ECOLE SUPERIEURE D'INGENIEUR DE RENNES

Compilation

code 3 adresses

Olivier Ridoux

UNIVERSITÉ DE RENNES 1

Plan

- Principes du code 3 adresses
- Génération de code
- Représentation du calcul
- Représentation des données

Principes du code 3 adresses

La sortie du *front-end*

- Le programme lu appartient bien au langage
  - lexicalement
  - syntaxiquement
  - « sémantiquement »

...il est représenté par un ASA + ou – décoré

  - table des symboles
  - types

La production de code

- Pensée globale – action locale ⟶ **code naïf**

...analyse globale ⟶ amélioration de ce code

...très technique

  – nombre et type des registres

  – modes d'adressages

- Mais beaucoup d'améliorations ne sont pas liées à une cible

  – factoriser des traitements pour plusieurs cibles

La production de code en deux étapes

- Étape 1 : produire du **code intermédiaire** pour une **cible large**
  - viser une famille de machines cibles

- Étape 2 : **traduire** le code intermédiaire en un code exécutable pour une **cible précise**
  - tous les codes intermédiaires ne conviennent pas à toutes les cibles

Étape 1

- Production d'un code **multi-plateforme** et **sans contrainte** de ressources
  - registres, sans limite
  - types de données, sans restriction
  - pensée globale – action locale ⟶ code naïf

  ...analyse globale ⟶ amélioration de ce code

Étape 2

- Traduction du code intermédiaire en un code cible **mono-plateforme** et à **ressources contraintes**
  - registres, en nombre limité
  - types de données, ceux de la cible
  - modes d'adressage, spécifiques
  - pensée globale – action locale ⟶ code naïf

  ...analyse globale ⟶ amélioration de ce code

Codes intermédiaires (1)

A + B × C

- Machine à pile

  push A ; push B ; push C ; mult ; add

- JVM, LUA 4.x

Codes intermédiaires (2)

$$A + B \times C$$

- Machine à registres

  $$R_1 = B \times C \; ; \; R_2 = A + R_1$$

  machine 3 adresses

- MIPS, LUA 5.x
- Intel ≈ machine 1,5 adresses

  1 adresse + 1 registre

Codes intermédiaires (3)

- Langage de programmation

  C, JavaScript

Aparté - Représentation textuelle du code intermédiaire

- Tentant de représenter le code intermédiaire par son texte, car plus facile à lire

...mais, représentation textuelle implique analyse syntaxique pour la lire

- La représentation textuelle n'a qu'un but documentaire

...donc la bonne représentation est comme une **structure de donnée interne**

Code 3 adresses (1)

- X = Y op Z, X = op Z, X = Y op, X = op
  - X, Y ou Z sont des adresses de mémoires ou de registres
  - op est une opération ou une constante
  - **au plus 3 adresses, au plus 1 opérateur ou constante**

X = X + 1 ∉ code 3@

Code 3 adresses (2)

- **X = Y op Z**
- **Lire** la mémoire en Y et Z
- **Écrire** la mémoire en X

Code 3 adresses (3)

- Représentation textuelle

  « X = Y op Z »

  lisibilité par un **humain**

- Représentation interne

  un quadruplet

  < op, X, Y, Z >, < op, X, Y, _ >

  < op, X, _, _ > ou < op, _, _, _ >

  lisibilité par une **machine**

Code 3 adresses (4)

