

Projet Langages et compilation (CS444)

Passe 3

Catherine Oriat
Ioannis Parissis

Grenoble INP-Esisar

2016-2017

Passe 3 : génération de code

Compilateur en trois passes

- Passe 1 : analyse lexicale et syntaxique
- Passe 2 : vérifications contextuelles
- Passe 3 : génération de code

Objectif

- On dispose de l'arbre abstrait décoré
- On parcourt cet arbre abstrait
- On génère du code pour une machine abstraite (proche de l'assembleur 68000)

Passe 3

- 1 Machine abstraite
- 2 Génération de code
- 3 Mise en oeuvre

Passe 3

- 1 Machine abstraite
 - Interprète de machine abstraite
 - Spécification de la machine abstraite

Interprète de machine abstraite

Le code de la machine abstraite n'est pas exécutable directement par une machine.

On dispose d'un interprète de machine abstraite (ima).

Installation :

- il faut installer Gnat (compilateur Ada)
- il faut compiler de code de la machine abstraite (code Ada)
commande : ima
- cf. Chamilo

Intérêt d'utiliser ce code intermédiaire : pouvoir générer ensuite du code assembleur pour différentes architectures de processeurs

Passe 3

- 1 Machine abstraite
 - Interprète de machine abstraite
 - Spécification de la machine abstraite

Spécification de la machine abstraite

Spécification : MachineAbstraite.txt

Guide utilisateur de ima : Ima.txt

Valeurs manipulées

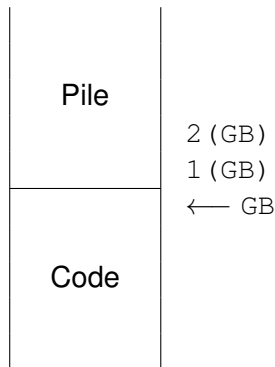
- entiers (32 bits)
- flottants (32 bits)
- adresses

codés sur 1 mot

typés (adresse, entier ou flottant).

Mémoire

- registres : R0, R1, R2, ... R15
- zone code (instructions du programme)
- pile



Modes d'adressage de la machine abstraite

- Immédiat : $\#1$, $\#1.5$
- Direct par registre : R_i
- Indirect avec déplacement : $d(R_i)$, $d(GB)$
- Indirect indexé avec déplacement : $d(R_i, R_j)$, $d(GB, R_j)$

Instructions de la machine abstraite

Transferts

- **Chargement dans un registre : LOAD**

LOAD #1, R0 ; $R0 \leftarrow 1$

LOAD 1(GB), R0 ; $R0 \leftarrow \text{contenu}(GB+1)$

LOAD 1(GB, R2), R0 ; $R0 \leftarrow \text{contenu}(GB+R2+1)$

- **Chargement à une adresse : STORE**

STORE R0, 1(GB) ; $\text{adr}(GB+1) \leftarrow R0$

STORE R0, 1(GB, R2) ; $\text{adr}(GB+R2+1) \leftarrow R0$

- **Chargement d'une adresse (Load Effective Address : LEA)**

LEA 1(GB), R0 ; $R0 \leftarrow \text{adr}(GB+1)$

LEA 1(GB, R2), R0 ; $R0 \leftarrow \text{adr}(GB+R2+1)$

- PUSH Rm
- POP Rm

Instructions de la machine abstraite

Opérations arithmétiques : entre flottants ou entre entiers

- ADD, SUB, MUL...

SUB #1, R0 ; $R0 \leftarrow R0 - 1$

Codes condition :

GT : >	GE : \geq	EQ : =	OV : overflow
LT : <	LE : \leq	NE : \neq	

Comparaisons : positionne les codes condition

- CMP #1, R0 : positionne les codes condition selon $R0 - 1$
CMP R1, R0 : positionne les codes condition selon $R0 - R1$
- Exemple : si $R0 = 2$, CMP #1, R0 positionne GT, GE, NE à vrai, et LT, LE, EQ à faux.

Instructions de la machine abstraite

Branchements

- `BRA etiq` : **branchement non conditionnel** à l'étiquette `etiq`
- `Bcc etiq` : **branchement si cc** à l'étiquette `etiq`
`BGT etiq`, `BGE etiq`, `BEQ etiq`, `BOV etiq`...

