Université Gustave Eiffel

INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE



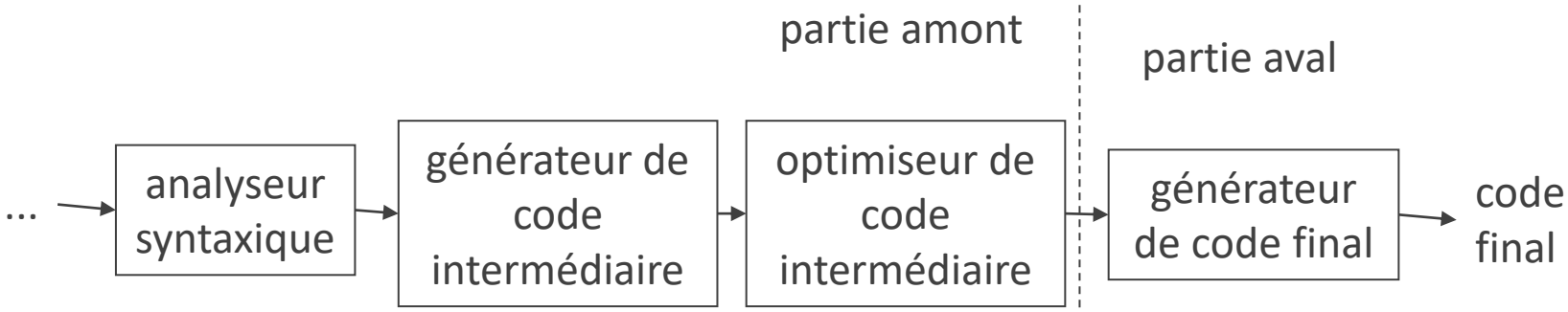
# Code intermédiaire

Université Gustave Eiffel

INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

# Sommaire

# Génération de code intermédiaire




**Code intermédiaire**

Exemple : bytecode Java (aussi exécutable sur une machine virtuelle Java)

Indépendant de l'architecture de la machine cible

Utilise des noms symboliques (*mnemonics*, ex. `add` ou `+`), comme le code final

**Séparer code intermédiaire et final permet de**

- réutiliser un générateur de code intermédiaire sur des machines différentes
- réutiliser un optimiseur de code intermédiaire

# Exemples de différences entre architectures cibles

```
; .nasm
mov    rax, 4
sub    [b], rax
```

```
; .nasm
push qword [b]
push qword 4
pop rax
sub [rsp], rax  ; b-4
pop qword [b]
```

```
; .nasm
mov    rax, [b]
sub    rax, 4
mov qword [b], rax
```

**Nombre de registres**

Environ 16 à 256

**Instructions**

CISC : certains opérandes peuvent être en mémoire

RISC : tous les opérandes sont dans des registres sauf pour les instructions d'accès mémoire

**Utilisation de la pile ou de registres**

Pour évaluer une expression, sauvegarder les opérandes

- soit dans la pile
- soit dans des registres tant qu'il y en a

# Code intermédiaire

```
a=(b-4)*c
```

```
tmp1 := &a
tmp2 := b
tmp3 := 4
tmp4 := tmp2 - tmp3
tmp5 := c
tmp6 := tmp4 * tmp5
base[tmp1] := tmp6
```

```
iload b
ldc &4
isub
iload c
imul
istore a
```

**Neutre**

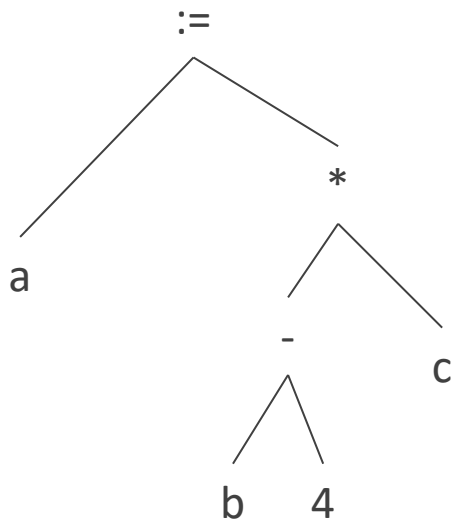
Pas de différence entre registres et accès mémoire

**Avec ou sans pile**

Pour évaluer une expression, sauvegarder les opérandes

- soit à des adresses mémoire
- soit dans la pile

# Code intermédiaire sous forme d'arbre abstrait




`a=(b-4)*c`

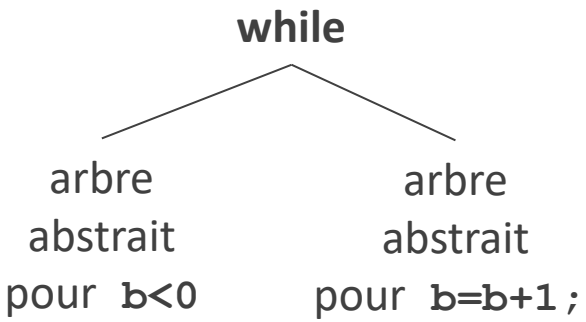
Chaque nœud correspond à une instruction

Pas de pile

Exemple : le langage GENERIC utilisé par gcc

Ici, l'ordre dans lequel il faut exécuter les instructions est donné par le parcours de l'arbre : l'ordre suffixe