- La liste des op est ouverte

  X = Y op Z, noté < **op**, X, Y, Z >

  X = k, noté < **const k**, X, _, _ >

  …

  X = & Y, noté < **&**, X, Y, _ >

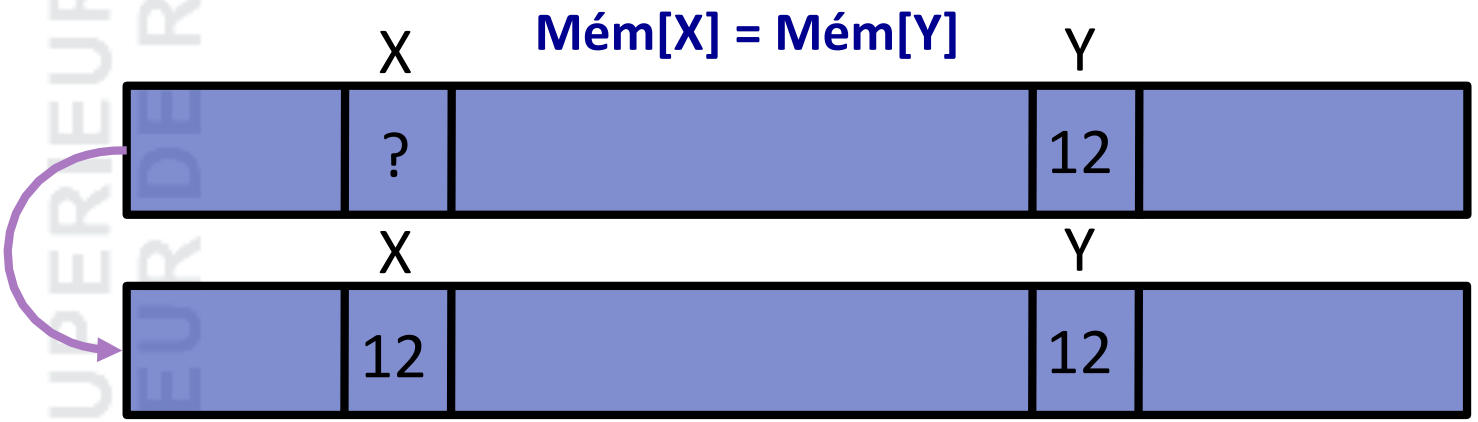
  X = * Y, noté < $*_R$, X, Y, _ >

  * X = Y, noté < $*_L$, _, X, Y >

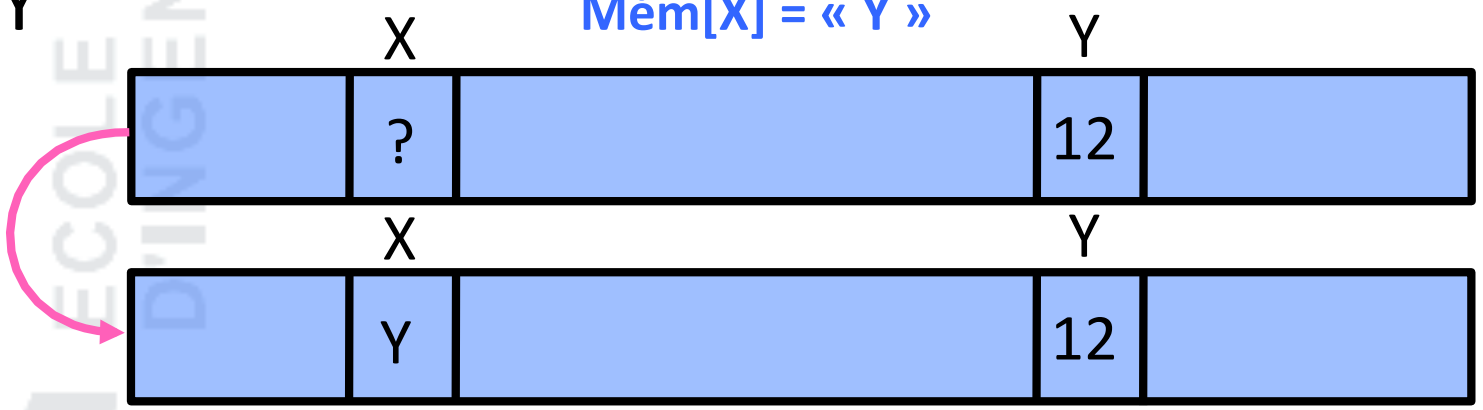
Aparté

X = Y vs X = &Y

- X = Y



- X = &Y

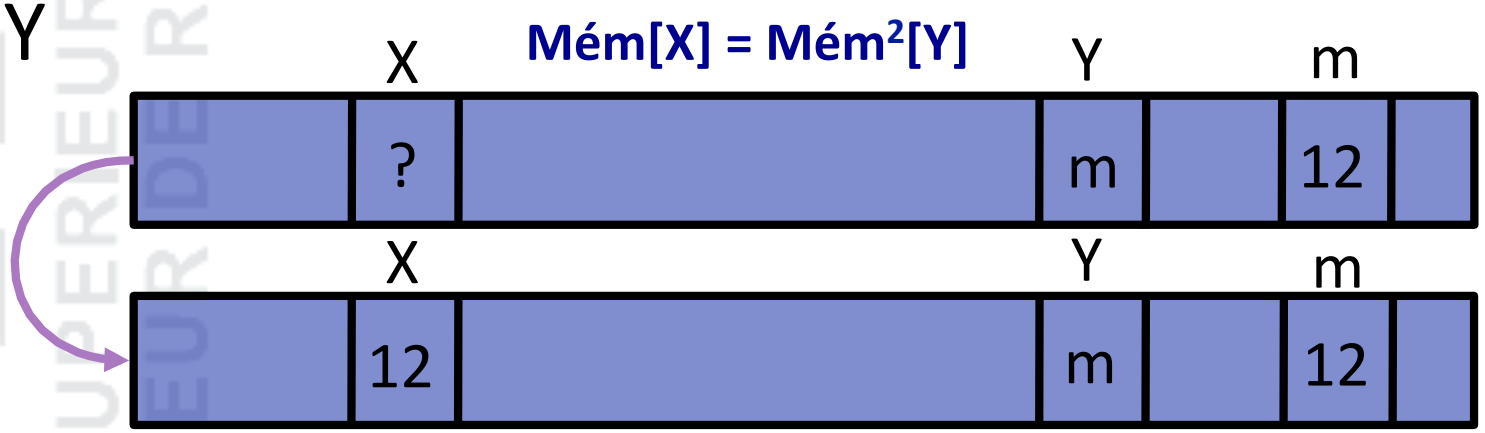


UNIVERSITÉ DE RENNES 1

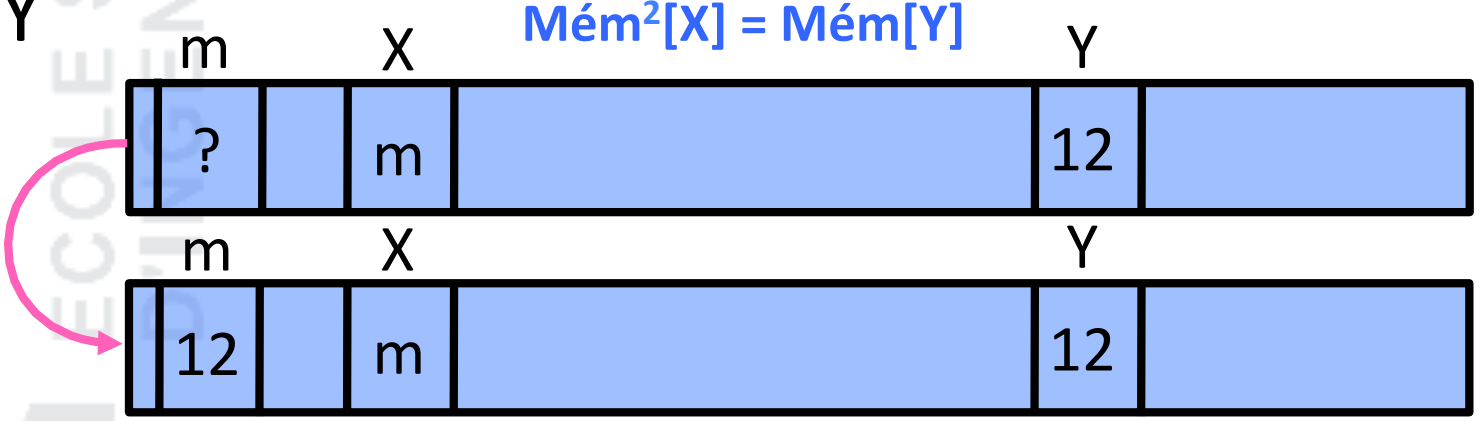
Aparté

X = * Y vs * X = Y

- X = * Y



- * X = Y



UNIVERSITÉ DE RENNES 1

Code 3 adresses (5)

- Des étiquettes

  étiquettes ≠ adresses de données

  L : < op, X, Y, Z >

- Des opérateurs de contrôle

  goto L, < **goto L**, _, _, _ >

  ifz X goto L, < **ifz L**, _, X, _ >

  …

Exemple – code intermédiaire (1)

- Le programme :

```
i = 1 ;
f = 1 ;
ttq i < n and f < MaxNum faire
    i = i+1 ;
    f = f * i
fait
```

Exemple – code intermédiaire (2)

- Le code :

```
        R1 = 1
        i = R1
        R2 = 1
        f = R2
boucle : R3 = i < n
        ifnz R3 goto L1
        goto sortie
L1 :    R4 = MaxNum
        R5 = f < R4
        ifnz R5 goto L2
        goto sortie
L2 : R6 = 1
        R7 = i + R6
        i = R7
        R8 = f * i
        f = R8
        goto boucle
sortie :
```

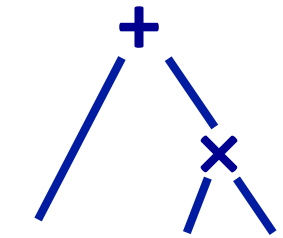
Génération de code

Production code
≈ linéarisation ASA

- Support structuré : ASA

⟶ support linéaire, liste d'instructions

sans perte de sémantique !

- Exemple : 

devient

$$R_1 = 2 \; ; \; R_2 = R_1 \times Z \; ; \; R_3 = X + R_2$$

Beaucoup de linéarisations possibles

- Exemple : $X + 2 \times Z$ peut devenir

$R_1 = 2 ; R_2 = R_1 \times Z ; R_3 = X + R_2$

$R_1 = 2 ; R_1 = R_1 \times Z ; R_1 = X + R_1$

$R_1 = Z ; R_2 = R_1 + Z ; R_3 = X + R_2$

$R_1 = Z ; R_1 = R_1 + Z ; R_1 = X + R_1$

$R_1 = X ; R_1 = R_1 + Z ; R_1 = R_1 + Z$

…

La prudence est de mise

- Exemple : **A×B + C×D** peut devenir

  $R_1 = A \times B$; $R_2 = C \times D$; $R_3 = R_1 + R_2$

ou

  $R_1 = A \times B$; $R_2 = C \times D$; $R_1 = R_1 + R_2$

mais pas

  $\underline{R_1 = A \times B}$; $\underline{R_1 = C \times D}$; $R_1 = R_1 + R_1$

  ☞ position conservatrice exigée

Stratégie générale de production de code

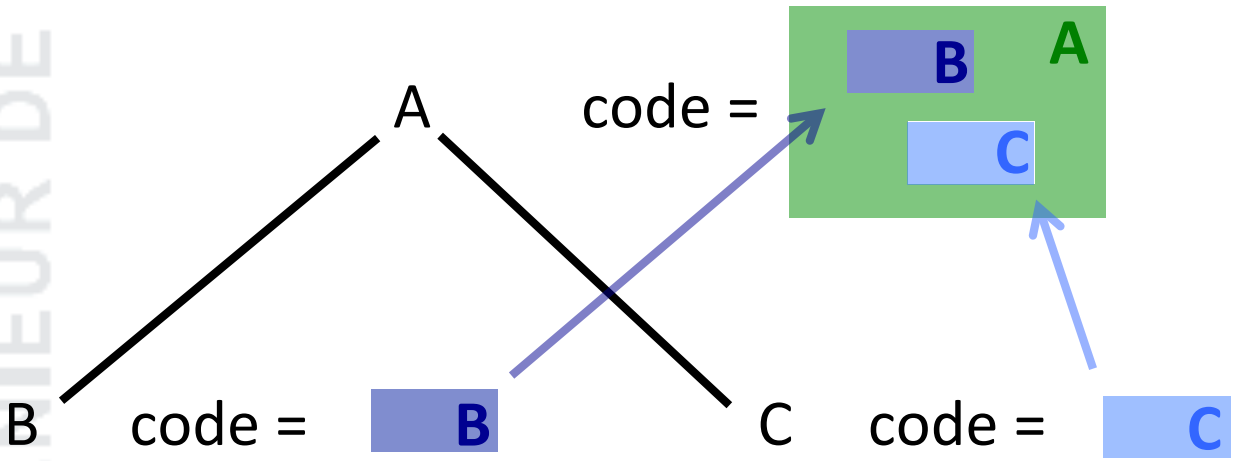
- Chaque nœud de l'ASA produit des bribes de code selon un patron prédéfini

...sans savoir ce que font les autres nœuds

- Compositionnalité
- Correction locale
- Définition d'une **interface**

Génération de code

- Au moins un attribut **synthétisé** : **code**



- Correction locale

$$SEM(B) \subset SEM(\mathbf{B.code}) \wedge SEM(C) \subset SEM(\mathbf{C.code})$$
$$\Rightarrow SEM(A) \subset SEM(\mathbf{A.code[B.code,C.code]})$$

Où est le résultat d'un calcul ?