Entrées-sorties

- `RINT` : lecture d'un entier dans le registre `R1`
- `RFLOAT` : lecture d'un flottant dans le registre `R1`
- `WINT` : écriture de l'entier contenu dans le registre `R1`
- `WFLOAT` : écriture du flottant contenu dans `R1`
- `WSTR "str"` : écriture de la chaîne `"str"`
- `WNL` : écriture d'un retour à la ligne

Instructions de la machine abstraite

Divers

- ADDSP #5 : incrémente le pointeur de pile de 5 mots
- SUBSP #5 : décrémente le pointeur de pile de 5 mots
- TSTO #5 : teste s'il reste 5 mots dans la pile
- HALT : fin de programme

Passe 3

1 Machine abstraite

2 Génération de code

3 Mise en oeuvre

Passe 3

2 Génération de code

- Génération de code pour un programme JCas simple
- Génération de code pour les expressions arithmétiques
- Génération de code pour les expressions booléennes
- Codage des structures de contrôle
- Génération de code pour les tableaux

Exemple simplifié

```

program
  x, y : integer;
begin
  x := 3;
  y := x - 1;
  write("y = ", y);
  new_line;
end.

x : 1(GB), y : 2(GB)

```

```

ADDSP #2 ; variables globales
; Affectation ligne 4
LOAD #3, R0
STORE R0, 1(GB)
; Affectation ligne 5
LOAD 1(GB), R0
SUB #1, R0
; write ligne 6
WSTR "y = "
LOAD 2(GB), R1
WINT
; new_line ligne 7
WNL
HALT

```

Éléments supplémentaires

Débordements

- Débordements arithmétiques

Toute opération arithmétique peut provoquer un débordement

- Débordements d'intervalle

$y : 1..10;$

$y := x;$

Il faut vérifier que la valeur affectée à y est dans l'intervalle

$1..10.$

Passe 3

2 Génération de code

- Génération de code pour un programme JCas simple
- Génération de code pour les expressions arithmétiques
- Génération de code pour les expressions booléennes
- Codage des structures de contrôle
- Génération de code pour les tableaux

Sémantique du langage compilé : ordre d'évaluation des expressions arithmétiques

Selon les langages, les expressions arithmétique sont évaluées

- de gauche à droite : $e_1 + e_2$ évalue d'abords e_1 puis e_2 puis fait l'addition (Java)
- dans un ordre indifférent (C, C++, Ada)

Différence si on a un appel de fonction qui fait un effet de bord.

Exemples :

- $f(1) + i$, où f modifie la variable globale i
- $(++i) + i$

Langage JCas : n'impose pas d'ordre pour l'évaluation des expressions

Expressions arithmétiques : algorithme en une passe

Gestion des registres

On gère une liste de registres *alloués*. Lorsque le code est exécuté, ces registres contiennent une valeur qu'il ne faut pas écraser.

Les autres registres sont *libres*.

Procédure de génération de code

- Génère du code pour l'expression A tel que l'expression soit évaluée dans le registre R_c .
- Précondition : le registre R_c est alloué.

procedure Coder_Exp (A : Arbre ; R_c : Registre) ;

Génération de code pour les expressions arithmétiques

- $\text{Coder_Exp}(F, Rc) =$
Générer(LOAD, L_Opérande(F), Rc) ;
- $\text{Coder_Exp}(E1 \text{ Op } F, Rc) =$
Coder_Exp(E1, Rc) ;
Générer(Op, L_Opérande(F), Rc) ;