# Code intermédiaire sous forme d'arbre abstrait

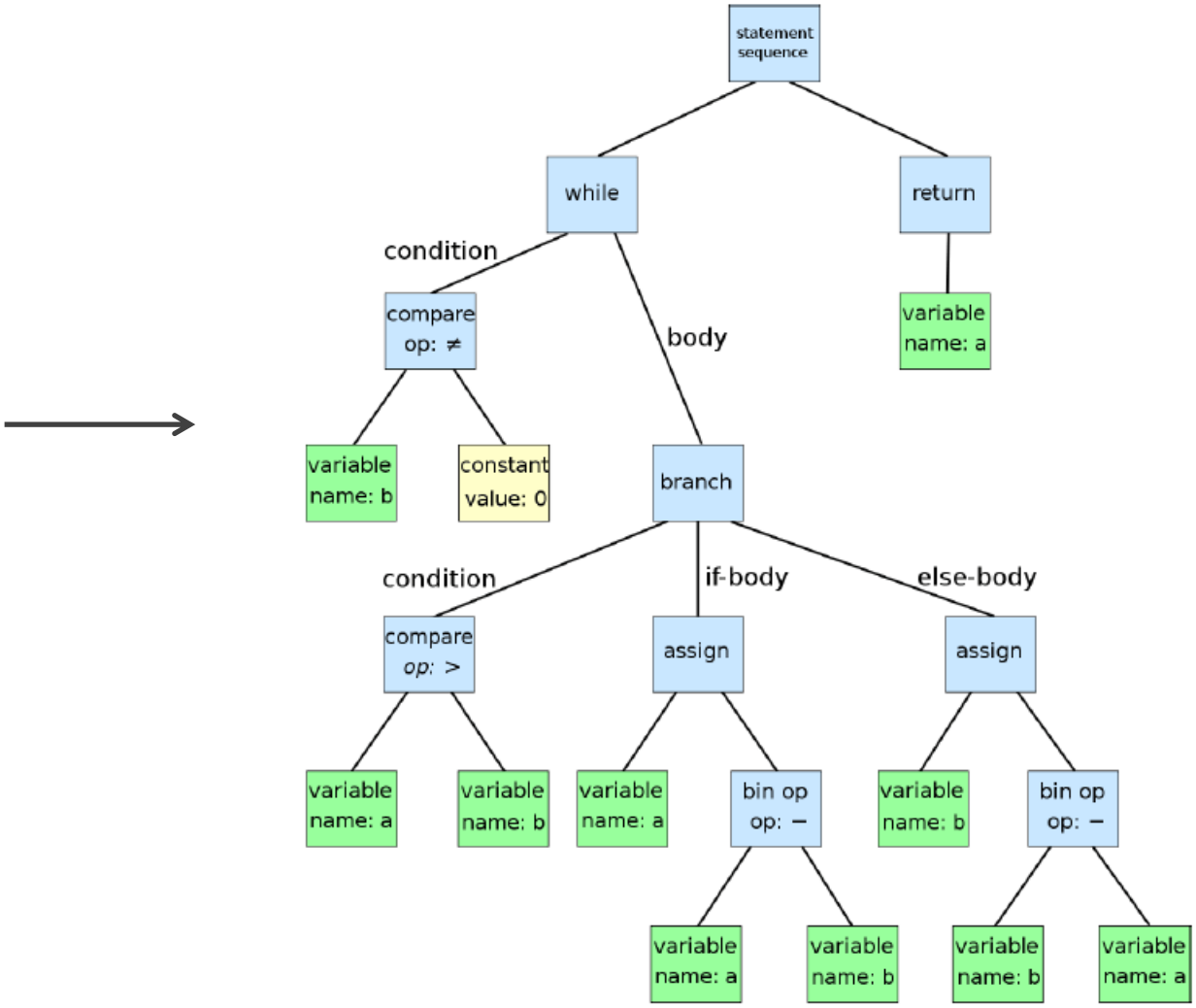



```
while(b<0) b=b+1;
```

L'arbre abstrait ne contient pas explicitement de sauts

# Un arbre abstrait satisfaisant

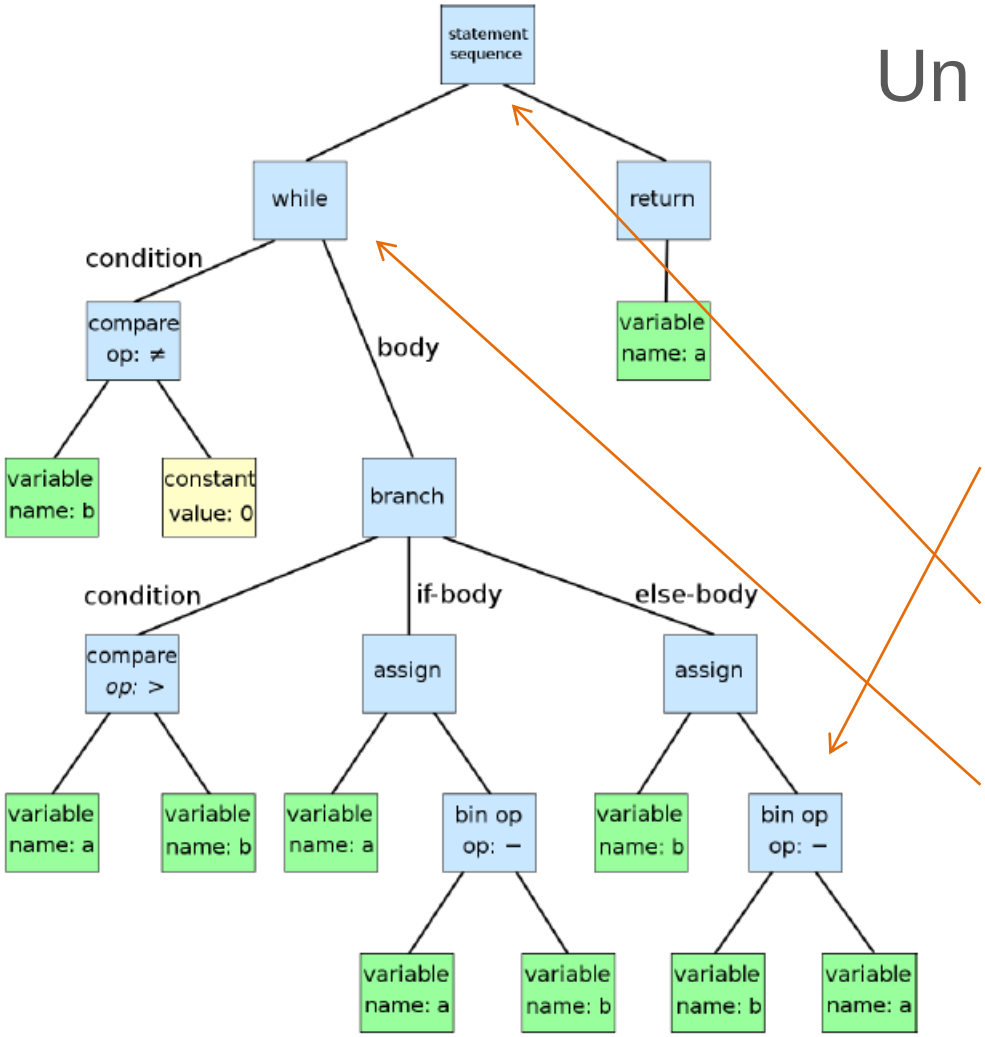
```
while b != 0
  if a > b
    a := a - b
  else
    b := b - a
return a
```