- Machines à pile
  - au sommet de la pile
  - localisation **implicite**
- Machines à registres
  - dans une mémoire, laquelle ?
  - localisation doit être **explicitée**

⇒ Un attribut **place** pour localiser le résultat

Interface

- Code **linéaire**
  - entrée en haut
  - sortie en bas
- Code **spaghetti**
  - entrée nommée, fournie au contexte
  - sortie(s) nommée(s), fournie(s) par le contexte
  - **goto** mais pas **comefrom** !

Production de code 3 adresses

- Calcul
  - expressions
  - expressions booléennes
  - contrôle
- Données
  - implantation
  - accès mémoire

Représentation du calcul

Expressions (1)

$E \longrightarrow E' \text{ op } E''$

{ E.place = new-var()

E.code = E'.code • E''.code

• < op.val, E.place, E'.place, E''.place > }

code'
code''
place = place' op place''

Expressions (2)

E ⟶ cst

{ E.place = new-var()

E.code = < const cst.val, E.place, _, _ > }

E ⟶ id

{ E.place = id.place ; E.code = ε }

Expressions (3)

- Code intermédiaire
  = concaténation d'instructions

  < …, …, …, … > , ● et ε

- Réservation de nouvelles variables

  new-var()

- Traitement des constantes

  < const …, … >

Expressions (4)

- Traitements des variables sources

  id.place, accès à la table des symboles

- Latitude de choix

  E.code = E'.code • E''.code

  ou **E.code = E''.code • E'.code**

  mais sémantique du langage source

  FORTRAN : **A + (B + C) + D**

Expressions (5)

- **(a+b) × (a+b)** donne
  - **ε** place = a
  - **ε** place = b
  - **< add, r_1, a, b >** place = r_1
  - **ε** place = a
  - **ε** place = b
  - **< add, r_2, a, b >** place = r_2
  - **< mult, r_3, r_1, r_2 >** **place = r_3**

Expressions (6)

- Mécanisme universel qui marche pour toutes les expressions

consomme beaucoup de registres

mal adapté à certains domaines de calcul

ex. expressions booléennes

Consommation de registres

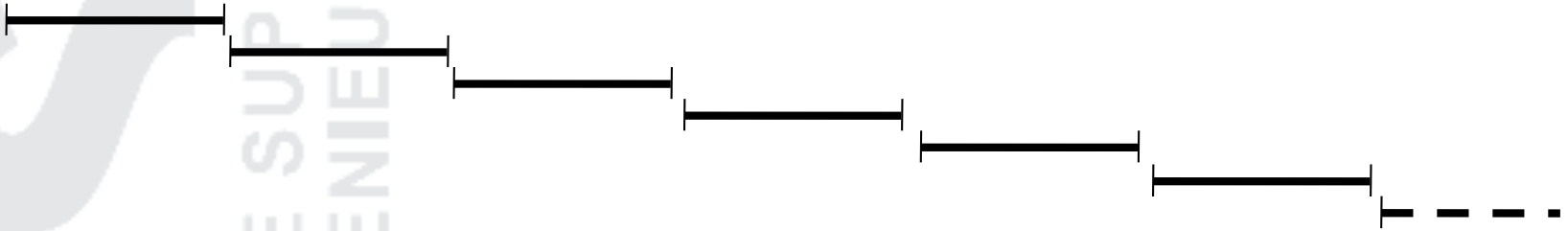
- Faire attention dès la production du code

...ou corriger plus tard,
lors de l'allocation des registres physiques

Exemple (1)

- a+(b+(c+(d+(e+(f+(g+h))))))

$g+h \rightarrow r_1$; $f+r_1 \rightarrow r_2$; $e+r_2 \rightarrow r_3$; $d+r_3 \rightarrow r_4$; $c+r_4 \rightarrow r_5$; $b+r_5 \rightarrow r_6$; $a+r_6 \rightarrow r_7$

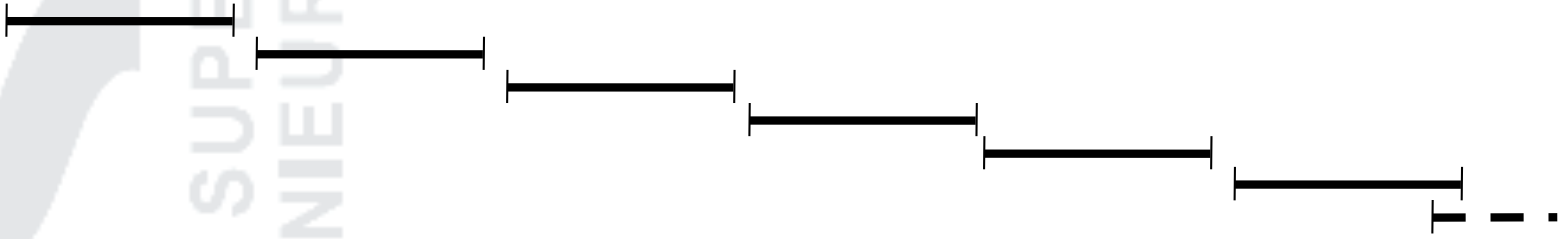


$g+h \rightarrow r_1$; $f+r_1 \rightarrow r_1$; $e+r_1 \rightarrow r_1$; $d+r_1 \rightarrow r_1$; $c+r_1 \rightarrow r_1$; $b+r_1 \rightarrow r_1$; $a+r_1 \rightarrow r_1$

Exemple (2)

- ((((((a+b)+c)+d)+e)+f)+g)+h

$a+b \rightarrow r_1$; $r_1+c \rightarrow r_2$; $r_2+d \rightarrow r_3$; $r_3+e \rightarrow r_4$; $r_4+f \rightarrow r_5$; $r_5+g \rightarrow r_6$; $r_6+h \rightarrow r_7$

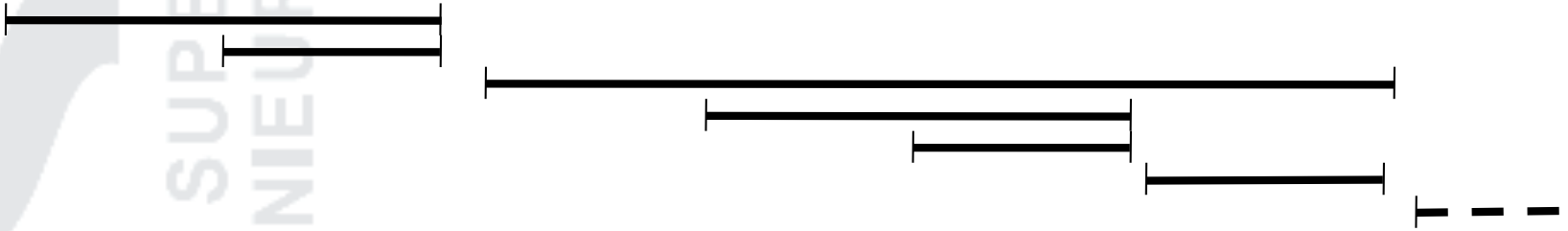


$a+b \rightarrow r_1$; $r_1+c \rightarrow r_1$; $r_1+d \rightarrow r_1$; $r_1+e \rightarrow r_1$; $r_1+f \rightarrow r_1$; $r_1+g \rightarrow r_1$; $r_1+h \rightarrow r_1$

