

# Compilation

## Plan du cours

- Chapitre I : Introduction : Rappels sur les étapes de la compilation
- Chapitre II : Traduction dirigée par la syntaxe
- Chapitre III : Allocation substitution
- Chapitre IV : Organisation des données à l'exécution
- Chapitre V : Optimisation du code
- Chapitre VI : Génération du code.

## Chapitre I : Introduction à la Compilation

### Introduction :

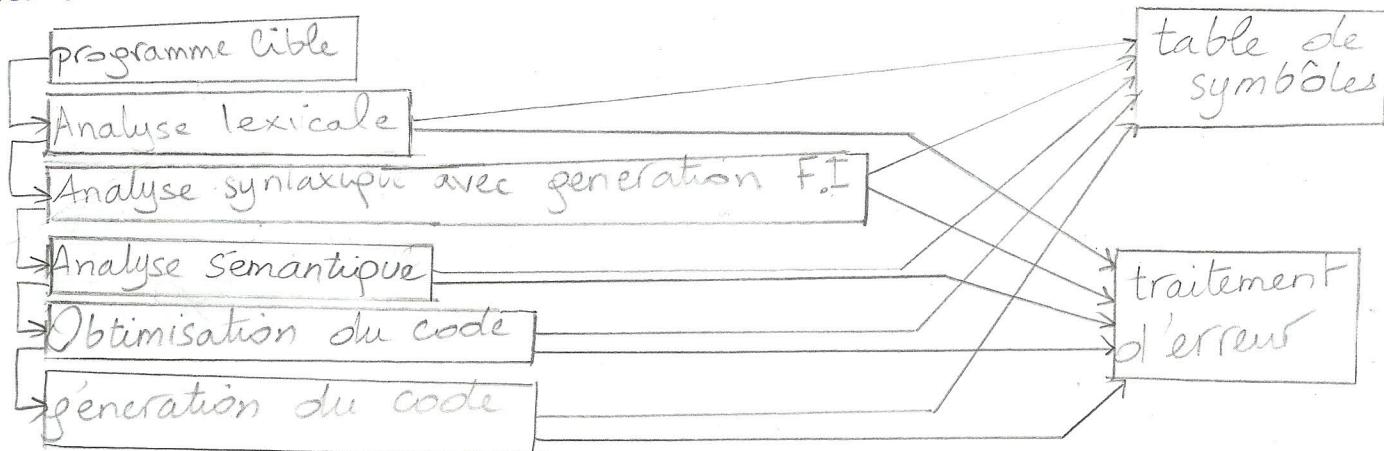
La compilation est le processus permettant de traduire un programme écrit dans un langage source vers des instructions machine.

Un compilateur est structuré en une chaîne de traitement appelé phase de compilation.

A travers ces phases passe le programme sous des représentations divers. Chaque phase reçoit de la précédente une représentation particulière du programme, et émis vers la suivante une autre représentation plus proche du programme cible.

De plus toutes les phases consultent et mettent à jour une table de symboles, où s'accumulent les données.

Le schéma suivant donne les différentes phases d'un compilateur



1- Analyse lexicale : La phase d'analyse lexicale lit le programme source caractère par caractère et tente de reconnaître une suite d'éléments lexicaux. Cette phase se base sur les automates à état finis.

Elle peut échouer et doit être capable de signaler des erreurs dite erreur lexicale

Exemple Construire un automate déterministe qui reconnaît les noms des procédures Cobol composé de lettres, de chiffres et de tirets, il ne peut pas commencer ni finir par un trait, et ne contiennent pas deux traits consécutifs. La longueur du nom ne dépasse pas 80 car.

