HEB - ESI

COBOL

2ème Gestion

Support de cours

M. CODUTTI
Marco.Codutti@gmail.com

version de juin 2001

Versions du document

- septembre 2000 – première version publique – complet à 85 %
- décembre 2000 – complet à 90 %
  - quelques nouvelles fiches
  - mise à jour de nombreuses fiches
  - nombreuses corrections typographiques
- mars 2001 – complet à 95 %
  - corrections des fiches concernant les fichiers et les sous-programmes
  - informations sur V/SAM
- juin 2001 – complet à 97 %
  - ajout fiche INSPECT et complétion fiche SEARCH
  - corrections typographiques

Avant-propos

Ce document est destiné aux étudiants suivant les cours de COBOL à l'ESI; Il ne prétend pas couvrir tous les aspects de COBOL ni même être exempt d'erreurs; toutefois, il couvre l'essentiel de la matière vue aux cours de 2ème et 3ème sauf les aspects trop fortement liés à l'AS/400. Je me suis inspiré du bon manuel de A. Detaille; qu'il en soit remercié. Il est à remarquer que ce manuel, plus pédagogique dans sa forme, reste d'actualité.

Bibliographie

- A. Detaille, *COBOL – 2ème année Informatique – 1ère partie*, ESI, Notes de cours
- A. Clarinval, *Comprendre, connaître et maîtriser le COBOL*, Presses Universitaires de Namur, 1981, 1ère édition
- *VS COBOL II, Application Programming Language Reference*, Release 4, IBM Press, Mars 1993
- *AS/400 Languages: Systems Application Architecture AD/Cycle COBOL/400 Reference*, Version 2, IBM Press, Septembre 1992
- Lawrence R. Newcomer, *Programmation en COBOL structuré. Théorie et applications,* Série Schaum, 1986
- Mo Budlong, Le programmeur COBOL, S&SM, 1998

Conventions d'écriture

Pour décrire le format des instructions nous utiliserons le méta-langage suivant

- Un mot en majuscule doit être tapé tel quel
- S'il n'est pas souligné, il est facultatif (sa présence ne modifie pas le sens)
- Les crochets indiquent un groupe facultatif
- Les accolades indiquent un choix entre plusieurs possibilités
- Les … indiquent une répétition facultative de ce qui précède
- Un nom en minuscule doit être remplacé par une valeur appropriée
  - id : identificateur
  - lit : littéral
  - val : identificateur ou littéral (pas d'expression)
  - expr : expression (peut se réduire à val)
  - cond : condition
  - proc : procédure imbriquée (suite de propositions ne formant pas une phrase)
  - par : nom de paragraphe ou de section

Table des matières

– Partie I –
Introduction

On introduit ici les notions générales du langage COBOL : Historique, structure du programme, mise en page, …

1. HISTORIQUE

- COBOL (COmmon Business Oriented Language)
- Langage de 2ème génération (comme FORTRAN) né dans les années '60.
- Destiné à la gestion informatique de problèmes commerciaux
- Standardisation par l'ANSI (American National Standard Institute)
  - COBOL 68 : première norme

COBOL 74 : heure de gloire

  - COBOL 85 : norme suivie sur le mainframe et que l'on va étudier ici.
  - OO-COBOL 97 : COBOL orienté objets + autres nouveautés
- Caractéristiques
  - Lisibilité : syntaxe proche de l'anglais. L'idée est qu'il puisse être lu par un commercial (mais faut pas rêver). Cela fait néanmoins qu'il est très verbeux.
  - Portable : Devrait pouvoir tourner sur différentes machines sans modification. En pratique, les modifications sont localisées à l'`ENVIRONMENT DIVISION`.
- Le standard divise le langage en modules, chacun avec différents niveaux. Chaque compilateur implémente un certain niveau d'un module.
  Ex: sur MVS, on a noyau (2/2), relatifs (2/2), debug (1/2), report (0/2)
- La plupart des compilateurs se permettent des extensions par rapport à la norme. Nous verrons les plus importantes introduites par IBM sur les systèmes MVS et AS/400.