$\text{L_Opérande}(1) = \#1$

$\text{L_Opérande}(x) = \text{adresse de } x \text{ (ex : 1 (GB))}$

- ```

Coder_Exp(E1 Op E2, Rc) =
 if Reste_Registres then
 -- Il reste des registres : évaluation gauche-droite
 Coder_Exp(E1, Rc) ;
 Rd := Allouer_Reg ;
 Coder_Exp(E2, Rd) ;
 Générer(Op, Rd, Rc) ;
 Libérer(Rd) ;
 else
 -- Plus de registre : évaluation droite-gauche
 -- On alloue une variable temporaire
 Coder_Exp(E2, Rc) ;
 Temp := Allouer_Temp ;
 Générer(STORE Rc, Temp) ;
 Coder_Exp(E1, Rc) ;
 Générer(Op, Temp, Rc) ;
 Libérer_Temp ;
 end if ;

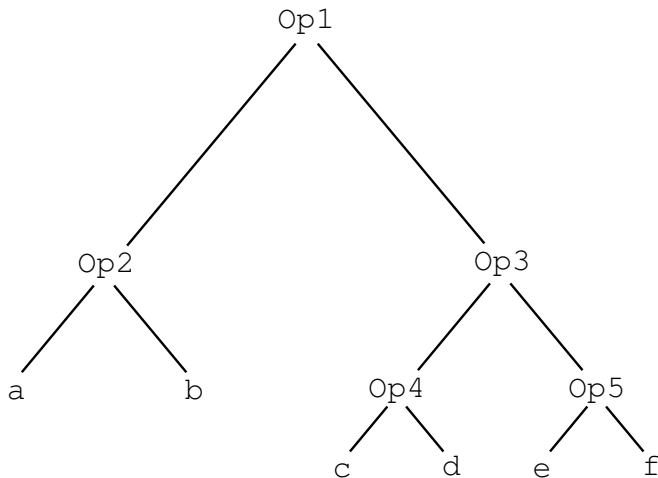
```

# Génération de code pour les expressions arithmétiques

En résumé :

- évaluation gauche-droite s'il reste des registres ;
- évaluation droite-gauche s'il ne reste plus de registre.

## Exemple



## Code généré

*; Évaluation de la partie gauche*

LOAD @a, R0

Op2 @b, R0

*; Évaluation de la partie droite*

*; R0 est occupé*

*; Sous-expression gauche*

LOAD @c, R1

Op4 @d, R1

*; Sous-expression droite*

LOAD @e, R2

Op5 @f, R2

Op3 R2, R1

Op1 R1, R0

⇒ Le code utilise 3 registres



## Code généré si on ne dispose que de 2 registres

*; Évaluation de la partie gauche*

LOAD @a, R0

Op2 @b, R0

*; Évaluation de la partie droite*

*; Sous-expression droite*

LOAD @e, R1

Op5 @f, R1

*; Sauve valeur intermédiaire dans une temporaire*

STORE R1, temp ; ex : temp = 3(GB)

*; Sous-expression gauche*

LOAD @c, R1

Op4 @d, R1

Op3 temp, R1

Op1 R1, R0

## Passe 3

### 2 Génération de code

- Génération de code pour un programme JCas simple
- Génération de code pour les expressions arithmétiques
- **Génération de code pour les expressions booléennes**
- Codage des structures de contrôle
- Génération de code pour les tableaux

## Modes d'évaluation de and et or

### Évaluation stricte

Les deux sous-expressions sont évaluées.

**A and B** = *vrai*, si *A = vrai* et *B = vrai*  
= *faux*, sinon

Exemple : en Java, *A & B* évalue les deux sous-expressions *A* et *B* successivement et dans cet ordre.

**A or B** = *faux*, si *A = faux* et *B = faux*  
= *vrai*, sinon

Exemple : en Java, *A | B* évalue les deux sous-expressions *A* et *B* successivement et dans cet ordre.

## Modes d'évaluation de and et or

### Évaluation paresseuse

La deuxième sous-expression est évaluée uniquement si cela est nécessaire.

**and** :  $B$  est évaluée uniquement si  $A$  est vrai.

$$\begin{aligned} A \text{ and } B &= \text{faux}, & \text{si } A = \text{faux} \\ &= B, & \text{sinon} \end{aligned}$$

Exemple : en Java,  $A \& \& B$  évalue  $A$ , puis, si  $A$  est vrai,  $B$ .

**or** :  $B$  est évaluée uniquement si  $A$  est faux.

$$\begin{aligned} A \text{ or } B &= \text{vrai}, & \text{si } A = \text{vrai} \\ &= B, & \text{sinon} \end{aligned}$$

Exemple : en Java,  $A \mid \mid B$  évalue  $A$ , puis, si  $A$  est faux,  $B$ .

## Différences entre les deux modes d'évaluation

- Lorsque  $B$  boucle ou provoque une erreur.

Exemple en Java : recherche d'un élément dans une liste.

```
while (l != null && l.val != v) {
 l = l.suiv;
}
```

`l.val` provoque une erreur lorsque `l = null`.

## Différences entre les deux modes d'évaluation

- Différence en efficacité si l'évaluation de  $B$  est coûteuse.