### Algorithm de reconnaissance

Début

lire (Entité);

$E_c = S_1$ ;

$T_c = 1^{\text{er}}$  caractère de l'entité;

$TQ (E_c \neq \emptyset \text{ et } T_c \neq \#)$

faire

$E_c := T[E_c, T_c]$ ;

$T_c := t_s ; cpt++$ ;

fait

Si  $E_c = \emptyset$  Alors Écrire ("Entité Erronée");

Sinon

Si  $E_c = S_2$  Alors

Si  $cpt > 80$  Alors Écrire ("Nom trop long");

Sinon

Coder;

Insérer TS;

FinSi

FinSi

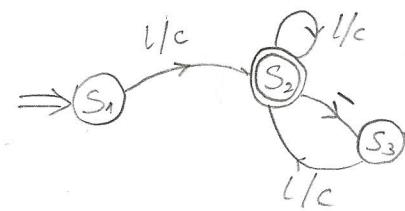
FinSi

Fin

2. Analyse Syntaxique: A comme but de reconnaître la suite des entités reconnus en lexicale, elle se base sur la grammaire Algébrique et les automates à pile. Le résultat de cette phase est un arbre syntaxique, cette phase est la mieux formaliser de toute la compilation.

Il existe deux méthodes d'analyse d'un programme. Les méthodes ascendantes et les méthodes descendantes.

1. Les méthodes descendantes : Elles consistent de partir de l'axiome et par une série de dérivation aboutir au programme à analysé syntaxiquement. Il existe deux types d'analyse descendante.



	l	c	-
S1	S2	S2	
S2	S2	S2	S3

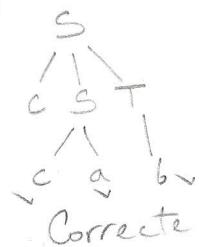
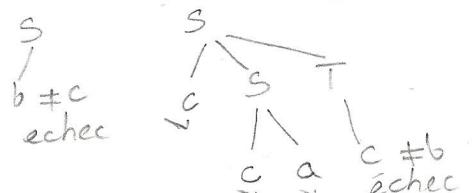
## a. Non Déterministe

1. Parallel : Elle consiste à construire plusieurs arbres syntaxiques en même temps jusqu'à aboutir à un succès, si tous les arbres n'aboutissent pas, alors le programme est erroné syntaxiquement et le problème de cette méthode est la perte de temps et d'espace mémoire.

2. Retour arrière Un arbre syntaxique est démarrer et si un chemin ne marche pas on va essayer un autre, et si aucun chemin ne marche le programme est erroné syntaxiquement

Exemple :

$$S \rightarrow ca/b/cST$$
$$T \rightarrow c/b$$



Correcte

Analyser la chaîne ccab#.

Le problème est même que pour la méthode précédente donc avec la méthode non déterministe y a une perte de temps et d'espace mémoire

b. Déterministe Ces méthodes consistent à emprunter un chemin et aller jusqu'au bout sans retour arrière.

Plusieurs méthodes existent pour faire une analyse syntaxique descendante déterministe (LL(1), LL(K), Descendante recursive, automate)

Avant de faire une analyse descendante déterministe faut s'assurer que la grammaire n'est pas recursive gauche

LL(1) : Une grammaire est dite LL(1) Si :

1. non recursive gauche
2. Factorisée
3. Table LL(1) mono défini.

2. Les méthodes Ascendante : Elles consistent de partir du programme à analyser et à faire une série de réduction jusqu'à aboutir à l'axiom. Il existe plusieurs méthodes d'analyse ascendante SLR(1),

Les compilateurs existant en littérature

Se base sur la méthode LALR.

Cette étape peut signaler des erreurs appeler erreurs syntaxique.

SLR(1),  
LR(K),  
LALR(1)  
LALR(K)  
SLR(1)  
SLR(K)

3. Génération des formes Intermediaire C'est une phase optionnelle mais qui peut faciliter l'exécution des programmes, parmis les formes intermédiaire nous citant le poste fixé, le préfixé, les triplets, les quadruplets, les arbres abstraits

## Forme intermédiaire

### Exemple

If  $a > b$  then  $b := b * a;$   
 else  $c := a * c;$

### Quadruplets:

quad (BGE, else, a, b)  
 then (\*, b, a, b)  
 (BR, Fin, , )  
 else (\*, a, c, c)  
 forme Post fixée  
 $a - b - \underline{\text{else}}; \text{BPZ } bb * = \text{fin BR}$   
 $c a c * := \uparrow$   
 fin

| (-, a, b, T1)  
 | (BPZ, else, T1)

10	
BP(>0)	BPZ( $\geq 0$ )
BM(<0)	BME( $\leq 0$ )
BZ(=0)	BNZ( $\neq 0$ )
a/b	
BG(>)	BGE(>=)
BL(<)	BLE(<=)
BE(=)	BNE( $\neq$ )

### Analyse sémantique:

L'Analyse sémantique est une étape de traitement liée à la syntaxique cette étape se base sur une grammaire appelé grammaire sémantique elle peut être ascendante ou descendante.