Université Gustave Eiffel

INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

# Un arbre abstrait satisfaisant




**Opérateur**

Un nœud interne, les opérandes comme fils

**Liste**

Un seul nœud interne, les éléments comme fils

**Structure de contrôle**

Un nœud interne, chaque partie comme fils

Université Gustave Eiffel

INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

# Code à trois adresses

```
tmp1 := &a
tmp2 := b
tmp3 := 4
tmp4 := tmp2 - tmp3
tmp5 := c
tmp6 := tmp4 * tmp5
base[tmp1] := tmp6
```

Forme générale : **x := y op z**

où **op** est un opérateur. Les trois adresses sont celles de **x**, **y** et **z**.

Pas de pile

**Instructions d'affectation**

**x := y op z** **x := op y**

**Instructions de saut**

inconditionnel **goto L**

conditionnel **if x relop y goto L**

**Instructions indicées**

**x := y [ i ]** **x [ i ] := y**

# Code intermédiaire sous forme de bytecode

```
; nasm
push   qword [b]
push   qword 4
pop    rcx
pop    rbx
sub    rbx, rcx
push   rbx
push   qword [c]
pop    rcx
pop    rbx
imul   rbx, rcx
push   rbx
pop    qword [a]
```

```
iload b
ldc &4
isub
iload c
imul
istore a
```

`a=(b-4)*c`

accès mémoire, **empiler**

accès zone de constantes, **empiler**

2 **dépiler**, soustraction, 1 **empiler**

**empiler**

2 **dépiler**, multiplication, 1 **empiler**

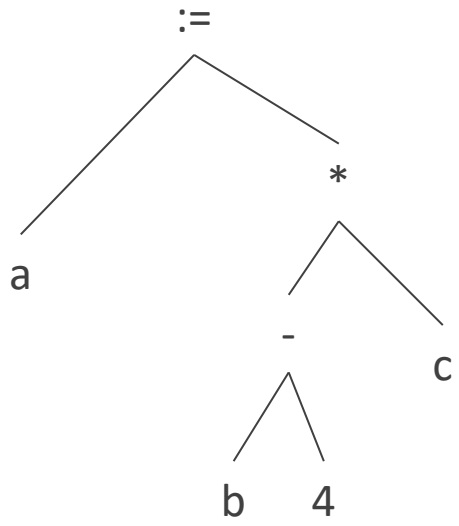
**dépiler**, accès mémoire

Université Gustave Eiffel

INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

# Sommaire

# Traduire un arbre abstrait en code à trois adresses



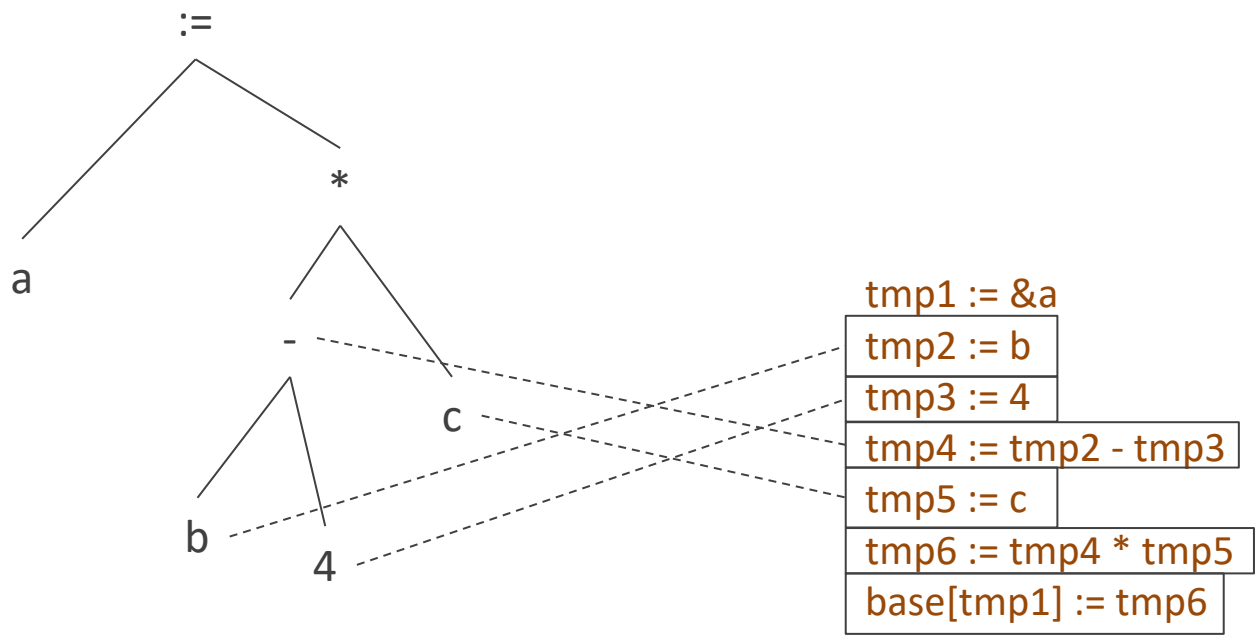

```
tmp1 := &a
tmp2 := b
tmp3 := 4
tmp4 := tmp2 - tmp3
tmp5 := c
tmp6 := tmp4 * tmp5
 base[tmp1] := tmp6
```