Exemple : recherche dans un tableau trié (précondition)

```
if (défensif && !a.estTrié()) {
 throw new RuntimeException("Préc incorrecte");
} else {
 val = a.rechercheDichotomique(v);
}
```

# Codage des expressions booléennes à l'aide d'entiers

Choix d'une (ou plusieurs) valeur qui correspond à vrai ;

Choix d'une (ou plusieurs) valeur qui correspond à faux.

Exemples :

1

|      |                   |   |
|------|-------------------|---|
| faux | $\leftrightarrow$ | 0 |
| vrai | $\leftrightarrow$ | 1 |

2

|      |                   |            |
|------|-------------------|------------|
| faux | $\leftrightarrow$ | 0          |
| vrai | $\leftrightarrow$ | $v \neq 0$ |

# Codage des expressions booléennes à l'aide d'entiers

- **Évaluation stricte**

Similaire aux expressions arithmétiques

Exemple : **and**  $\leftrightarrow$   $*$

- **Évaluation paresseuse**

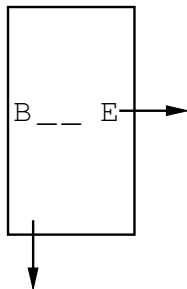
Il faut ajouter des tests pour ne pas évaluer la deuxième sous expression systématiquement.



## Codage des expressions booléennes par flots de contrôle

Codage adapté à l'évaluation paresseuse des expressions booléennes, lorsque le résultat de l'expression ne nécessite pas d'être stocké dans une variable.

***Expression booléenne : suite de lignes de code comportant un ou plusieurs branchements à une étiquette E.***



## Codage des booléens par flots de contrôle

Les deux valeurs booléennes correspondent

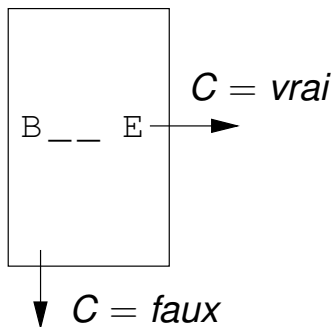
- au branchement à l'étiquette  $E$  (rupture de séquence) ;
- à la poursuite des instructions (continuation en séquence).

Pour une expression booléenne  $C$ , on a deux possibilités :

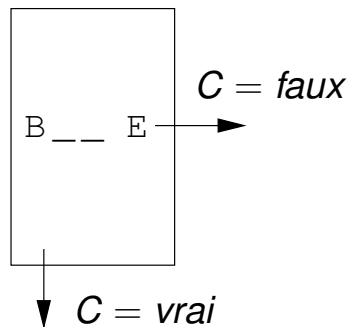
- 1 Le branchement est effectué si  $C$  est vrai ;
- 2 Le branchement est effectué si  $C$  est faux.

Pour distinguer les deux cas, on introduit un paramètre *Saut*, qui vaut *vrai* dans le premier cas et *faux* dans le second.

## Codage d'une expression booléenne



$Saut = \text{vrai}$



$Saut = \text{faux}$

## Procédure de génération de code

```
-- Génère du code pour l'expression booléenne C.
-- Si Saut = vrai, branchement en E si C est vrai
-- continue en séquence si C est faux.
-- Si Saut = faux, branchement en E si C est faux
-- continue en séquence si C est vrai.
```

**procedure** Coder\_Cond

(C : Arbre; Saut : Boolean; E : Etiquette);

## Générateur d'étiquettes

```
-- Génère une nouvelle étiquette.
```

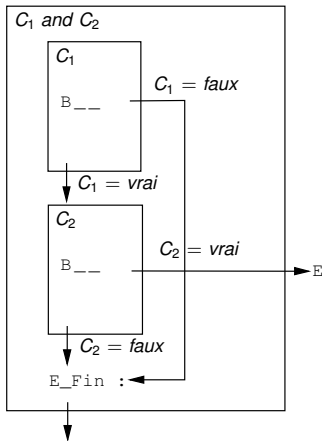
**function** Nouvelle\_Etiqu return Etiquette;

## Procédure Coder\_Cond

- `Coder_Cond(Noeud_Ident("true"), True, E) =`  
    Générer(BRA, E) ;
- `Coder_Cond(Noeud_Ident("true"), False, E) =`  
    null ;    *-- Il n'y a rien à faire*
- `Coder_Cond(Noeud_Ident("false"), True, E) =`  
    null ;    *-- Il n'y a rien à faire*
- `Coder_Cond(Noeud_Ident("false"), False, E) =`  
    Générer(BRA, E) ;