Cas descendant Si une routine syntaxique doit être insérée on crée un nom terminal dérivant en E

Si

$$\begin{aligned} A &\rightarrow \alpha \beta \\ A &\rightarrow \alpha \beta \\ B &\rightarrow E \end{aligned}$$

Cas ascendant Si une routine syntaxique doit être insérée on procède au découpage de grammaire si :  $A \rightarrow \alpha / \beta$

Comme la phase précédente cette phase peut échouer, et doit être capable de signaler des erreurs dite syntaxique.

Optimisation du code : d'une manière générale le rôle de l'optimisation est de trouver des informations sur le comportement à l'exécution et d'utiliser ces informations pour améliorer le code généré par le compilateur en utilisant l'optimisation pour les boucles imbriquées.

- Eliminer les fonctions jamais utilisées.

- Eliminer les variables jamais utilisées

## Exemple

For  $i = 1$  to  $n$

Do

| for  $j = 1$  to  $n$

| Do

| |  $a[ij] = b[ji]$

| EndDo

EndDo

## Code Optimisé

for  $j = 1$  to  $n$

  for  $i = 1$  to  $n$

$a[ij] = b[ij]$

## Génération du code :

Cette étape a pour donnée un programme compilé et produit comme résultat un programme cible ???

## Chapitre II Introduction dirigé par la syntaxe

L'analyse syntaxique permet de vérifier le sens de la phrase.

Exemple : vérifier si un identificateur a été déclaré,

compatibilité des types dans une expression

l'utilisation correct d'une étiquette.

Notre but est de récupérer le code intermédiaire auquel on ajoute des actions ou des contrôles sémantiques.

Une routine sémantique est nécessaire après une réduction si l'agit d'une analyse ascendante ou bien après une dérivation dans le cas descendant.

Schema de traduction C'est l'ensemble de grammaire.

La grammaire (associé à grammaire syntaxique transformée.)

Code intermédiaire

des routines sémantiques.

1/ Cas descendant

Exemple :

1/  $\langle \text{inst\_if} \rangle \rightarrow \text{if} \langle \text{cond} \rangle \text{ then} \langle \text{inst 1} \rangle \text{ else} \langle \text{inst 2} \rangle$

2/ Code intermédiaire = (sous forme quadruplets)

quadruplet de  $\langle \text{cond} \rangle \rightsquigarrow T, \text{cond}$  (BZ, else, T, cond)

Quad ( $\langle \text{inst 1} \rangle$ )

(BR, fin, , )

Quad ( $\langle \text{inst 2} \rangle$ )

Fin

### 3°/ grammaire transformées "<"

$\langle \text{inst\_if} \rangle \rightarrow \text{If } \langle \text{cond} \rangle \langle A \rangle \text{ then } \langle \text{inst1} \rangle \langle B \rangle \text{ else } \langle \text{inst2} \rangle$

### 4°/ Routine sémantique

#### Routine (A)

Début

Quad ( $Q_c$ ) := (BZ, -, T, cond)

Sav-BZ :=  $Q_c$  ;

$Q_c++$  ;

fin

#### Routine (c)

Début

Quad (Sav-BZ, 2) =  $Q_c$  ;

Fin

#### Forme intermédiaire

#### Exemple

If  $a < b$  then  $b := b * a$  else  $c := a * c$ ;

Quadruplets:

(BGE, else, a, b)

(\* , b , a , b)

(BR, fin, , )

(\*, a, c, c)

Forme post fixé

ab-else BRZ bb a \* = fin BR CaC - := Fin

**Analyse sémantique** L'analyse sémantique est une étape étroitement liée à la syntaxe, cette étape se base sur une grammaire appelée grammaires sémantiques et l'analyse peut être ascendante ou descendante

**Cas descendante** Si une routine sémantique doit être insérée ou créée un non-terminal devient en  $\epsilon$

Si:  $A \rightarrow \alpha B \Rightarrow \begin{cases} A \rightarrow \alpha B \\ B \rightarrow \epsilon \end{cases}$

**Cas ascendant** Si une routine sémantique doit être insérée on procède au découpage de la grammaire Si:  $A \xrightarrow{B} \alpha B \rightarrow \begin{cases} A \rightarrow \alpha B \\ B \rightarrow \epsilon \end{cases}$

Comme la phrase précédente cette phase peut échouer et donc être capable de signaler des erreurs dite sémantique.