**`a=(b-4)*c`**

Parcours de l'arbre dans l'ordre suffixe :

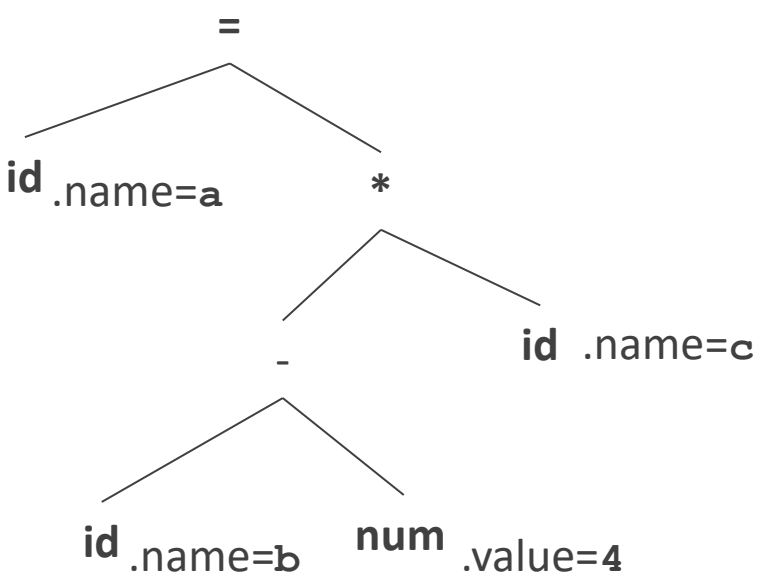
| | | | |
|---|---|---|---|
| 1. | a | | |
| 2. | b | | |
| 3. | 4 | | |
| 4. | - | 2 | 3 |
| 5. | c | | |
| 6. | * | 4 | 5 |
| 7. | := | 1 | 6 |

# Traduire un arbre abstrait en code à trois adresses




Avec cette méthode, la traduction d'une opération est indépendante de la façon dont on calcule les opérandes

# Traduire un arbre abstrait en code à trois adresses




```
tmp1 := &a
tmp2 := b
tmp3 := 4
tmp4 := tmp2 - tmp3
tmp5 := c
tmp6 := tmp4 * tmp5
base[tmp1] := tmp6
```

**Principe**

Le compilateur parcourt l'arbre abstrait

À chaque nœud de l'arbre abstrait il écrit du code à trois adresses dans un fichier

L'ordre des instructions dans le fichier cible correspond à l'ordre dans lequel on les écrit

Université Gustave Eiffel

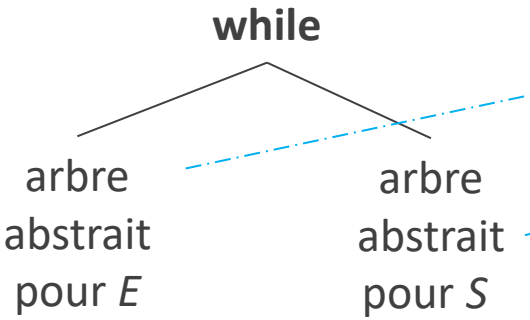
INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

# Traduire un arbre abstrait en code à trois adresses

| | |
|---|---|
| Nœud = | p := lookup(firstchild.name) ; visit(=.secondchild);<br>if (p) write('base[' p.place ']:=' =.secondchild.place)<br>else error() ; |
| Nœud - | visit(-.firstchild); visit(-.secondchild);<br>-.place := newtemp() ;<br>write(-.place ':=' -.firstchild.place '-' -.secondchild.place) ; |
| Nœud * | visit(*.firstchild); visit(*.secondchild);<br>*.place := newtemp() ;<br>write(*.place ':=' *.firstchild.place '*' *.secondchild.place) ; |
| Nœud **id** | **id**.place := newtemp() ;<br>p := lookup(**id**.name) ;<br>if (p) write(**id**.place ':=base[' p.place ']') else error() ; |
| Nœud **num** | **num**.place := newtemp();<br>write(**num**.place ':=' **num**.value) ; |

# Traduction des structures de contrôle

*S* --> **while ( *E* ) *S***




**Exemple**

Code source : `while(b<0) b=b+1;`

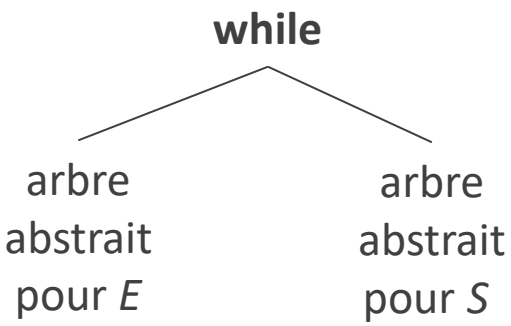
Code cible :

```
label1:
tmp1:= b<0
if tmp1=0 goto label2
tmp2:= b+1
b:= tmp2
goto label1
label2:
```

# Traduction des structures de contrôle

```
label1:
tmp1:= b<0
if tmp1=0 goto label2
tmp2:= b+1
b:= tmp2
goto label1
label2:
```