## Procédure Coder\_Cond

- `Coder_Cond(Noeud_And(C1, C2), True, E)`



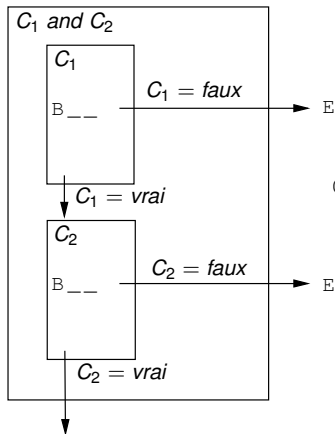
```

Coder_Cond(Noeud_And(C1,C2), True, E) =
declare
 E_Fin : Etiq := Nouvelle_Etiq;
begin
 Coder_Cond(C1, False, E_Fin);
 Coder_Cond(C2, True, E);
 Générer_Etiq(E_Fin);
end ;

```

## Procédure Coder\_Cond

- `Coder_Cond(Noeud_And(C1, C2), False, E)`



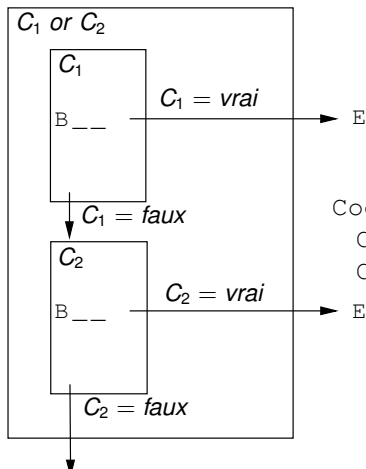
```

Coder_Cond(Noeud_And(C1, C2), False, E) =
 Coder_Cond(C1, False, E);
 Coder_Cond(C2, False, E);

```

## Procédure Coder\_Cond

- `Coder_Cond(Noeud_Or(C1, C2), True, E)`

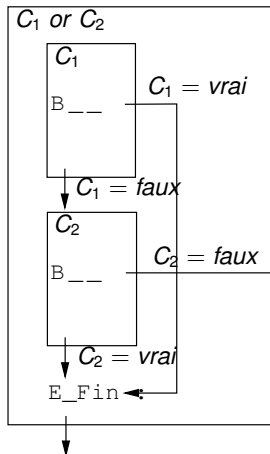


`Coder_Cond(Noeud_Or(C1, C2), True, E) =`  
`Coder_Cond(C1, True, E) ;`  
`Coder_Cond(C2, True, E) ;`



## Procédure Coder\_Cond

- `Coder_Cond(Noeud_Or(C1, C2), False, E)`



```

Coder_Cond(Noeud_Or(C1,C2),False,E) =
declare
 E_Fin : Etiqu := Nouvelle_Etiqu;
begin
 Coder_Cond(C1, True, E_Fin);
 Coder_Cond(C2, False, E);
 Générer_Etiqu(E_Fin);
end;

```

## Procédure Coder\_Cond

- `Coder_Cond(Noeud_Non(C), Saut, E) =`  
`Coder_Cond(C, not Saut, E) ;`

## Procédure Coder\_Cond

Valeur stockée en mémoire.

On suppose que faux est codé par #0.

- `Coder_Cond(Noeud_Ident("idf"), True, E) =`  
    Générer(LOAD, @idf, R0);  
    Générer(CMP, #0, R0);  
    Générer(BNE, E);
- `Coder_Cond(Noeud_Ident("idf"), False, E) =`  
    Générer(LOAD, @idf, R0);  
    Générer(CMP, #0, R0);  
    Générer(BEQ, E);

## Opérateurs de comparaison =, <, >, ≠, ≤, et ≥

Exemple : code pour l'expression  $E_1 < E_2$ .

```
Coder_Cond (Noeud_Inf (E1, E2), True, E) =
 ⟨évaluer E_1 dans R1⟩
 ⟨évaluer E_2 dans R2⟩
 CMP R2, R1
 BLT E
```

```
Coder_Cond (Noeud_Inf (E1, E2), False, E) =
 ⟨évaluer E_1 dans R1⟩
 ⟨évaluer E_2 dans R2⟩
 CMP R2, R1
 BGE E
```

⇒ traitement similaire aux opérateurs binaires des expressions arithmétiques

## Passe 3

### 2 Génération de code

- Génération de code pour un programme JCas simple
- Génération de code pour les expressions arithmétiques
- Génération de code pour les expressions booléennes
- **Codage des structures de contrôle**
- Génération de code pour les tableaux

## Codage des structures de contrôle

On suppose que les expressions booléennes sont codées par des flots de contrôle.