2. Petit programme COBOL

Exemple de programme COBOL

```
01
02 IDENTIFICATION DIVISION.
03 PROGRAM-ID. EX.
04
05* ======================================================
06* Lit un fichier et affiche les enregistrements, un à un
07* ======================================================
08
09 ENVIRONMENT DIVISION.
10
11 CONFIGURATION SECTION.
12 SOURCE-COMPUTER. DD.
13 OBJECT-COMPUTER. DD.
14
15 INPUT-OUTPUT SECTION.
16 FILE-CONTROL.
17     SELECT FICHIER-CLIENT ASSIGN TO CLIENTS.
18
19 DATA DIVISION.
20
21 FILE SECTION.
22 FD FICHIER-CLIENT.
23 01 ENR-CLIENT.
24    03 NUMERO PIC 9(6).
25    03 NOM    PIC X(20).
26    03 PRENOM PIC X(20).
27
28 WORKING-STORAGE SECTION.
29 77 EOF PIC X VALUE "0".
30
31 PROCEDURE DIVISION.
32 0-LISTE-CLIENTS.
33     OPEN INPUT FICHIER-CLIENT
34     READ FICHIER-CLIENT
35          AT END MOVE "1" TO EOF
36     END-READ
37     PERFORM 1-TRAITER-CLIENT UNTIL EOF = "1"
38     CLOSE FICHIER-CLIENT
39     STOP RUN
40     .
41
42 1-TRAITER-CLIENT.
43     DISPLAY ENR-CLIENT
44     READ FICHIER-CLIENT
45          AT END MOVE "1" TO EOF
46     END-READ
47     .
```

3. STRUCTURE DU LANGAGE

3.1. STRUCTURE

- Un programme

4 divisions	nom réservé + DIVISION
sections	nom réservé ou libre + SECTION
paragraphes	nom réservé ou libre
phrases	terminé par un point
propositions	instruction
mots	unité du langage
caractères	

Exemple : structure du programme type

Si on examine la structure du programme de la fiche précédente, on trouve la structure suivante :

2-3 : division
    3 : paragraphe + phrase/proposition/mot
9-17 : division
    11-13 : section
        12,13 : paragraphe + phrase/proposition/mot
    15-17 : section
        16,17 : paragraphe + phrase/proposition/mot
19-29 : division
    21-26 : section
        22,23,…,26 : phrase/proposition
    28-29 : section
        29 : phrase/proposition
32-47 : division
    32-40 : paragraphe
        33-40 : phrase
        33,34,…39 : proposition
    42-47 : paragraphe
        43-47 : phrase
        43,44,45,46 : proposition

3.2. DIVISION

Un programme COBOL est composé de 4 divisions au nom et au rôle bien définis. Dans la norme ANSI 85, une partie vide peut-être omise.

- `IDENTIFICATION` : cette partie a diminué au fil des normes et ne contient plus que le nom du programme.
- `ENVIRONMENT` : lien avec le système (matériel, fichiers, OS, …)
- `DATA` : description des données
- `PROCEDURE` : description du traitement des données.

3.3. SECTIONS ET PARAGRAPHES

- Avec un nom réservé pour indiquer le but (sauf dans la procédure division où le nom est libre)

3.4. Phrases et propositions

- Une phrase est une suite de propositions terminée par un point. Souvent, elle contient une seule proposition.
- Une proposition recoupe presque parfaitement le concept classique d'instruction. Cela est plus subtil pour les instructions complexes contenant des clauses d'exceptions et pour les structures de contrôle.

3.5. Caractères

- A-Z a-z 0-9 + - * / ( ) < > = $ " ; , .
- Pas de différence minuscule/majuscule
- Les autres caractères du système peuvent apparaître dans les chaînes de caractères et les commentaires.

3.6. Noms

- Les noms servent à nommer les variables, les paragraphes, …
- Suite de 1 à 30 caractères : lettre, chiffre et –
- Ne peux pas commencer ou terminer par un tiret ni contenir 2 tirets consécutifs

Exemple : noms corrects et incorrects	
correct	pas correct
ABC	-ABC
235	A*T
64C	AB.D
MOVE	(BROL)
A14-UYT	AB--CD

- **Nom réservé :** Nom ayant un sens dans le langage (noms d'instructions, …)
  Exemple : `MOVE, DATE`[1]`, IF`
- **Identificateur (variable)** : nom non réservé avec au moins 1 lettre
  Exemple : 1A est correct
- **Nom de procédure (paragraphe, section)** : nom non réservé sans autre restriction
  Exemple : 123 est correct
- **Convention** : Il est important de bien choisir un nom de variable ou de paragraphe. Le nom doit expliciter clairement le rôle de la variable.
  - Utiliser des noms longs et explicites
  - Séparer les mots par des tirets

    TOTSAL → TOTAL-SALAIRE
    I → NUMERO-CLIENT
    TPSMOY → TEMPS-MOYEN
    TRTCLI → TRAITER-CLIENT

[1] Attention : certains mots clés ont un sens en Français et on aura tendance à les utiliser comme nom de variable (DATE est un exemple typique). A éviter!