Nœud **while**




```
begin := newlabel() ;
after := newlabel() ;
write(begin ':');
visit(while.firstchild);
write('if' while.firstchild.place '= 0 goto' after);
visit(while.secondchild);
write('goto' begin);
write(after ':') ;
```

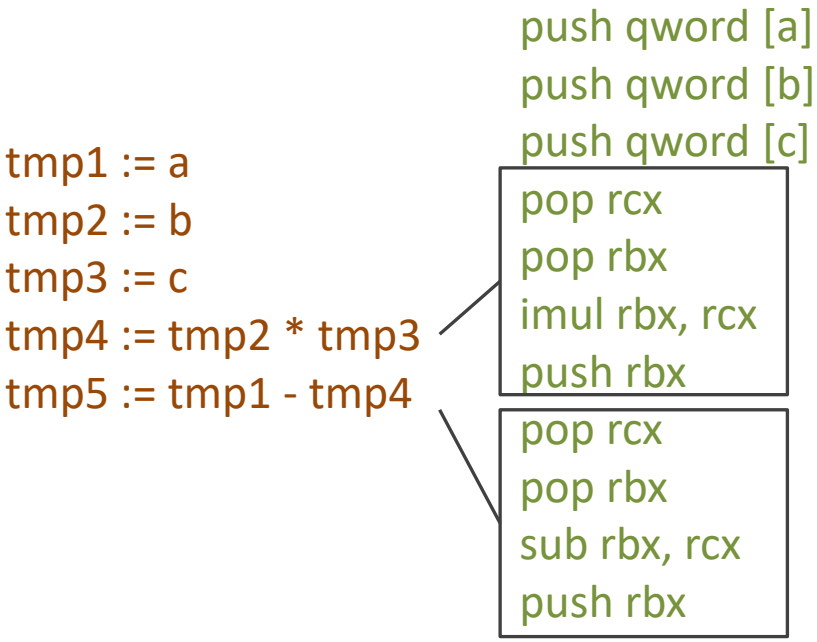
Université Gustave Eiffel

INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

# Sommaire

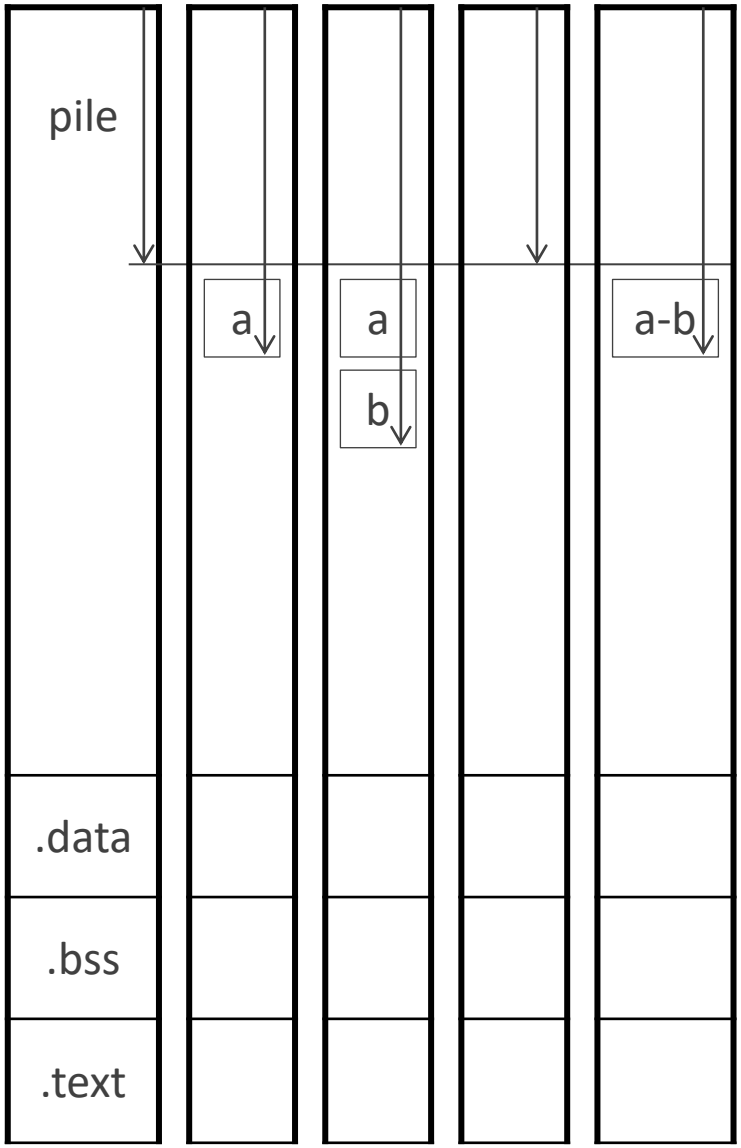
# Traduire du code à trois adresses en code final

```
a-(b*c)
```