On définit une procédure `Coder_Inst` qui produit du code pour une instruction.

- Génère du code pour l'instruction correspondant
- à l'arbre A.

**procedure** `Coder_Inst` (A : Arbre) ;

## Conditionnelles

- `Coder_Inst (Noeud_Si (C, Alors, Sinon)) =`  
**declare**  
    `E_Sinon : Etiquette := Nouvelle_Etiquette;`  
    `E_Fin : Etiquette := Nouvelle_Etiquette;`  
**begin**  
    `Coder_Cond (C, False, E_Sinon);`  
    `Coder_Inst (Alors);`  
    `Générer (BRA, E_Fin);`  
    `Générer_Etiquette (E_Sinon);`  
    `Coder_Inst (Sinon);`  
    `Générer_Etiquette (E_Fin);`  
**end;**

## Conditionnelles

- `Coder_Inst (Noeud_Si (C, Alors, Noeud_Vide)) =`  
**declare**  
    `E_Fin : Etiqu := Nouvelle_Etiqu;`  
**begin**  
    `Coder_Cond (C, False, E_Fin) ;`  
    `Coder_Inst (Alors) ;`  
    `Générer_Etiqu (E_Fin) ;`  
**end ;**



## Boucles : Noeud\_Tantque (C, I)

On pourrait générer :

```
E_Début :
 ⟨Code de C avec branchement à E_Fin si C est faux⟩
 ⟨Code de I⟩
 BRA E_Début
E_Fin :
```

On choisit plutôt de générer le code :

```
BRA E_Cond
E_Début :
 ⟨Code de I⟩
E_Cond :
 ⟨Code de C avec branchement à E_Début si C est vrai⟩
```

On gagne un branchement par itération.

## Boucles : Noeud\_Tantque (C, I)

```

Coder_Inst (Noeud_Tantque (C, I)) =
 declare
 E_Cond : Etiqu := Nouvelle_Etiqu;
 E_Début : Etiqu := Nouvelle_Etiqu;
 begin
 Générer (BRA, E_Cond) ;
 Générer_Etiqu (E_Début) ;
 Coder_Inst (I) ;
 Générer_Etiqu (E_Cond) ;
 Coder_Cond (C, True, E_Début)
 end ;

```

## Passe 3

### 2 Génération de code

- Génération de code pour un programme JCas simple
- Génération de code pour les expressions arithmétiques
- Génération de code pour les expressions booléennes
- Codage des structures de contrôle
- Génération de code pour les tableaux

## Codage d'un tableau en mémoire

Soit le tableau A :

A : **array**[1..2] **of array** [1..3] **of** integer;

Ce tableau peut être représenté par la matrice

|         |         |         |
|---------|---------|---------|
| A[1][1] | A[1][2] | A[1][3] |
| A[2][1] | A[2][2] | A[2][3] |

## Codages possibles en mémoire

- Codage contigu par colonnes : on stocke en mémoire les différentes colonnes successivement.

|               |
|---------------|
| A [ 1 ] [ 1 ] |
| A [ 2 ] [ 1 ] |
| A [ 1 ] [ 2 ] |
| A [ 2 ] [ 2 ] |
| A [ 1 ] [ 3 ] |
| A [ 2 ] [ 3 ] |

Codage utilisé en Fortran.

## Codages possibles en mémoire

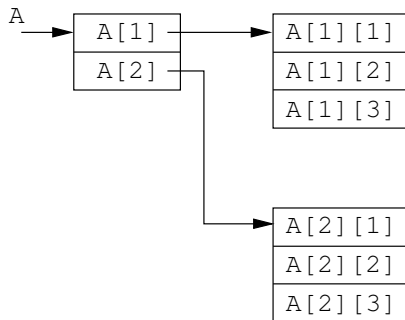
- Codage contigu par lignes :

|               |
|---------------|
| A [ 1 ] [ 1 ] |
| A [ 1 ] [ 2 ] |
| A [ 1 ] [ 3 ] |
| A [ 2 ] [ 1 ] |
| A [ 2 ] [ 2 ] |
| A [ 2 ] [ 3 ] |

Codage utilisé en Pascal et en Ada.