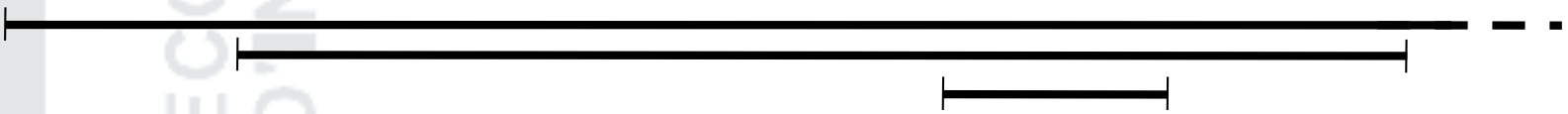
Exemple (3)

- ((a+b)+(c+d))+((e+f)+(g+h))

$a+b \rightarrow r_1$; $c+d \rightarrow r_2$; $r_1+r_2 \rightarrow r_3$; $e+f \rightarrow r_4$; $g+h \rightarrow r_5$; $r_4+r_5 \rightarrow r_6$; $r_3+r_6 \rightarrow r_7$



$a+b \rightarrow r_1$; $c+d \rightarrow r_2$; $r_1+r_2 \rightarrow r_1$; $e+f \rightarrow r_2$; $g+h \rightarrow r_3$; $r_2+r_3 \rightarrow r_2$; $r_1+r_2 \rightarrow r_1$



Registres nécessaires (1)

- Évaluer E_{small} op E_{big}
  - dans l'ordre

$place_{small}$ = évaluer E_{small} **utilise n registres, en garde 1, total = n**

$place_{big}$ = évaluer E_{big} **utilise N > n registres +1, en garde 1, total = N+1**

$place_{small}$ = $place_{small}$ op $place_{big}$ **utilise 2 registres, en garde 1**

  - à l'envers

$place_{big}$ = évaluer E_{big} **utilise N registres, en garde 1, total = N**

$place_{small}$ = évaluer E_{small} **utilise n < N registres +1, en garde 1, total ≤ N**

$place_{small}$ = $place_{small}$ op $place_{big}$ **utilise 2 registres, en garde 1**

☞ Évaluer le plus gros opérande d'abord

Registres nécessaires (2)
(en réordonnant)

- NbReg(E' op E'') =

  si NbReg(E') = NbReg(E'') alors NbReg(E')+1

  sinon max(NbReg(E'), NbReg(E''))

- NbReg(cst) = 1
- NbReg(id) = 0

Registres nécessaires (3)
(sans réordonner)

- NbReg(E' op E'') =

  si NbReg(E') > NbReg(E'') alors NbReg(E')

  sinon 1 + NbReg(E'')

- NbReg(cst) = 1
- NbReg(id) = 0

Expressions booléennes (1)

- Algèbre booléenne
  - des valeurs absorbantes
    - faux $\wedge$ **?** = faux
    - vrai $\vee$ **?** = vrai
- Un certain style de conditionnelles
  - en C : if (p != NULL $\wedge$ **p->a == ...**) ...
  - en C : (Cond ? Then : Else)
- Court-circuiter les évaluations **inutiles** ou **nuisibles**

Expressions booléennes (2)

- Ne pas calculer une valeur dans un registre, mais un comportement

  si vrai alors **faire ceci** sinon **faire cela**

- 2 attributs hérités, **si-vrai** et **si-faux**
- Le code produit provoque un branchement en **si-vrai** ou **si-faux** selon la valeur de l'expression

Exemple - code court-circuit

- (A and B) or C donne

ifnz A goto L_2

goto L_1

L_2 : ifnz B goto SiVrai

goto L_1

L_1 : ifnz C goto SiVrai

goto SiFaux

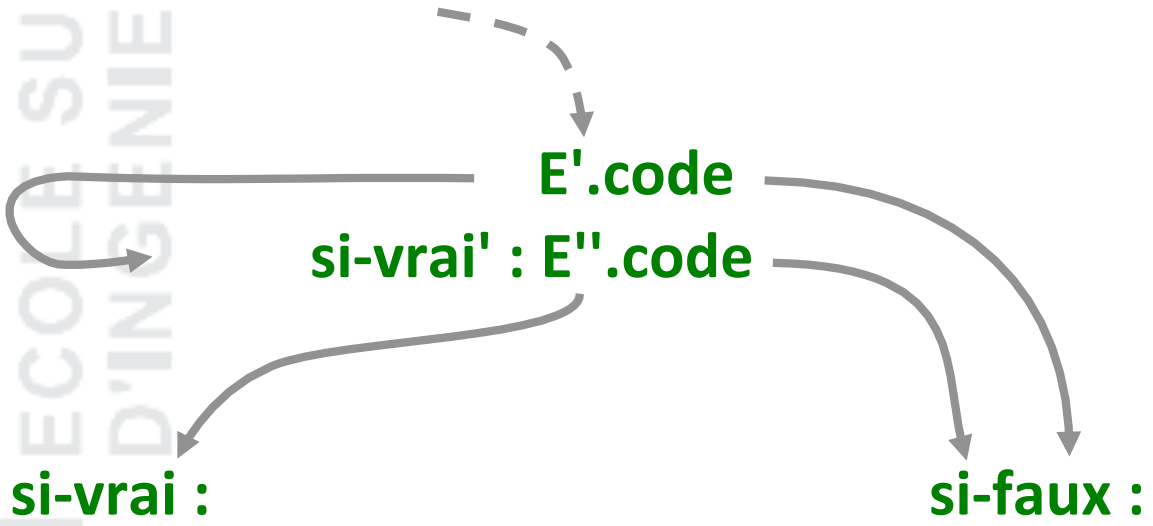
Expressions booléennes (3)

E ⟶ E' and E''

{ E'.si-faux = E.si-faux ; E'.si-vrai = new-label()

E''.si-faux = E.si-faux ; E''.si-vrai = E.si-vrai

E.code = E'.code ● E'.si-vrai : E''.code }



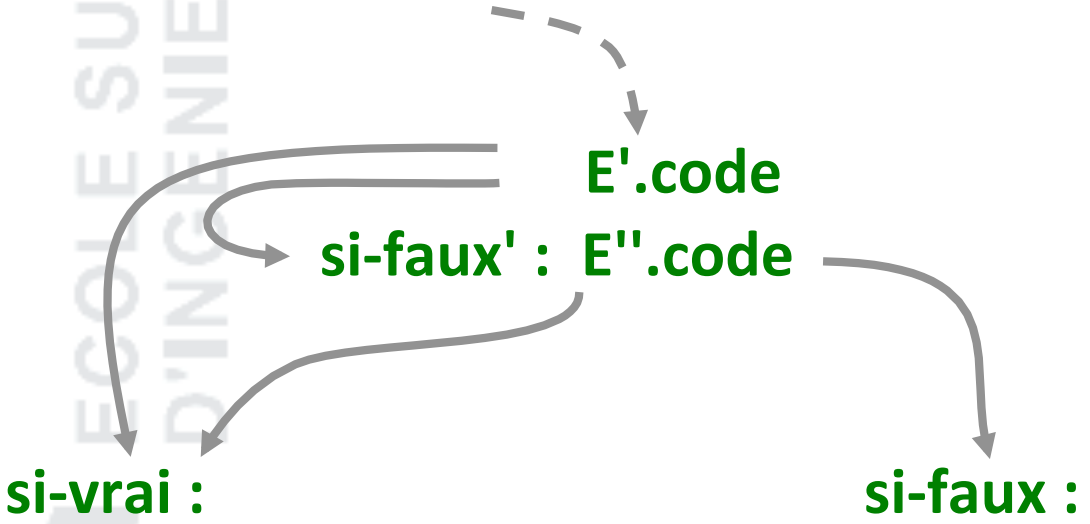
Expressions booléennes (4)

E ⟶ E' or E''