**Optimisation du code** D'une manière générale le rôle de l'optimisation et de trouver/chercher des infos sur le comportement à l'exécution et l'utiliser cette info pour améliorer le code généré par le compilateur imbriquées.

#### Boucles imbriquées

• Éliminer fonctions jamais utilisées

• Éliminer les variables jamais utilisées

Exemple: for( $i=1$  to  $n$ ) Do for  $j=1$  to  $n$

do  $a[i,j] = b[j,i]$  EndDo; EndDo; - 6 -

#### Routine (B)

Début

Quad ( $Q_c$ ) := (BR, -, , )

Sav-BR =  $Q_c$  ;

$Q_c++$  ;

Quad (Sav-BZ, 2) :=  $Q_c$  ;

Fin

#### Code optimisé

Génération du code: Cette étape a pour donner un programme compilé et produit comme résultat un programme cible équivalent.

Exemple 2 :

$\langle \text{inst-for} \rangle \rightarrow \text{for } \text{id} := \langle \text{exp1} \rangle \text{ to } \langle \text{exp2} \rangle \text{ step } \langle \text{exp3} \rangle \text{ Do } \langle \text{inst} \rangle$

Donner le schéma de traduction de cette instruction

Les quadruples :

{ Quadruplets ( $\text{exp1}$ )  $\rightsquigarrow T. \text{exp1}$

( $:=$ ,  $T. \text{exp1}$ ,  $\text{id}$ )

{ quadruplets de ( $\text{exp2}$ )  $\rightsquigarrow T. \text{exp2}$

test  $\rightarrow (\text{BG}, \text{Fin}, \text{id}, T. \text{exp2}) *$

{ quadruplets de ( $\text{exp3}$ )  $\rightsquigarrow T. \text{exp3}$

\* { quad (inst)

(+,  $\text{id}$ , temps,  $\text{id}$ )

(BR, test, , )

fin  $\rightarrow$

La grammaire postfixe transformée

$\langle \text{inst-For} \rangle \rightarrow \text{for } \text{id} \langle A \rangle := \langle \text{exp1} \rangle \langle B \rangle \text{to} \langle \text{exp2} \rangle \langle C \rangle \text{step} \langle \text{exp3} \rangle \langle D \rangle \text{Do} \langle \text{inst} \rangle$

$\langle E \rangle \langle A \rangle \langle B \rangle \langle C \rangle \langle D \rangle \langle E' \rangle \rightarrow \epsilon$

Les routines sémantique :

Routine (A)

Début

Lookup ( $\text{id\_nom}$ , p)

Si  $p.\text{declare} = 0$  Alors écrire ("erreur id nom de declare")

Si non Si  $p.\text{type} <> \text{entier}$

| Alors écrire (erreur de type)

Fsi

Fsi.

Routine (B)

/\* Vérification de type et génération de l'affectation \*/

Début

Si  $(\text{exp1}).\text{type} <> \text{entiers}$

| Alors écrire (erreur de type)

Si non

Quad( $\text{Qc}$ ) : ( $:=, (\text{exp1}).\text{temp}$ ,  $\text{id\_nom}$ )

Fsi  $\text{Qc} := \text{Qc} + 1;$

Fin

Routine (C)

/\* vérification du type de  $\text{exp2}$  \*/

Si  $(\text{exp2}).\text{type} <> \text{entier}$  Alors écrire ('erreur de type');

Fsi

Fin



# Traitement des branchements non inconditionnel GotoEtip:

1<sup>er</sup> Cas

{  
etip <insts>

{  
Goto etip

2<sup>eme</sup> Cas

{  
goto etip

{  
goto etip

{  
etip <insts>

Une étiquette peut être déclarer, référencier et utiliser en 2 cas.