## Codages possibles en mémoire

- Codage non contigu : on stocke les vecteurs de façon contiguë. Un tableau à  $n$  dimensions est un vecteur de pointeurs sur un tableau à  $n - 1$  dimensions.



Codage utilisé en Java.

## Calcul de l'adresse d'un élément du tableau

Codage contigu par lignes.

Bornes connues à la compilation.

### Exemple 1. Tableau à une dimension

```
v : array[1..3] of integer ;
v[i] := k ;
```

Adresse de  $v[1]$  :  $dv$  (GB).

Entiers codés sur un mot.



## Tableau à une dimension

**; Calcul de l'indice**

LOAD @i, R0

**; Vérifie que l'indice est dans les bornes du tableau**

CMP #1, R0

BLT erreur\_indice\_tableau

CMP #3, R0

BGT erreur\_indice\_tableau

**; Calcul de l'adresse de  $v(i)$**

SUB #1, R0

**; L'adresse de  $v(i)$  est  $dv(GB, R0)$**

**; Calcul de l'expression**

LOAD @k, R1

**; Affectation**

STORE R1,  $dv(GB, R0)$

## Exemple 2. Tableau à deux dimensions

```
A : array[1..2] of array [1..3] of integer ;
k := A[i][j] ;
```

; Calcul du premier indice

```
LOAD @i, R0
```

; Vérifie que l'indice est dans les bornes du tableau

```
CMP #1, R0
```

```
BLT erreur_indice_tableau
```

```
CMP #2, R0
```

```
BGT erreur_indice_tableau
```

## Tableau à deux dimensions

; Calcul de l'adresse de  $A[i]$

SUB #1, R0

MUL #3, R0 ; multiplie par la taille des éléments

;  $A[i]$  est de taille 3

LEA  $dA(GB, R0)$ , R0 ; R0 contient l'adresse de  $A[i]$

; Calcul du deuxième indice

LOAD @j, R1

; Vérifie que l'indice est dans les bornes du tableau

CMP #1, R1

BLT erreur\_indice\_tableau

CMP #3, R1

BGT erreur\_indice\_tableau

## Tableau à deux dimensions

**; Calcul de l'adresse de  $A[i][j]$**

SUB #1, R1

**; L'adresse de  $A[i][j]$  est 0 (R0, R1)**

**; Calcul de l'expression  $A[i][j]$**

LOAD 0(R0, R1), R0 ; R0 **contient la valeur**  $A[i][j]$

**; Affectation**

STORE R0, @k

## Calcul de l'adresse d'un élément

Soit  $T$  un tableau à  $n$  dimensions

$T : \mathbf{array} [d_1 .. f_1, d_2 .. f_2, \dots, d_n .. f_n] \text{ of integer} ;$

Soit  $@T$  est l'adresse de  $T[d_1][d_2]...[d_n]$ .

L'adresse de  $T[i_1][i_2]...[i_n]$  est :

$$@T + (i_1 - d_1) \times e_1 + (i_2 - d_2) \times e_2 + \dots + (i_n - d_n) \times e_n$$

où  $e_k$  est défini par :

$$e_n = 1 \quad (\text{taille d'un entier})$$

$$e_{n-1} = (f_n - d_n + 1) \times e_n$$

...

$$e_2 = (f_3 - d_3 + 1) \times e_3$$

$$e_1 = (f_2 - d_2 + 1) \times e_2$$

## Calcul de l'adresse d'un élément

**Remarque** :  $e_k$  est la taille des éléments du tableau  $T[i_1][i_2] \cdots [i_k]$ .

L'adresse de  $T[i_1][i_2] \dots [i_n]$  est :

$$\begin{aligned} & @T + (i_1 - d_1) \times e_1 + (i_2 - d_2) \times e_2 + \cdots + (i_n - d_n) \times e_n \\ &= @T + \sum_{k=1}^n (i_k - d_k) \times e_k \\ &= (@T - \sum_{k=1}^n d_k \times e_k) + \sum_{k=1}^n i_k \times e_k \end{aligned}$$

Le terme  $(@T - \sum_{k=1}^n d_k \times e_k)$  est connu à la compilation :

c'est l'*adresse virtuelle* de  $T$ .