{ E'.si-faux = new-label() ; E'.si-vrai = E.si-vrai

E''.si-faux = E.si-faux ; E''.si-vrai = E.si-vrai

E.code = E'.code ● E'.si-faux : E''.code }

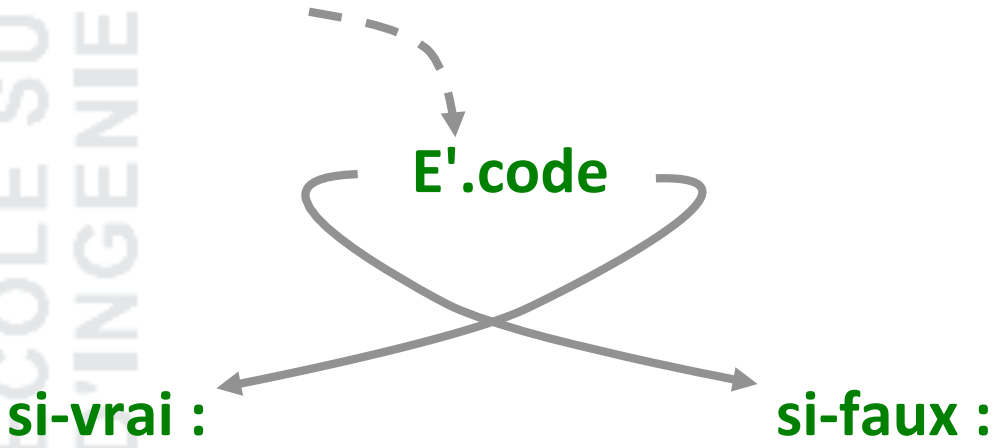


Expressions booléennes (5)

E ⟶ not E'

{ E'.si-faux = E.si-vrai ; E'.si-vrai = E.si-faux ;

E.code = E'.code }



Expressions booléennes (6)

E ⟶ E' rop E''

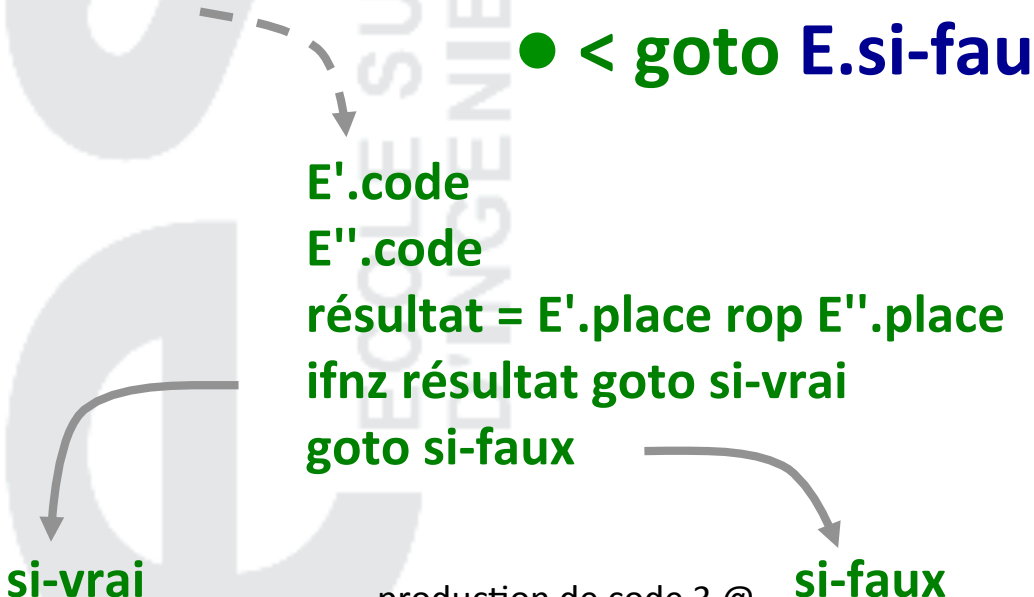
{ résultat = new-var() ;

E.code = E'.code ● E''.code

● < rop.val, résultat, E'.place, E''.place >

● < ifnz E.si-vrai, résultat, _, _ >

● < goto E.si-faux, _, _ > }



Expressions booléennes (7)

E ⟶ cst

{ E.code = < goto si cst.val = vrai alors E.si-vrai
sinon E.si-faux, _, _ > }

E ⟶ id { E.code = < ifnz E.si-vrai, _, id.place, _ >
● < goto E.si-faux, _, _ > }

Attention aux 3 langages

Représentation du contrôle

Instructions (1)

- Un attribut synthétisé **code**
- Reconstruire le flot de donnée
  - beaucoup d'étiquettes intermédiaires
  - **avec**/sans court circuit

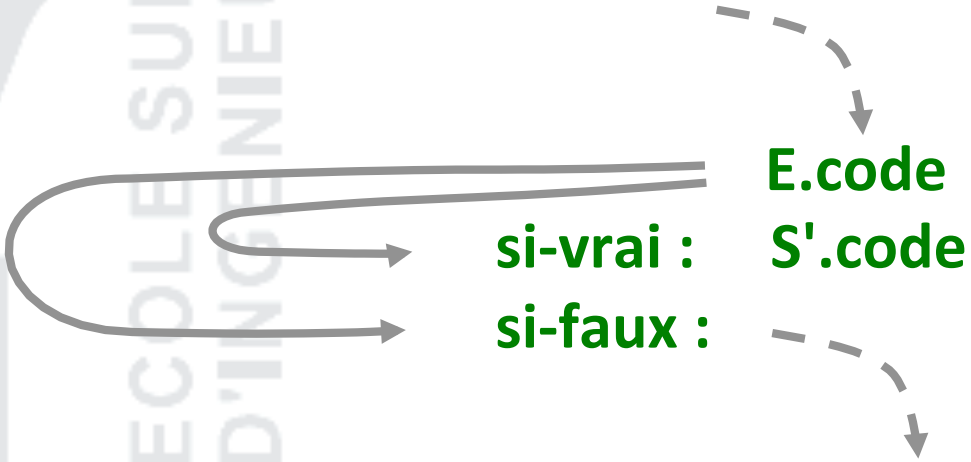
Instructions (2)

S ⟶ si E alors S' fsi

{ E.si-faux = new-label() ; E.si-vrai = new-label() ;

S.code = E.code ● E.si-vrai : S'.code

● E.si-faux : }



Instructions (3)

S ⟶ si E alors S' sinon S'' fsi

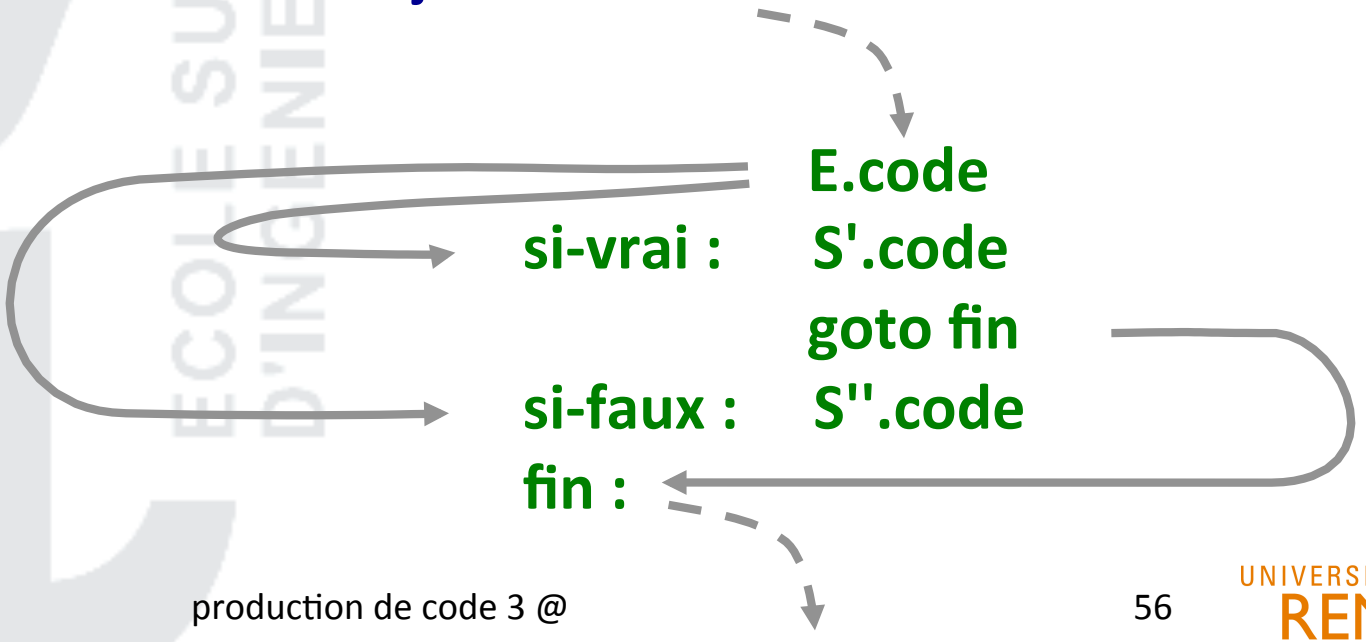
{ E.si-faux = new-label(); E.si-vrai = new-label();

fin = new-label()