Organisation de la TS dans le cas des étiquettes

nom	etique	declarer ou non	Defini ou non	@ ou non	adresse correspondante au quadruples de l'instr etip
-----	--------	-----------------------	---------------------	-------------	--

On dispose ~~deux~~ de deux fonctions

look up (nom etip) permet la recherche de l'étiquete dans la TS

[Avec P: Position dans la TS]

Insert (nom etip) Permet l'insertion de l'entité dans la TS

Il existe deux méthode de transaction des étiquette en forme intermédiaire.

1<sup>er</sup> méthode

BRL P

Elle consiste à traduire goto etip en "h"

BRL  $\rightarrow$  P (Position de l'étip dans la-table des symboles)

2 cas se présente

. Si l'étip a été définit avant la référence on peut poser directement un BR P. adresse.

. L'étip est non défini au moment de la référence on génère un BRL P qui sera traduit lors d'un 2<sup>ème</sup> passage en BR P. adresse.

2<sup>eme</sup> méthode méthode de chaînage des references.

P. adresse e = 0 (contient le dernier élément avant étip)

Goto etip

10 (BR, 0)

Goto etip

20 (BR, 10)

Goto etip

30 (BR, 20)

120

Schéma de traduction :

La grammaire "cas ascendant"

<Decl-étip>  $\rightarrow$  Label <list-étip> ; ①

<list-étip>  $\rightarrow$  <list-étip \* / étip \*

<Ref-étip>  $\rightarrow$  Goto étip ②

$\langle \text{inst\_etip} \rangle \rightarrow \text{etip } \textcircled{3} : \langle \text{inst} \rangle$   
 $\langle \text{inst\_etip} \rangle \rightarrow \langle A \rangle : \text{inst}$   
 $\langle A \rangle \rightarrow \text{etip } \textcircled{3}$ 
grammaire transformé

Routine ① /\* Chercher l'étiquette dans la table de symbole \*/

Début

Lookup(nom-etip, P)

Si p.declare = 1

alors écrire ("erreur double déclaration");

Sinon

P.declare = 1 ;

P.adresse = 0 ;

P.utilise = 0 ;

Fin fsi

Routine ② /\* Générer branchemet BR \*/

Début

Lookup(etip.nom, P) ;

Si P.declare = 0

alors ('Erreur etip non déclaré');

Sinon Si P.utilise = 1

alors quad(Qc) = (BR, P.adresse, , )

Qc = Qc + 1 ;

Sinon Quad(Qc) = (BR, P.adresse, , )

P.adresse = Qc ;

Qc ++ ;

fsi

fsi

Fin

Routine ③ (n.a.j.. des quad BR)

Début

Lookup(etip.nom, P) ;

Si P.declare = 0 alors écrire ("Erreur: etip non déclaré");

Sinon p.utilise = 1, x = P.adresse ;

Tant que x < > 0

y := quad(P.adresse, 2)

quad(P.adresse, 2) = Qc ;

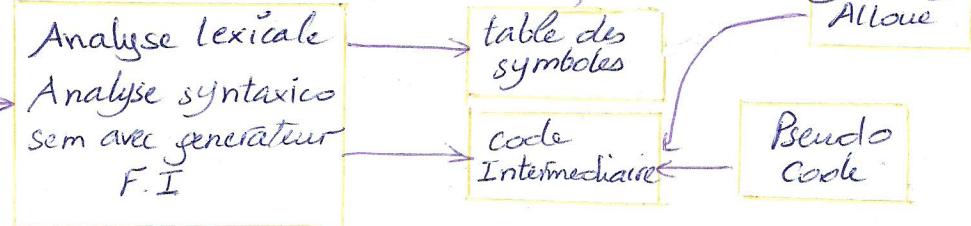
Fait x = y ;

P.adresse = Qc ;

### Chapitre III Allocation Substitution

#### Introduction

Programme source



L'analyse syntaxico-émantique a permis d'engendrer deux textes

Le dictionnaire qui contient les noms des objets manipulés par le programme

Le texte intermédiaire correspondant aux instructions du programme.