## Calcul de l'adresse d'un élément

Le terme  $\sum_{k=1}^n i_k \times e_k$  doit être calculé à l'exécution.

### Exemple 1. Tableau à une dimension

Soit  $dv_V$  (GB) l'adresse virtuelle du tableau  $v$ .

; Calcul de l'indice

LOAD @i, R0

; Vérification que l'indice est dans les bornes du tableau

; [...]

; Calcul de l'adresse de  $v[i]$  : c'est  $dv_V$  (GB, R0)

LOAD @k, R1

; Affectation

STORE R1,  $dv_V$  (GB, R0)

## Exemple 2 : tableau à deux dimensions

Soit  $dAv(GB)$  l'adresse virtuelle du tableau A.

; Calcul du premier indice

LOAD @i, R0

; Vérifie que l'indice est dans les bornes du tableau

; [...]

; Calcul de l'adresse (virtuelle) de  $A[i]$

MUL #3, R0 ; multiplie par la taille des éléments

LEA  $dAv(GB, R0)$ , R0

; R0 contient l'adresse virtuelle de  $A[i]$

; Calcul du deuxième indice

LOAD @j, R1

; Vérifie que l'indice est dans les bornes du tableau

; [...]



## Exemple 2 : tableau à deux dimensions

**; Calcul de l'adresse de  $A[i][j]$  : c'est 0 (R0, R1)**

**; Calcul de l'expression  $A[i][j]$**

LOAD 0(R0, R1), R0

**; R0 contient la valeur  $A[i][j]$**

**; Affectation**

STORE R0, @k

## Génération de code pour les places

- Génère du code pour la place correspondant
- à l'arbre A.
- Le résultat est l'opérande de la place à la fin de ce code.

```
function Coder_Place
 (A : Arbre) return Opérande ;
```

On doit ensuite définir cette fonction pour toutes les places du langage, en particulier :

- `Coder_Place (Noeud_Ident ("idf")) ;`
- `Coder_Place (Noeud_Index (Place, Exp)) .`

`Noeud_Index` : cf. expressions arithmétiques

## Erreurs à l'exécution

Le code généré peut provoquer une erreur à l'exécution.

Le programme assembleur doit afficher un message d'erreur et s'arrêter.

Erreurs possibles :

- Débordement d'intervalle

```
i : 1 .. 10;
```

```
i := 0;
```

- Débordement arithmétique (dont la division par 0)
- Débordement d'indice de tableau
- Débordement de la pile (à tester avec TSTO)

## Passe 3

1 Machine abstraite

2 Génération de code

3 Mise en oeuvre

## Passe 3

- 3 Mise en oeuvre
  - Classes Java fournies
  - Travail à effectuer

## Classes Java fournies

- **Classe** `JCas` du **paquetage** `ProjetCompil.Gencode.Src` : programme principal du compilateur `JCas`
- Dans le répertoire `ProjetCompil/Gencode/Test` : script  
`jcasc`  
`jcasc fich.cas` compile le programme `fich.cas` et produit le fichier assembleur `fich.ass`
- **Paquetage** `ProjetCompil.Global.Src3` : permet de manipuler le programme assembleur généré  
**Classes** `Prog`, `Ligne`, `Inst`, `Etq`, `Operation`, `Operande`

## Passe 3

- 3 Mise en oeuvre
  - Classes Java fournies
  - Travail à effectuer

## Travail à effectuer

- Coder la passe 3 (dans `ProjetCompil/Gencode/Src`)  
Parcours de l'arbre abstrait
- Écrire des tests (dans `ProjetCompil/Gencode/Test`)  
**Exécuter les programmes générés avec ima**
- Rédiger une documentation (dans `Gencode/Doc`) décrivant :
  - les messages d'erreur (limitations du compilateur),
  - l'architecture de la passe 3,
  - les différents algorithmes utilisés en passe 3.



## Conseil

Pour tester efficacement le code écrit

- Générer du code pour la machine abstraite
- Exécuter ce code avec ima
- **Commencer par coder les instructions write et new\_line**