Le code final utilise souvent la pile pour sauvegarder les opérandes

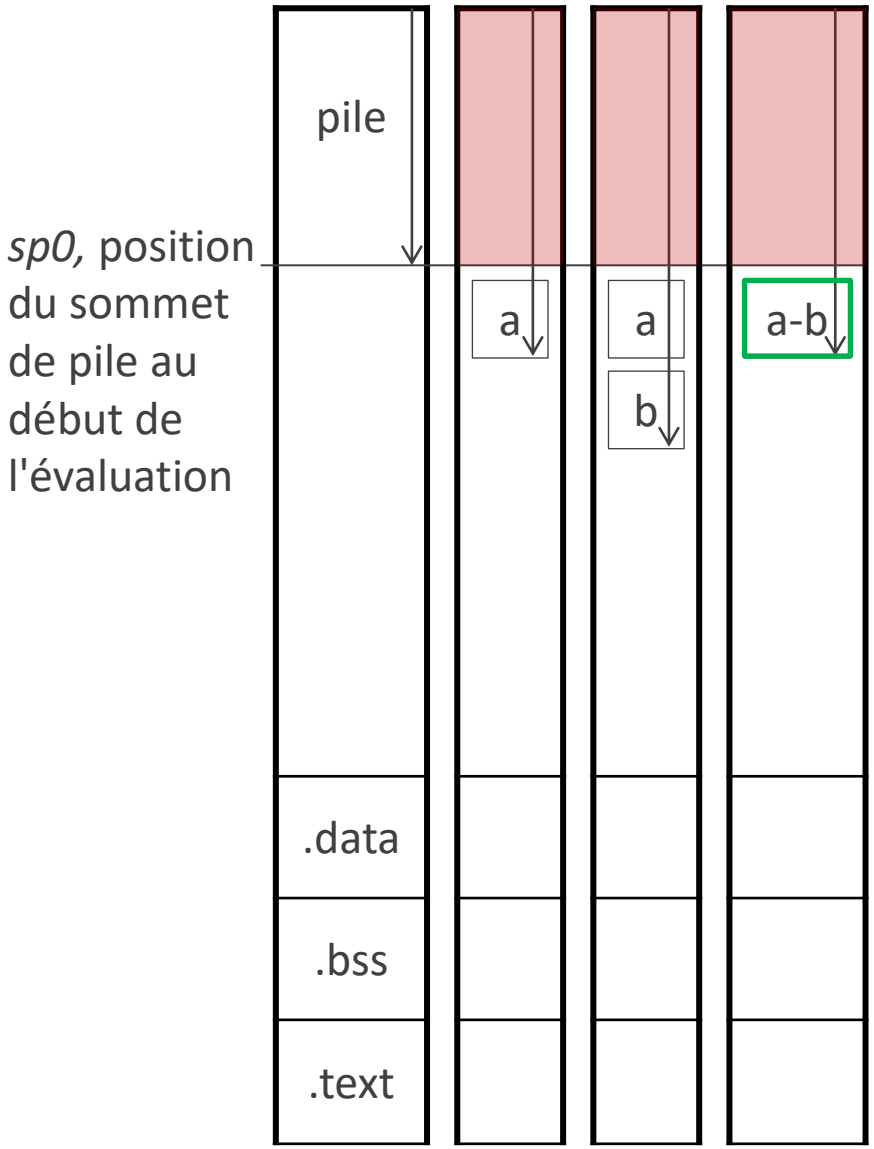
# Traduire des expressions en code qui évalue l'expression en utilisant la pile




**Étapes**

Évaluer tous les opérandes
Dépiler tous les opérandes
Faire l'opération
Empiler le résultat

# Traduire des expressions en code qui évalue l'expression en utilisant la pile




**Principe**

Le code qui évalue une expression ne dépile pas les informations qui étaient déjà dans la pile au moment où l'évaluation commence

À la fin, la valeur de l'expression est dans la pile juste au-dessus du niveau *sp0*

**Étapes**

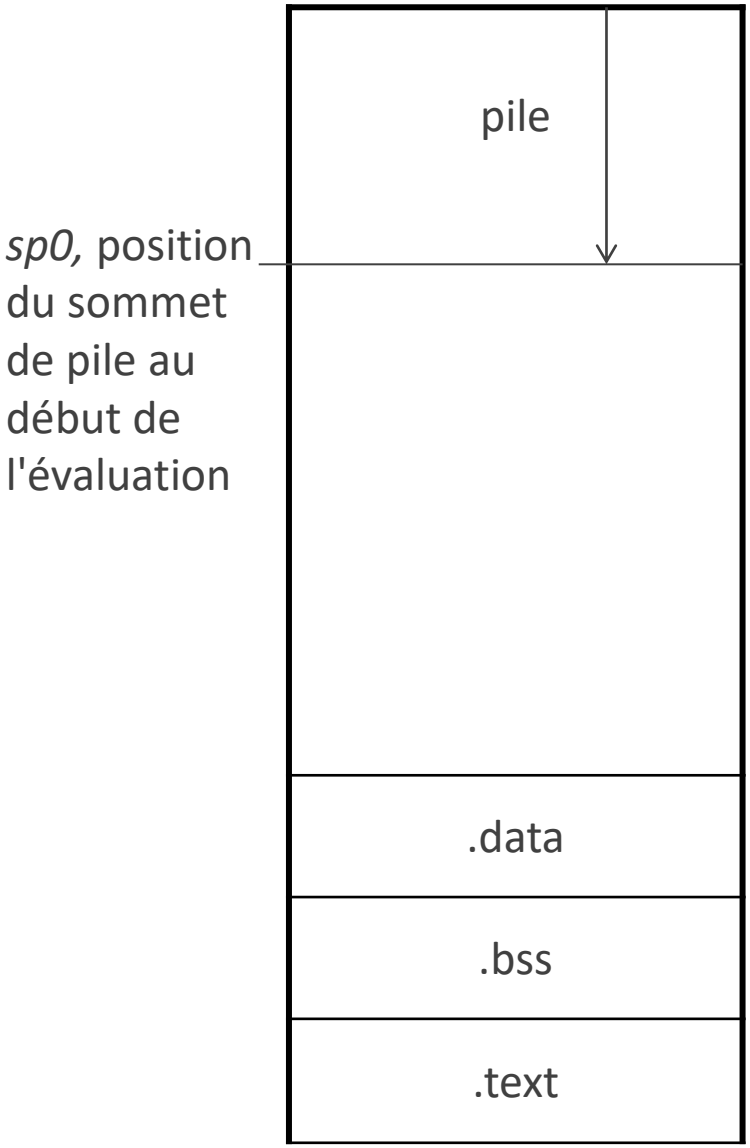
Évaluer tous les opérandes

Dépiler tous les opérandes

Faire l'opération

Empiler le résultat

# Traduire des expressions en code qui évalue l'expression en utilisant la pile




**Traduire du C en code nasm 64 bits**

1. a=b;
2. a=b*c*(b+c);
3. a=(b/c)*(b-c);
4. a=(b=c);
5. a=b+2;