L'allocation a pour objet d'attribuer une adresse mémoire, ou d'une façon général un mécanisme d'adressage aux objets que le programme manipule

L'allocation opère sur la table des symboles, et produit la table de symbole alloué où à chaque objet est associé une information d'adressage.

L'allocation peut effectuer certaine vérification syntaxique (cohérence des déclaration des variables)

La substitution a pour objet de remplacer dans les instructions chaque occurrence d'un objet par son adresse, ou encore par le mécanisme d'évaluation de son adresse

La substitution utilise le code intermédiaire et le dico à l'ouverture qui est produit par l'allocation

Elle produit un pseudo code qui est utilisé par la suite dans la génération de code **Variable simple**

A une variable on associe en général du point de vue allocation une longueur et un cadrage. La longueur d'une variable est le nombre de cellules mémoire servant à sa représentation. Cette longueur peut être variable ou constante. Le cadrage d'une variable est une notion que beaucoup d'ordinateurs obligent à introduire pour qu'une variable soit directement utilisable pour un calcul.

Ainsi nous nous avons pour chaque variable deux informations : sa longueur et son cadrage. Pour procéder à la location de mémoire puisqu'il s'agit d'entité indépendante on peut regrouper les variables en fonction de leur longueurs et leur cadrage de façon à perdre le moins d'espace mémoire.

possible

Les éléments de longueurs variables posent un problème quant à leurs présentations.

Les schémas les plus réalistes dans les langages de programmation précise que la longueur d'un élément est variable mais ne peut pas dépasser une certaine valeur, c'est à dire la valeur maximum. En fonction de cette longueur maximale se pose le problème de l'implémentation de ces éléments.

La première méthode consiste à allouer systématiquement la longueur maximale et la deuxième y a des méthodes qui consistent à allouer uniquement la place nécessaire. Cette méthode pose le problème de la gestion de la mémoire.

Groulement homogène On entend sous groupement au niveau du langage d'objet de nature et logique identique.

On peut distinguer le cas selon la dimension du groupement.  
Dimension du groupement Elle est connue à la compilation.  
Exemple : Langage à allocation statique ~~FORTRAN~~

Dimension du groupement non connue à la compilation  
comme (ALGOL).

L'adressage d'un élément d'un vecteur ne pose aucun problème puisque les éléments d'un vecteur sont stockés séquentiellement en mémoire.

On utilise les bornes par le calcul de l'adresse de l'elt et le test de validité d'indice.

Le problème devient compliqué dans le cas des tableaux à plusieurs dimensions. Lorsque les bornes sont connues à la compilation, on peut directement construire le mécanisme d'adressage.

Soit si un vecteur ou une matrice par contre dans le cas où les bornes ne sont pas connues. Il s'agit d'implémenter une routine qui fera l'allocation mémoire à l'exécution. Avant tout on doit choisir le mode de représentation de tableaux en mémoire. On parle alors de rangement ligne par ligne ou colonne/colonnes.

Soit le tableau  $M[2; 1, 1:3]$

Donner le rangement du tableau M ligne/ligne.

$M[-2, 1], M[-2, 2], M[-2, 3]$

$M[-1, 1], M[-1, 2], M[-1, 3]$

$M[0, 1], M[0, 2], M[0, 3]$

$M[1, 1], M[1, 2], M[1, 3]$

$$\begin{matrix} & 1 & 2 & 3 \\ -2 & | & & \\ -1 & | & & \\ 0 & | & & \\ 1 & | & & \end{matrix}$$

On remarque que l'indice 2 qui varie plus vite.

$$@ M[i, j] := @ base + [(i - u_1) * (L_2 - U_2 + 1) + (j - U_2)] \quad \text{taille d'elt.}$$

$$@ M[0, 3] := @ base + [(-2 - (-2)) * (4 - 1 + 1) + (3 - 1)] \quad \text{taille d'elt.}$$

$$= @ base + 2 * 4 + 2$$

$$= @ base + 10$$

Table à 3 Dimensions :

$$@ M[i_1, i_2, i_3] := @ base + [(i_1 - u_1) * (L_2 - U_2 + 1) * (L_3 - U_3 + 1) + (i_2 - U_2) * (L_3 - U_3 + 1) + (i_3 - U_3)] \quad \text{taille d'elt}$$