S.code = E.code ● E.si-vrai : S'.code

● < goto fin, _, _, _ > ● E.si-faux : S''.code

● fin : }



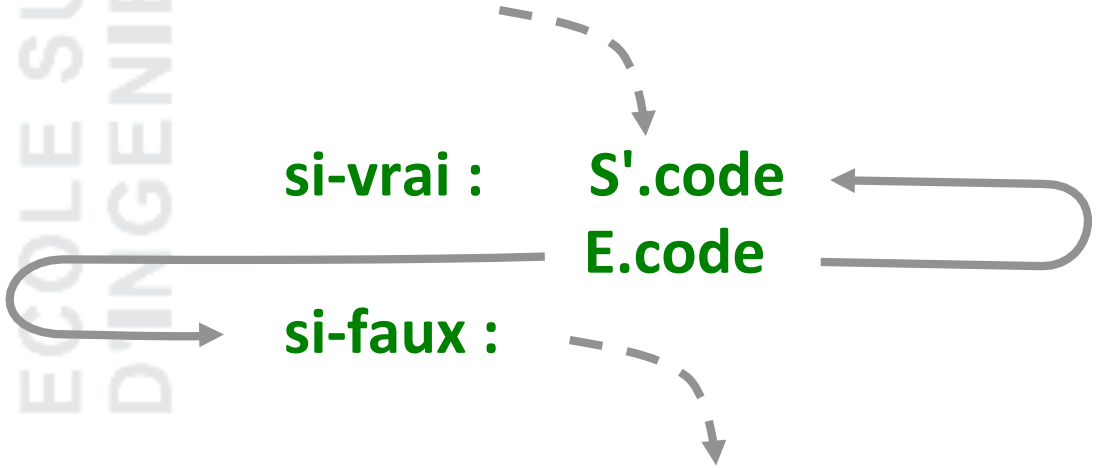
Instructions (4)

S ⟶ faire S' ttq E fait

{ E.si-faux = new-label(); E.si-vrai = new-label();

S.code = E.si-vrai : S'.code ● E.code

● E.si-faux : }



Instructions (5)

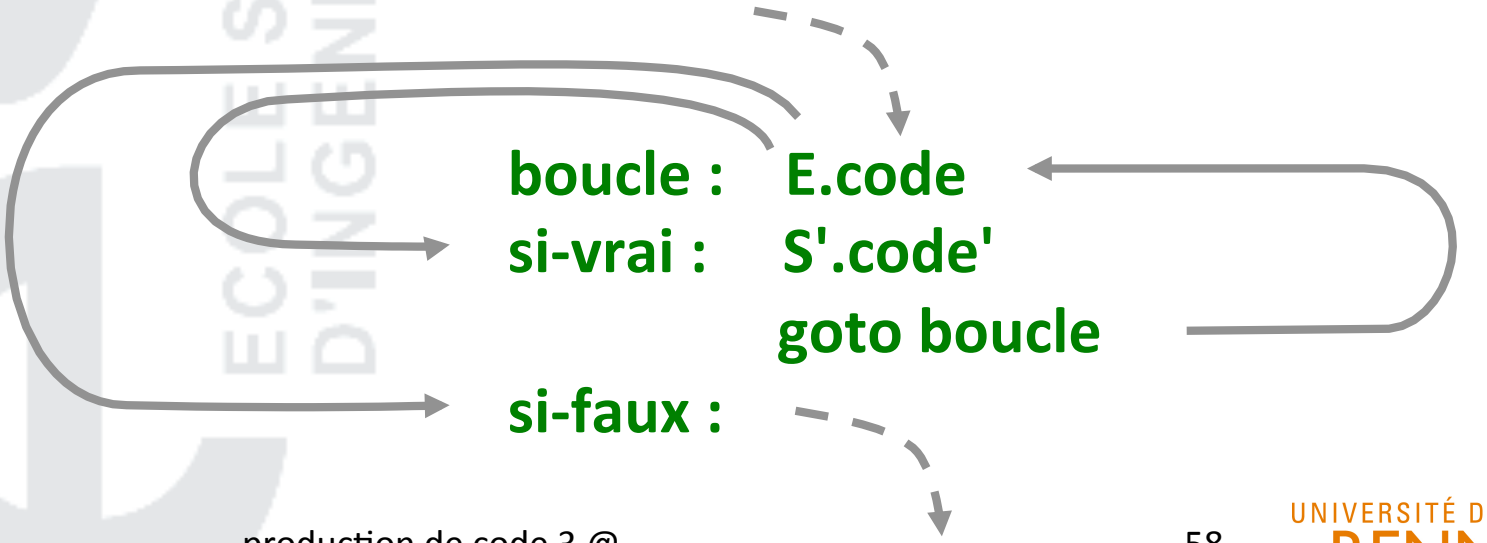
S ⟶ ttq E faire S' fait

{ E.si-faux = new-label(); E.si-vrai = new-label();

boucle = new-label()

S.code = boucle : E.code ● E.si-vrai : S'.code

● < goto boucle, _, _, _ > ● E.si-faux : }



Instructions (6)

$S \longrightarrow id = E$

$\{ S.code = E.code \bullet < =, id.place, E.place, \_ > \}$

$S \longrightarrow LS \quad \{ S.code = LS.code \}$

$LS \longrightarrow S\ ;\ LS' \quad \{ LS.code = S.code \bullet LS'.code \}$

$LS \longrightarrow S \quad \{ LS.code = S.code \}$

Instructions (7)

- Mécanisme universel qui marche pour toutes les instructions

- Consomme beaucoup d'étiquettes

Attention aux cascades de gotos

Conclusion temporaire

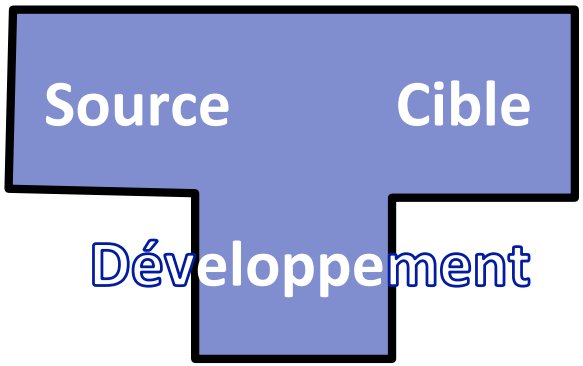
- Les 3 langages
- Penser global - agir local
- Attribut synthétisé **code**
- Attribut synthétisé **place**

pour calculer des valeurs

- Attributs hérités **si-vrai** et **si-faux**

pour court-circuit

- Reste structures de donnée et appels de fonction



Conclusion : les 3 langages

- Exemple : WHILE ⟶ C développé en Java

" ; " WH ≈ " ; " C ?

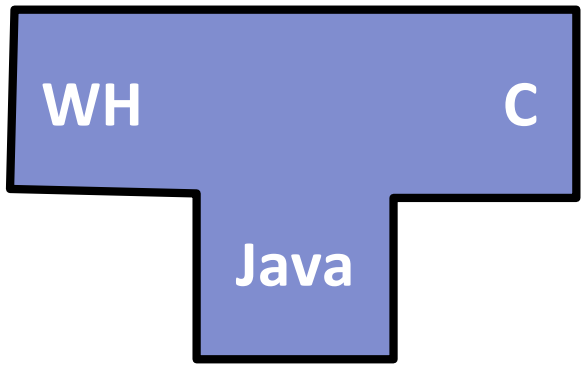
"if" WH ≈ "if" C ?

"while" WH ≈ "while" C ?