Université Gustave Eiffel

INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

# Sommaire

# Instructions de code à trois adresses pour les tableaux

```
; nasm
mov rax, [y+z]
mov [x], rax

; nasm
mov rax, [z]
mov [x+y], rax
```

**x := y [ z ]**

**y** est l'adresse générale du tableau

**z** est l'adresse relative (*offset*) en octets d'un élément dans le tableau

**x [ y ] := z**

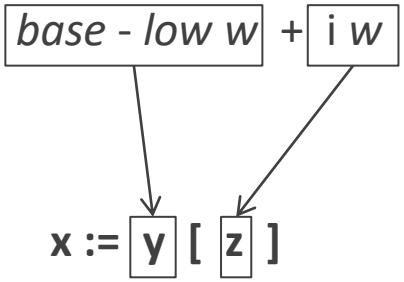
**x** est l'adresse générale du tableau

**y** est l'adresse relative d'un élément dans le tableau

**Différences avec un langage de haut niveau**

- a[i+1] n'est pas forcément 1 octet après a[i]
  Taille d'une case : *w*
- Le premier élément n'est pas forcément a[0]
  Borne inférieure des indices : *low*

# Adresse d'un élément de tableau




**Adresse de a[i]**

$$base + (i - low)\ w$$

*base* est l'adresse du premier élément du tableau

En langage C, *low* = 0, donc l'expression se réduit à

$$base + i\ w$$

Un programme peut être compilé une fois et exécuté des milliers de fois

Faire la plus grande partie possible du calcul à la compilation

**Quelle partie peut être calculée à la compilation ?**

# Grammaire attribuée pour les tableaux

*S* --> *Lvalue* **=** *E*

*E* --> *E* **+** *E*

*E* --> **(** *E* **)**

*E* --> *Lvalue*

*Lvalue* --> **id**

*Lvalue* --> **id [** *E* **]**

**Exemple**

Code source `a[i]=b[j];`

Code cible

```
tmp1:= a.const
tmp2:= a.width*i
tmp3:= b.const
tmp4:= b.width*j
tmp5:= tmp3[tmp4]
tmp1[tmp2]:= tmp5
```

**Entrée dans la table des symboles**

.type

.const : l'adresse du tableau calculable à la compilation

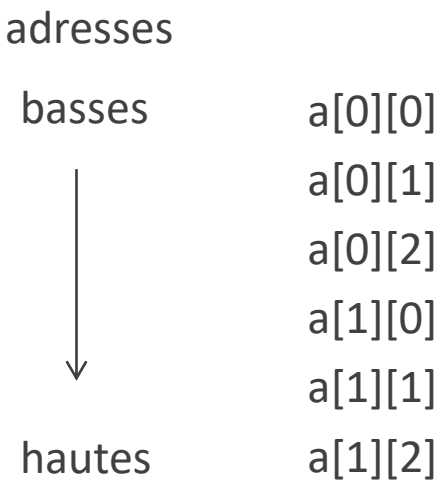
.width : *w*

# Grammaire attribuée pour les tableaux

| | | |
|---|---|---|
| *S* | --> *Lvalue* **=** *E* | if (*Lvalue*.offset)<br>write(*Lvalue*.place, '[', *Lvalue*.offset, ']', ':=', *E*.place) ;<br>else write(*Lvalue*.place, ':=', *E*.place) ; |
| *E* | --> *E* **+** *E* | ... |
| *E* | --> **(** *E* **)** | ... |
| *E* | --> *Lvalue* | if (*Lvalue*.offset) {<br>*E*.place := newtemp() ;<br>write(*E*.place, ':=', *Lvalue*.place, '[', *Lvalue*.offset, ']') ; }<br>else *E*.place = *Lvalue*.place ; |
| *Lvalue* | --> **id** | *Lvalue*.place := **id**.entry.place ; *Lvalue*.offset := 0 ; |
| *Lvalue* | --> **id [** *E* **]** | *Lvalue*.place := newtemp() ; *Lvalue*.offset := newtemp() ;<br>write(*Lvalue*.place, ':=', **id**.entry.const) ;<br>write(*Lvalue*.offset, ':=', **id**.entry.width, '*', *E*.place) ; |

# Tableaux à plusieurs dimensions

| a[i][j] | j = 0 | j = 1 | j = 2 |
|---|---|---|---|
| i = 0 | *base* | *base*+*w* | *base*+2*w* |
| i = 1 | *base*+3*w* | *base*+4*w* | *base*+5*w* |




Dans la plupart des langages, dont C, les éléments sont rangés ligne par ligne

Quand on passe de chaque case du tableau à sa voisine dans la mémoire, c'est le dernier indice qui varie le plus vite

En Fortran : colonne par colonne

# Tableaux à 2 dimensions

| $a[i_1][i_2]$ | $i_2 = 1$ | ... | $i_2 = n_2$ |
|---|---|---|---|
| $i_1 = 1$ | *base* | ... | $base+(n_2-1)w$ |
| ... | ... | | ... |
| $i_1 = n_1$ | $base+(n_1-1)n_2w$ | ... | $base+(n_1n_2-1)w$ |

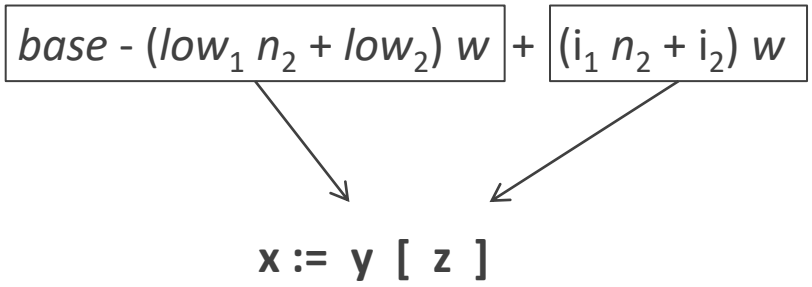
**Adresse de $a[i_1][i_2]$**

Chaque ligne contient $n_2$ éléments, où $n_2$ est le nombre des valeurs possibles de $i_2$