	U_2				L_2
U_1	0	1	2	3	4
1	x	x	x	x	x
2	x	x	x	x	x
3	x	x	x	x	x

génération à plusieurs dimension

$$T[U_1: L_1, U_2: L_2, \dots, U_n: L_n]$$

$$@ T(i_1, i_2, \dots, i_n) := @ base + [(i_1 - u_1) * (L_2 - U_2 + 1) * (L_3 - U_3 + 1) * \dots * (L_n - U_n + 1) + (i_2 - U_2) * (L_3 - U_3 + 1) * \dots * (L_n - U_n + 1) + \dots + (i_n - U_n)] * \text{taille d'elt.}$$

$$\begin{cases} d_j = L_j - U_j + 1 \\ d_{n+1} = 1 \end{cases}$$

$$@ T[i_1, \dots, i_n] = @ base + [(i_1 - u_1) * \prod_{k=2}^n d_k + (i_2 - U_2) * \prod_{k=3}^n d_k + \dots + (i_n - U_n) \underbrace{\prod_{k=j+1}^{n+1} d_k}_{\text{taille d'elt}}]$$

$$= @ base + \left[ \sum_{j=1}^n (i_j - U_j) \prod_{k=j+1}^{n+1} d_k \right] \text{taille d'elt.}$$

$$= @ base + \left( \underbrace{\sum_{j=1}^n i_j \prod_{k=j+1}^{n+1} d_k}_{\text{partie variante var part}} - \underbrace{\sum_{k=j+1}^{n+1} U_j \prod_{k=j+1}^{n+1} d_k}_{\text{partie constante const part}} \right) \text{taille d'elt.}$$

$$\text{partie variante } \left( \sum_{j=1}^n i_j \prod_{k=j+1}^{n+1} d_k \right)$$

$$\text{partie constante } \left( \sum_{j=1}^n U_j \prod_{k=j+1}^{n+1} d_k \right) \text{taille d'elt}$$

A fin de référencer 1 elt de tableau nous avons besoin de vérifier que les indices appartiennent bien au domaine autorisé, il est donc nécessaire de mémoriser les informations  $U_i, L_i, d_i, n, u_i > l_i$  dans un vecteur de renseignement

Vecteur de renseignement on le trouve dans la table des symboles.

Dans le cas de l'langage @ base c'est l'@ du  
à allocation statique 1<sup>er</sup> elt du tableau.

Les bornes sont comme à la compilation ;  
on peut engendré le code correspondant au  
calcul de l'@ d'un élément.

Dans le cas de l'allocatoin dynamique

les bornes sont inconnu, elle peuvent

varier d'une exécution à une autre, le compilateur intervenir

vector de renseigné

pour 1. générer le code permettant le calcul des paramètres  
d'allocation  $u_i, l_i, t_i$  et la taille max (m)

2. insérer dans le code la macro instruction d'allocation  
alloc (taille, adr. base)

Soit  $T[u_1 : l_1; u_2 : l_2, \dots, u_n : l_n]$

Debut

$i = 1;$  /\* nombre de dimension du tableau \*/

taille := 1;

const part := 0;

pour chaque dimension i du tableau  
faire

Calculer ( $u_i$ );

Calculer ( $l_i$ );

Si ( $u_i > l_i$ ) alors ("erreur : borne inférieur > borne sup")  
Sinon calculer ( $d_i$ );

const part := const part \*  $d_i + u_i$ ;

taille := taille \*  $d_i$ ;

$i = i + 1$ ;

Fait Fsi

taille := taille \* taille d'i elt;

const part := - const part;

alloc (taille, adr. base);

Fin

n	
u <sub>1</sub>	
d <sub>1</sub>	
⋮	
u <sub>n</sub>	
d <sub>n</sub>	
const part	
@ base	q

@ relative au 1<sup>er</sup>  
elt du Tableau dans  
la zone de donnée

Représentation colonne / colonne :

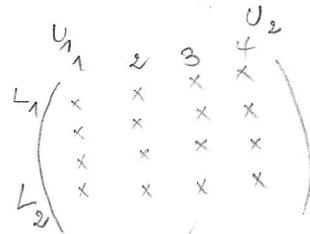
$n=2$  @  $M[i_1, i_2] := @base * [(i_2 - u_2) * (L_1 - V_1 + 1) + (i_1 - U_1)] * \text{taille d'1 elt.}$