4. Mise en page d'un programme

Quelques règles simples doivent être suivies dans la mise en page d'un programme COBOL.
La ligne est divisée en différentes parties

- Colonnes 1 à 6 : réservé pour le numéro de ligne.
  Ce numéro n'est plus obligatoire et suppléé automatiquement par le compilateur.
- Colonne 7 : statut de la ligne (décrit ci-dessous)
- Colonnes 8 à 11 : zone A
- Colonnes 12 à 72 : zone B
- Colonnes 73 à 80 : zone libre de commentaire

4.1. Zone A

Certaines instructions COBOL doivent impérativement commencer en zone A.
Essentiellement, tout ce qui concerne la structure du programme. Plus précisément

- Les noms de divisions, de sections de paragraphes
- Les clauses FD et SD
- Les descriptions de variables de niveau 01 et 77

4.2. Zone B

Certaines instructions peuvent commencer en zone A ou B :

- les descriptions de variables de niveau autre que 01 ou 77

Toutes les autres lignes COBOL doivent impérativement commencer en zone B.

4.3. Colonne 7

Cette colonne indique le statut de la ligne

- espace : ligne normale de code
- étoile (*) : toute la ligne est un commentaire et n'est pas interprétée par le compilateur.
- barre (/) : idem + saut de page dans le source.
- tiret (-) : ligne de continuation. Toute instruction COBOL peut librement être répartie sur plusieurs lignes si on reste en ligne B et qu'on ne coupe pas un mot. Si on doit couper un mot (le plus souvent une chaîne), il faut l'indiquer par un tiret. La chaîne comprendra tous les caractères jusqu'à la colonne 72 incluse de la première ligne et reprendra à partir d'un guillemet sur la ligne suivante.

Exemple.

```
           1              7
      78   2       …      2
           MOVE "Hello W
      -       "orld" TO A
```

A ← "Hello W orld"

4.4. Utilisation des espaces

Les mots COBOL sont séparés par des espaces. Un espace a exactement la même signification qu'un ensemble d'espaces.

4.5. Utilisation du point

Il y a 3 signes de ponctuations en COBOL (, ; .). Seul le point est significatif; il indique une fin de phrase. Il doit se placer directement après le mot le précédant, sans espace et être suivi d'un espace. Il est obligatoire après: un nom de division, section ou paragraphe, en fin de paragraphe et phrase (il peut donc modifier le sens d'une suite d'instructions!)

– Partie II –
Identification et Environment Division

On voit ici tout ce qui concerne les deux premières divisions d'un programme COBOL.

5. Identification et Environment Division

5.1. Identification division

Cette division, qui sert à identifier le programme, a vu son rôle diminuer.
De manière obligatoire, on ne retrouve plus que

```
IDENTIFICATION DIVISION.
PROGRAM-ID. nom-du-programme.
```

5.2. Environment division

Cette division introduit le lien entre le programme et son environnement.

```
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
[SOURCE-COMPUTER …]
[OBJECT-COMPUTER …]
[SPECIAL-NAMES].
INPUT-OUPUT SECTION.
(description des fichiers)
```

- `SOURCE-COMPUTER` et `OBJECT-COMPUTER` permettent d'indiquer l'OS où on compile et l'OS pour lequel doit être construit l'exécutable (le même, la plupart du temps). Optionnel.
- `SPECIAL-NAMES` permet de configurer le fonctionnement de COBOL. Il est décrit dans une fiche à part.
- `INPUT-OUTPUT SECTION` contient essentiellement les clauses `SELECT` qui permettent de décrire les fichiers externes à utiliser.

6. SPECIAL-NAMES

On peut ici donner quelques directives configurant COBOL. Tout doit figurer dans une seule phrase; attention donc à ne pas terminer les clauses par un point.