Les lignes avant la $i_1$ contiennent au total $(i_1 - low_1)\ n_2$ éléments

Donc $base + ((i_1 - low_1)\ n_2 + i_2 - low_2)\ w$

**Quelle partie peut être calculée à la compilation ?**



Université Gustave Eiffel

INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

# Tableaux à 2 dimensions

**Entrée dans la table des symboles**

.type

.const : l'adresse générale du tableau

.width

.extent(2) : nombre de valeurs possibles du 2$^{e}$ indice

On n'a pas besoin du nombre de valeurs possibles du 1$^{er}$ indice

# Tableaux à *k* dimensions

Adresse de a[$i_1$][$i_2$] : *base* + (($i_1$ - *low*$_1$) $n_2$ + $i_2$ - *low*$_2$) *w*

On généralise à *k* dimensions

*base* + ((...(($i_1$ - *low*$_1$) $n_2$ + $i_2$ - *low*$_2$) $n_3$ + $i_3$ - *low*$_3$)...) $n_k$ + $i_k$ - *low*$_k$) *w*

Une partie peut être calculée à la compilation

*base* - ((...((*low*$_1$ $n_2$ + *low*$_2$) $n_3$ + *low*$_3$)...) $n_k$ + *low*$_k$) *w* + ((...(($i_1$ $n_2$ + $i_2$) $n_3$ + $i_3$)...) $n_k$ + $i_k$) *w*

Le reste doit être calculé à l'exécution (adresse relative)

Le compilateur produit du code qui le calcule :

$e_1 = i_1$

$e_d = e_{d-1}\, n_d + i_d$ pour $1 < d \leq k$

$offset = e_k\, w$

Université Gustave Eiffel

INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

# Tableaux à *k* dimensions

```
int process_space(int field[][WIDTH][DEPTH]);
```

**Entrée dans la table des symboles**

.type
.const
.width
.extent() : nombre de valeurs possibles de chaque indice sauf le 1er

# Grammaire attribuée : tableaux à *k* dimensions

| | |
|---|---|
| *S* | --> *L* **=** *E* |
| *E* | --> *E* **+** *E* |
| *E* | --> **(** *E* **)** |
| *E* | --> *L* |
| *L* | --> **id** |
| *L* | --> **id [** *Elist* **]** |
| *Elist* | --> *Elist* **] [** *E* |
| *Elist* | --> *E* |

Une grammaire simple, mais l'attribut donnant les $n_d$ sera hérité : il est obtenu à partir du nom du tableau et transmis de haut en bas par les nœuds *Elist*

Une version où l'attribut est synthétisé :

| | |
|---|---|
| *L* | --> *Elist* **]** \| **id** |
| *Elist* | --> *Elist* **] [** *E* \| **id [** *E* |

L'attribut donnant les $n_d$ obtenu à partir du nom du tableau peut être transmis de bas en haut par les nœuds *Elist*

# Grammaire attribuée : tableaux à *k* dimensions

| | |
|---|---|
| *S* | --> *L* = *E* |
| *E* | --> *E* + *E* |
| *E* | --> ( *E* ) |
| *E* | --> *L* |
| *L* | --> *Elist* ] |
| *L* | --> **id** |
| *Elist* | --> *Elist* ] [ *E* |
| *Elist* | --> **id** [ *E* |

**Exemple**

Code source

```
a[i]=b[j][k];
```

Code cible

```
tmp1:= a.const
tmp2:= a.width*i
...
```

Université Gustave Eiffel

INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

# Grammaire attribuée : tableaux à *k* dimensions

| | | |
|---|---|---|
| *S* | --> *L* = *E* | if (*L*.offset)<br>  write(*L*.place, '[', *L*.offset, ']', ':=', *E*.place) ;<br>else write(*L*.place, ':=', *E*.place) ; |
| *E* | --> *E* + *E* | ... |
| *E* | --> ( *E* ) | ... |
| *E* | --> *L* | if (*L*.offset) {<br>  *E*.place := newtemp() ;<br>  write(*E*.place, ':=', *L*.place, '[', *L*.offset, ']', ) ; }<br>else *E*.place = *L*.place ; |

Université Gustave Eiffel — INSTITUT D'ÉLECTRONIQUE ET D'INFORMATIQUE GASPARD-MONGE

$e_1 = i_1$

$e_d = e_{d-1}\, n_d + i_d$ pour $1 < d \leq k$

$offset = e_k\, w$

# Grammaire attribuée : tableaux à *k* dimensions

| | | |
|---|---|---|
| *L* | --> *Elist* **]** | *L*.place := newtemp() ; *L*.offset := newtemp() ;<br>write(*L*.place, ':=', *Elist*.entry.const) ;<br>write(*L*.offset, ':=', *Elist*.entry.width, '*', *Elist*.place) ; |
| *L* | --> **id** | *L*.place := **id**.entry.place ; *L*.offset := 0 ; |
| *Elist* | --> *Elist* **] [** *E* | t := newtemp() ; d := $Elist_1$.count + 1 ;<br>write(t, ':=', $Elist_1$.place, '*', $Elist_1$.entry.extent(d)) ;<br>write(t, ':=', t, '+', *E*.place) ; *Elist*.count := d ;<br>*Elist*.entry := $Elist_1$.entry ; *Elist*.place := t ; |
| *Elist* | --> **id [** *E* | *Elist*.count := 1 ; *Elist*.entry := **id**.entry ;<br>*Elist*.place := *E*.place ; |