Exple : Soit  $M[-2:1, 1:3]$

$$M[-\infty, 1] \quad M[-1, 1] \quad M[0, 1] \quad M[1, \infty]$$

$$M[-2, 2] \quad M[-1, 2] \quad M[0, 2] \quad M[1, 2]$$

$$M[-2, 3] \quad M[-1, 3] \quad M[0, 3] \quad M[1, 3]$$



l'indice qui varie le moins vite est l'indice colonne.

## Généralisation

$\text{@ } N[i_1, i_2, \dots, i_n] := \text{@base} + (i_n - U_n) * (L_{n-1} - U_{n-1} + 1) + \dots + (L_1 - U_1 + 1)$

$$+ (i_{n-1} - U_{n-1}) * (L_{n-2} - U_{n-2} + 1) + \dots + (L_1 - U_1 + 1)$$

$$\vdots$$

$$+ (i_1 - U_1)] * \text{taille d'1 elt}$$

$\text{@ } M[i_1, \dots, i_n] := \text{@base} + (i_1 - v_n) * d_{n-1} * \dots * d_2 * d_1$

$$@ M[i_1, \dots, i_n] := @base + \sum_{j=1}^n (i_j - U_j) \prod_{k=0}^{j-1} j_k$$

Cond  
d<sub>2</sub>  
L<sub>2</sub>  
U<sub>2</sub>  
d<sub>1</sub>  
L<sub>1</sub>  
U<sub>1</sub>

VR  
T.S

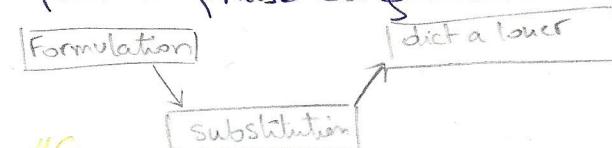
$$+ \sum_{j=1}^n i_j \underbrace{\prod_{k=0}^{j-1} d_k - U_j}_{\text{partic variant}} \underbrace{\prod_{k=j}^{j-1} d_k}_{\text{const part}} L_k - U_k + 1$$

partie valante const part

Représentation des tableaux creux: On ne représente en mémoire que les éléments non nuls, si le tableau possède une symétrie on peut définir une fonction spécifique d'accès à ce tableau. Par contre si on a un tableau creux non symétrique, on peut utiliser un vecteur associé pour décrire l'absence ou la présence d'élément dans un tableau, le vecteur associé est considéré comme une suite de bits, un bit à 1 spécifiant que l'elt associé est présent.

Groupement hétérogène tout groupement au niveau du langage objet de nature différente. Dans un groupement hétérogène on doit allouer les éléments dans l'ordre qui a été spécifié par le programmeur. On peut pas reorganiser les élts de façon à minimiser les pertes de places dû au cadrage. lors de l'allocation on doit disposer d'une description du groupement la longueur, le nombre d'apparition...

Substitution l'objet de la substitution est de remplacer dans les instructions chaque occurrence d'un objet par son adresse, ou encore le mécanisme d'évaluation de son adresse. Donc la phase de substitution opère sur le texte intermédiaire et sur le dictionnaire à l'ouvr. pour créer un pseudo code qui sert de texte d'entrée pour la phase de génération



# Organisation des données à l'exécution Chapitre 4:

- Statique
- Dynamique
- en liste
- Paramétrée

Organisation Statique : comme fortran, Html, ...

Équivalence On peut utiliser même espace mémoire pour plusieurs variable : équivalence  $A, B, C \equiv @A = @B = @C$ .

Common pour partager les variables entre le programme principale et les procédure

Prog Prin: common A,B,C

Prog Sous common F,G,K

A / F
B / G
C / K

Un programme FORTRAN

PP

STOP

END

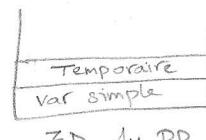
SP1

RETURN

END

Définition d'une zone de données :

ZD d'un PP



ZD du PP

ZD du common