
     BOV tas_plein
     LEA 6 (GB), R0
     STORE R0, 0 (R2)
     PUSH R2
     BSR init.Point3D
     POP R2
     STORE R2, 11 (GB)
```

```
; p3.diag(2) ;
     ADDSP #2
     ; empile p3
     LOAD 11 (GB), R2
     STORE R2, 0 (SP)
     ; empile 2
     LOAD #2, R2
     STORE R2, -1 (SP)
     ; appel de méthode
     LOAD 0 (SP), R2
     ; objet null
     ; dans appel de methode ?
     CMP #null, R2
     BEQ dereferencement_null
     ; adresse de la
     ; méthode diag de p3.
     LOAD 0 (R2), R2
     BSR 2 (R2)
     SUBSP #2
```

Code du programme principal (3)

```
; p2 = p3;
     LOAD 11 (GB), R2
     STORE R2, 10 (GB)

; p2.diag(3);
; Liaison dynamique :
; Appel de Point3D.diag
;
; Comme précédemment
```

```
; println("p3.z = ", p3.z);
; affiche : "p3.z = 5"
     WSTR "p3.z = "
     LOAD 11 (GB), R2
     ; objet null dans
     ; selection de champ ?
     CMP #null, R2
     BEQ dereferencement_null
     LOAD 3 (R2), R2
     LOAD R2, R1
     WINT
     WNL
     HALT
```


Problème du TSTO

Lignes à ajouter aux extraits de code ci-dessus pour tester les débordements de pile :

```
code.Point2D.diag :
   TSTO #2
   BOV pile_pleine

code.Point3D.diag :
   TSTO #2
   BOV pile_pleine

init.Point3D :
   ; pour pouvoir appeler
   ; init.Point2D:
   TSTO #3
   BOV pile_pleine
```

```
; construction de la
; table des méthodes
; + programme principal
TSTO #15 ; (11 pour le
         ;     "ADDSP #11"
         ; + 4 pour le programme
         ;     principal)
BOV pile_pleine
```


Récapitulatif des erreurs à l'exécution

II [Semantique]

- erreurs de débordement :
  - pile
  - tas
  - arithmétique (sur flottants, inclus la division par 0.0)
- division entière par 0 et reste de la division entière par 0
- sortie de méthode sans passer par "return"
- conversion de type impossible
- déréférencement de null
- erreur de lecture (RINT et on ne tape pas un entier; RFLOAT et on ne tape pas un flottant)
- accès à des variables non initialisées (non traité : si on y accède ima donne un message d'erreur).