"for" WH ≈ "for" C ?

valeur WH ≈ valeur C ≈ **valeur Java** ?

variable WH ≈ variable C ?



Représentation des données

Les structures de données

- La structure
  - plusieurs champs **singuliers** accessibles simultanément
- Le tableau
  - plusieurs éléments **réguliers** accessibles simultanément
- Le pointeur
  - notion de **référent/référé**

Référent / référé (1)

- Un référé : M. Adolphe Thiers
- 3 référents (au moins) :
  - Foutriquet
  - le 1er Président de la IIIème République
  - Adolphe Thiers
- Foutriquet est ridicule
- « Foutriquet » est ridicule

Référent / référé (2)

- Un référé : 12
- 3 référents (au moins) :

12 **7+5** **X (si X == 12)**

7+5 est pair / "7+5" est pair

X est pair / &X est pair

&12 et &(7+5) ont rarement du sens

Référent / référé (3, en C)

- int X ;

X est un **référent constant** pour une mémoire

...qui est un **référent variable** pour des valeurs

X = 12 ;

Référent / référé (4, en C)

- int X[12] ;

  X est un **référent constant** pour un ensemble de mémoires (le tableau)

  X[3] est un **référent constant** pour une mémoire

  ...qui est un **référent variable** pour des valeurs

  X[3] = 12 ;

on ne peut pas écrire X = ... ;

Référent / référé (5)

- Position de référent de valeur

X+5

X ⟶ mémoire ⟶ contenu

contenu ⟶ mémoire est impossible

- Position de référent de mémoire

X = 12 , & X , X[12] , X.f

X ⟶ mémoire

Distinguer les deux types de positions

Référent / référé (6)

- **E ⟶ … | A | &A**
- **A ⟶ id | A[E] | A.id | *A**

Dépendances mutuelles

- Pointeur sur (élément de) tableau
- Pointeur sur (champ de) structure
- Tableau de pointeurs
- Structures de pointeurs

traiter pointeurs,

puis structures,

puis tableau

Le pointeur (1)

- Adresse homogène à une donnée
  - même un peu d'arithmétique !
- Deux opérations
  - dérepérage, *
    - calcule le référé d'un référent
  - repérage, &
    - inhibe le dérepérage
    - **n'est pas le calcul du référent d'un référé !**

    ≈ guillemets

Le pointeur (2)

A ⟶ id

{ A.place = new-var()

A.code = < &, A.place, id.place, _, _ > }

A ⟶ * A'

{ A.place = new-var()

E.code = A'.code

- **< $*_D$, A.place, A'.place, _ > }**

E ⟶ & A

{ E.place = A.place ; E.code = A.code }

La structure (1)

- Adressable en totalité ou en partie

  &(s.f)

- Une opération
  - sélection de champs
    - adresse d'une structure ⟶ contenu d'un champ
    - adresse d'une structure ⟶ adresse d'un champ
  - en lecture ou en écriture
  - en cascade : s.a.b.c

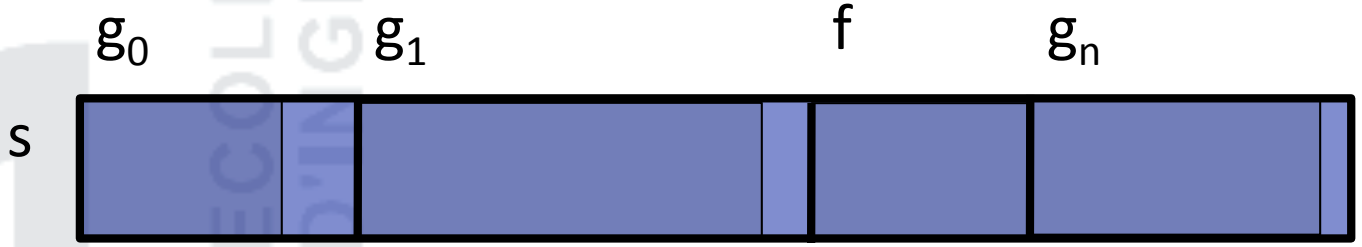
La structure (2)

- Modèle mémoire
  - champs consécutifs en mémoire

Adr(s.f) = Adr(s) + Depl(f)

Depl(f) = $\sum_{g \text{ « à gauche de » } f}$ Occup(g)

Occup(f) = arrondi sup. Taille(f)



La structure (3)

A ⟶ A' . id

{ E.place = new-var()

depl = selStruct(A'.type,id).depl

A.code = A'.code

- **< const depl, A.place, _, _ >**
- **< +, A.place, A'.place, depl > }**

Le tableau (1)

- Adressable en totalité ou en partie

  &(t[i])

- Une opération
  - indexation
    - adresse d'une structure ⟶ valeur d'un élément
    - adresse d'une structure ⟶ adresse d'un élément
  - en lecture ou en écriture
  - en cascade : t[i][j][k]
  - multidimensionnel : t[i,j,k]

Le tableau (2)

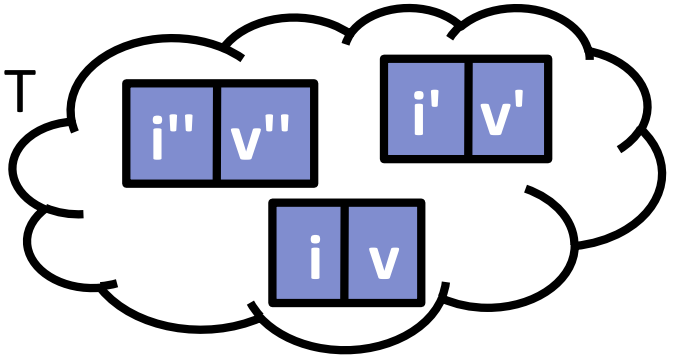
- Définir la sémantique !
  - domaine des indices

    [0 n-1] ou [1 n] ou [a a+n-1]
  - extensible ou **non**
    - changement de taille d'un tableau
  - dynamique ou **non**
    - calcul de tableau
  - déplaçable ou **non**

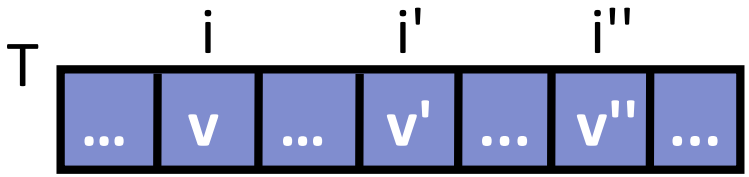
Le tableau (3)

- Définir la mise en œuvre !
  - rangement compact ou dispersé
  - ordre des dimensions
  - tableau plein ou creux

Cas unidimensionnel (1)



- Rangement dispersé
  - coûteux
  - excellent pour tableaux creux
  - permet tableaux dynamique ou extensibles
  - nature du nuage ?
- **Rangement compact**
  - efficace
  - contraignant



Cas unidimensionnel (2)