6.1. Classes

Permet de donner un nom à un ensemble de caractères. Ce nom sera utilisé dans les tests pour vérifier qu'une variable non numérique ne contient que les caractères requis.

```
CLASS nom IS lit1 [THRU lit2] …
```

- sans `THRU`, `lit1` peut contenir plus d'un caractère et tous les caractères sont intégrés à la classe
- avec `THRU`, `lit1` et `lit2` sont formés d'un seul caractère et tout l'intervalle est intégré à la classe. (pas de problème si `lit2` < `lit1`, on prend tout)
- On peut combiner à loisir ces clauses
- Pour tester si une chaîne appartient à la classe, on indique

```
id IS nom-classe
```

`id` ne peut pas être un littéral

6.2. Européanisation

Deux clauses pour indiquer le signe monétaire et l'utilisation de la virgule pour les nombres non entiers

Signe monétaire

```
CURRENCY SIGN IS lit
```

`lit` est un seul caractère qui ne doit pas prêter confusion avec les autres caractères que l'on peut trouver dans une clause PIC.

Exemple – Signe monétaire

```
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SPECIAL-NAMES.
    CURRENCY IS "F".

77 A PIC 99F.

    MOVE 13 TO A
    DISPLAY A      (affiche 13F)
```

Signe décimal

```
DECIMAL-POINT IS COMMA
```

Exemple – Signe décimal

```
DECIMAL-POINT IS COMMA

77 A PIC 99,9.
```

La virgule doit être utilisée à la place du point et apparaîtra comme une virgule sur les outputs.

– Partie III –
Data Division

On étudie ici en détail ce qui compose la DATA DIVISION. On reporte à des parties suivantes la déclaration de tables et de fichiers.

7. DATA DIVISION

Cette division est dédiée à la déclaration des variables. Il y a 3 sections.

```
DATA DIVISION.
FILE SECTION.
WORKING-STORAGE SECTION.
LINKAGE SECTION.
```

7.1. FILE SECTION.

Définit le structure des enregistrements se trouvant dans les fichiers. Il n'y a aucune allocation mémoire mais uniquement la définition de masques pour lire les tampons de fichiers.

```
FD nom-fichier.
01 enr1.
    …
[01 enr2.
    …]
```

- nom-fichier est le nom interne à COBOL du fichier (cf. clause SELECT).
- Chaque enregistrement concerne donc la même zone mémoire.

7.2. WORKING-STORAGE SECTION.

C'est ici qu'on définit toutes les variables La syntaxe générale de définition d'une variable est

```
niveau nom-variable [PIC image] [VALUE const] [USAGE …]
```

plus encore bien d 'autres options qui sont vues dans des fiches à part

7.3. LINKAGE SECTION.

Cette section est dédiée à la description des variables qui vont être échangées entre programme. Cette notion est liée à l'appel de sous-programme (CALL) et est examinée dans une partie séparée.

8. Littéraux et Constantes

8.1. Littéraux numériques

Format simple (il existe également un format scientifique)

- Suite de chiffres avec utilisation du point décimal et du signe
- Maximum 18 chiffres
- Le point ne peut pas être final (mais il peut commencer le nombre)
- Le signe est placé en tête

8.2. Littéraux alphanumériques

- N'importe quels caractères inclus dans des apostrophes.
- Pour indiquer une apostrophe, on la dédouble.
- Taille maximale, 160 caractères

8.3. Constantes figuratives

Noms représentant des littéraux. Chaque constante représente une suite infinie du caractère spécifié. La taille réellement utilisée dépendra du contexte.

- `ZERO`, `ZERO(E)S` : numérique 0 ou suite de caractères 0, en fonction du contexte
- `SPACE(S)` : suite d'espaces
- `HIGH-VALUE(S)` : suite du caractère de code le plus élevé. (FF)
- `LOW-VALUE(S)` : suite du caractère de code le plus bas (00)
- `QUOTE(S)` : suite d'apostrophes.
- `ALL littéral` : suite d'occurrence de la chaîne de caractères spécifiée.
- Une expression comme `ALL SPACES` est redondante.

9. EXPRESSION ARITHMÉTIQUE

Les opérateurs intervenant dans une expression arithmétique sont, dans l'ordre de priorité d'évaluation:

- - + moins et plus unaires
- ** exposant
- * / multiplication et division
- - + soustraction et addition

COBOL impose des règles strictes quant à l'écriture des expressions afin de lever certaines ambiguïtés.

- Les opérateurs sont entourés d'un espace ou d'une parenthèse. Le non respect de cette règle provoque des avertissements à la compilation

Exemple – Expressions correctes et incorrectes
`A - B` est correct
`A-B` est incorrect; il ne peut distinguer la soustraction d'une variable ayant ce nom

L'utilisation d'expressions complexes étant fortement limitée dans bon nombre de contextes, les expressions les plus générales se rencontreront presque exclusivement dans l'ordre `COMPUTE`.