- Rangement compact

  $Adr(t[i]) = Adr(t) + i \times Occup_{élem}$

  $Occup_{élem}$ = arrondi sup. $Taille_{élem}$

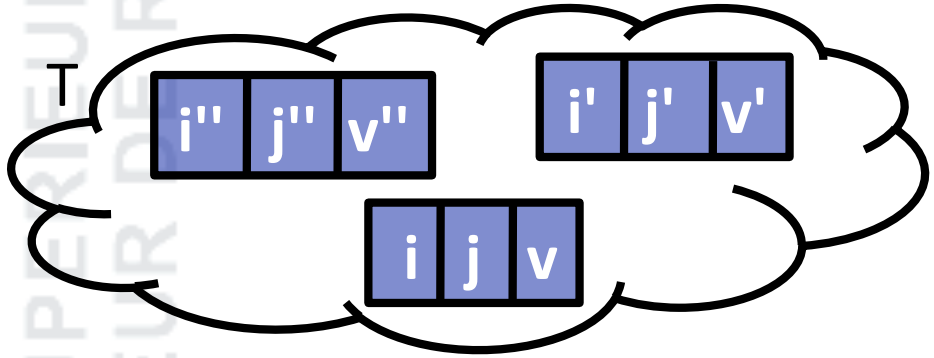
- Variante

  $Adr(t[i]) = Adr(t) + (i - \mathbf{bi}) \times Occup_{élem}$

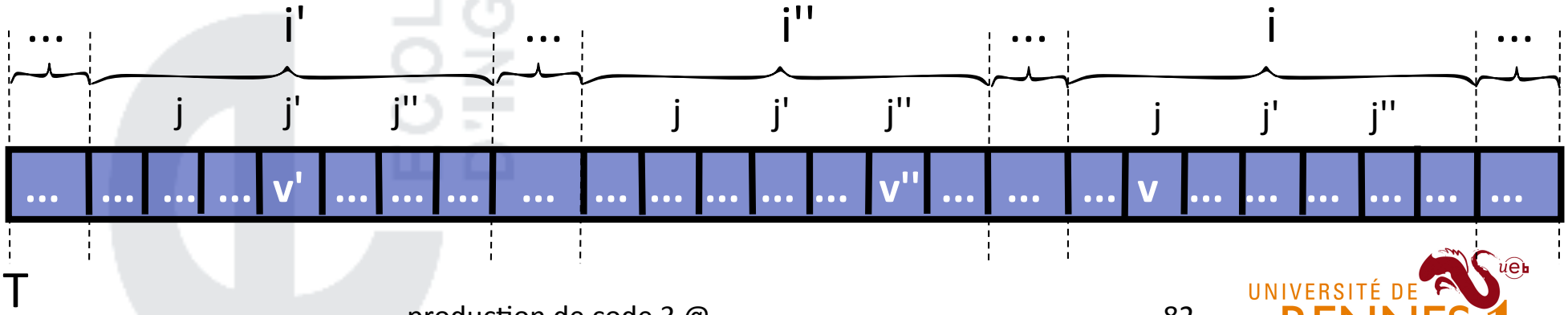
  $= Adr(t) + i \times Occup_{élem} - \mathbf{bi \times Occup_{élem}}$

Cas bidimensionnel (1)

- Rangement dispersé



- **Rangement compact**



Aparté - **la tyrannie de la décomposition dominante**

- Dissymétrie ligne / colonne
  - une dimension favorisée
  - l'autre sacrifiée
- Problème universel
  - projection d'une structure multi-dimensionnelle sur une structure unidimensionnelle
  - ex. système de fichiers, structure des programmes

Cas bidimensionnel (2)

- Rangement compact

$$Adr(t[i,j]) = Adr(t) + i\times\underline{NbCol\times Occup_{élem}} + j\times\underline{Occup_{élem}}$$

$$= Adr(t) + [i\times\underline{NbCol} + j]\times\underline{Occup_{élem}}$$

- Variante

$$Adr(t[i,j])$$

$$= Adr(t) + (i - i_0)\times\underline{NbCol\times Occup_{élem}} + (j - j_0)\times\underline{Occup_{élem}}$$

$$= Adr(t) + i\times\underline{NbCol\times Occup_{élem}} + j\times\underline{Occup_{élem}}$$

$$- \underline{i_0\times NbCol\times Occup_{élem}} - \underline{j_0\times Occup_{élem}}$$

$$= Adr(t) + [(i\times\underline{NbCol} + j) - \underline{(i_0\times NbCol + j_0)}]\times\underline{Occup_{élem}}$$

Cas multidimensionnel (1)

- Rangement compact

  Adr(t[i,j,k, ...,l]) = Adr(t)

  + i×**NbJ×NbK×...×NbL×Occup**$_{élem}$

  + j×**NbK×...×NbL×Occup**$_{élem}$

  + k×**...×NbL×Occup**$_{élem}$ + ...

  + l×**Occup**$_{élem}$

  – beaucoup d'opérations arithmétiques

  mais souvent beaucoup de constantes

  = Adr(t) + ((...((i×**NbJ** + j)×**NbK** + k)×**...**)×**NbL** + l)×**Occup**$_{élem}$

  – efficace même quand pas de constantes

Cas multidimensionnel (2)

- Variante

Adr(t[i,j,k, ...,l]) = Adr(t)

+ (i $-i_0$)×**NbJ×NbK×...×NbL×Occup$_{élem}$**

+ (j $-j_0$)×**NbK×...×NbL×Occup$_{élem}$**

+ (k $-k_0$)×**...×NbL×Occup$_{élem}$** + ...

+ (l $-l_0$)×**Occup$_{élem}$**

= Adr(t)

+ ((...((i×**NbJ** + j)×**NbK** + k)×**...**)×**NbL** + l)×**Occup$_{élem}$**

-**((...((i$_0$×NbJ + j$_0$)×NbK + k$_0$)×...)×NbL + l$_0$)×Occup$_{élem}$**

Cas multidimensionnel (3)

- Descripteur de tableau

  < **NbJ**, **NbK**, **...**, **NbL**, **$Occup_{élem}$** >

...ou bien

< **$NbJ \times NbK \times ... \times NbL \times Occup_{élem}$** ,

$NbK \times ... \times NbL \times Occup_{élem}$,

... ,

$NbL \times Occup_{élem}$,

$Occup_{élem}$ >

Le tableau (4)

A ⟶ A' [E]

{ A.place = new-var()

taille = valArray(A'.type).taille

A.code = A'.code ● E.code

- ● < const taille, A.place, _, _ >
- ● < ×, A.place, E.place, A.place >
- ● < +, A.place, A'.place, A.place > }

Les appels de fonction (1)

- Préparer les paramètres
- Effectuer l'appel

...en abstrayant les détails

Les appels de fonction (2)

E ⟶ (call id LA)

{ E.place = new-var()

E.code = LA.code

● < call id.val, E.place, LA.n, _ > }

Les appels de fonction (3)

LA ⟶ LA' , E

{ LA.n = LA'.n + 1

LA.code = LA'.code ● E.code

● < arg, _, E.place, _ > }

LA ⟶ E

{ LA.n = 1

LA.code = E.code ● < arg, _, E.place, _ > }

Conclusion (1)

- Une conception **systématique**
  - les expressions
    - de valeur (booléenne ou non)
    - de mémoire
  - les instructions

- Actions locales pour un effet global

Conclusion (2)

- Des patrons de **code** bien distincts
  - **linéaire**

    = 1 entrée en haut, 1 sortie en bas, pas de branchement
  - **court-circuit**

    = 1 entrée en haut, 2 sorties nommées (si-vrai, si-faux)
  - **bien structuré**

    = 1 entrée en haut, 1 sortie en bas, mais pas linéaire dedans

Conclusion (3)

- Pour les expressions
  - calcul d'une valeur dans une **place**
  - calcul d'un comportement
  - calcul d'une adresse dans une **place**

Conclusion (4)

- Distinguer les **référents de mémoire**

  souvent **constants**

...et les **référents de valeurs**

souvent **variables**

Conclusion (5)

- Des attributs **synthétisés** pour la plupart
  - sauf **si-vrai** et **si-faux**

  ...**hérités** du contexte

Application au projet WHC (1)

- WH ⟶ MIPS ou MV3@
  - le passage par le code 3 adresses est **idéal**
- WH ⟶ C (ou JAVA, etc)
  - le passage par le code 3 adresses doit être **aménagé...**

  ...identifier dans le code intermédiaire les structures C qui seront visées

Application au projet WHC (2)

- Aménager le code 3 adresses
  - ex. introduire une construction

    if place then code else code'

    < if L L', place, _, _ >

    ...ou bien

    while place do code

    < while L, place, _, _ >
  - dans le style **< arg, ... >** et **< call L, ...>**
  - garder le code 3 adresses pour les expressions et les séquences

Application au projet WHC (3)

- Dans tous les cas, prévoir de nouvelles instructions

  X = Y cons Z ou **< cons, X, Y, Z >**

  X = nil ou **< nil, X, _, _ >**

  X = Y =? Z ou **< =? , X, Y, Z >**

  X = hd Y ou **< hd, X, Y, _ >** (tl aussi)

  ...et d'autres si nécessaire

  ifcons X goto L ou **ifnil X goto L**

  ...

Application au projet WHC (4)

- Prévoir aussi la réalisation des opérations **cons**, **nil** et **=?**
  - allocation de mémoire
  - initialisation de l'espace mémoire
  - comment reconnaître un nil d'un cons
  - procédure de décision

  ...le système *runtime* WH (la **libWH**)