10. Expression logique

10.1. Comparaison

La comparaison sera numérique si les 2 opérandes sont numériques. Un opérande numérique édité entraînera une comparaison non numérique. Les opérateurs sont

```
[IS] [NOT] {GREATER THAN, >}
[IS] [NOT] {LESS THAN, <}
[IS] [NOT] {EQUAL TO =}
[IS] [NOT] {GREATER THAN OR EQUAL TO, >=}
[IS] [NOT] {LESS THAN OR EQUAL TO, >=}
```

10.2. Test de signe

Uniquement pour les numériques

```
val [IS] [NOT] {POSITIVE, NEGATIVE, ZERO}
```

10.3. Condition complexe

On peut combiner les résultats avec les opérateurs logiques NOT, AND et OR, dans cet ordre de priorité. Les opérandes sont évalués de gauche à droite et l'évaluation est arrêtée dès que le résultat peut être déterminé.

10.4. Simplification ecriture

On peut simplifier l'écriture d'expressions logiques complexes en

1. omettant le membre de gauche d'une relation : il est alors pris égal au précédent membre de gauche.
2. ommetant également la relation : elle est alors prise identique à la relation précédente.

Simplification d'expressions logiques
NOT(A = B AND NOT = C AND NOT D) ***est une abbréviation pour***
NOT(A = B AND A NOT = C AND NOT A NOT = D)

10.5. Nom de condition

cf. la fiche associée

10.6. Nom de classe

Test d'appartenance à un groupe de caractères (cf. description de la clause CLASS)

```
[IS] [NOT] {NUMERIC, ALPHABETIC, ALPHABETIC-LOWER,
            ALPHABETIC-UPPER, nom-de-classe}
```

11. Types de variables

11.1. Variable alphabétique

- Ne peut contenir que des lettres et des espaces
- PIC admis : A
- USAGE admis : DISPLAY
- VALUE peut recevoir tout littéral non numérique ne contenant que des lettres et des espaces

11.2. Variable numérique

- Nombre entier ou non, signés ou non
- PIC admis : 9, S et V
- USAGE admis : DISPLAY, BINARY, COMP, PACKED-DECIMAL, COMP-3, COMP-4 (fort dépendant d'un compilateur à l'autre)
- VALUE peut recevoir tout nombre, et ZERO
- Peuvent intervenir comme opérande d'un calcul
- Attention : Pour certains compilateurs, on ne peut pas assigner une valeur négative si pas de signe dans la définition. Pour d'autres, le signe est perdu sans message d'erreur.

11.3. Variable numérique éditée

- PIC admis : 9, S et V, B, Z, 0, /, ',', '.', +, - , CR, DB, *, $
- USAGE admis : DISPLAY
- VALUE doit recevoir une chaîne représentant exactement le contenu mémoire (pas d'édition à partir de cette valeur)
- Ne peuvent pas être utilisées comme opérande dans un calcul (mais peut recevoir le résultat d'un calcul)

Exemple – Variables éditées numériques et la clause VALUE	
`PIC +ZZ9 VALUE "+ 21"`	*OK*
`PIC +ZZ9 VALUE 21`	*Non*
`PIC +ZZ9 VALUE "21"`	*Dépend du compilateur (Fujitsu Oui. MVS : Non)*

11.4. Variable alphanumérique

- PIC admis : X, A, 9 (au moins un X, les A et 9 sont considérés comme des X)
- USAGE admis : DISPLAY

Exemple – Variables alphanumériques et la clause VALUE	
`PIC XA9 VALUE "ABC"`	OK

11.5. Variable éditée

- PIC admis : X, A, 9, B, 0, / (au moins un X ou A et au moins un des caractères d'édition)
- USAGE admis : DISPLAY
- VALUE doit recevoir une chaîne représentant exactement le contenu mémoire (pas d'édition à partir de cette valeur)

Exemple – Variables éditées non numériques et la clause VALUE	
`PIC XX/XX VALUE "AB/CD"`	*OK*
`PIC XX/XX VALUE "ABCD"`	*compilateur (Fujitsu Oui. MVS : Non)*

12. VARIABLES SIMPLES ET STRUCTURÉES

La notion de variable structurée en COBOL est très proche de la notion classique de structure dans les autres langages. On regroupe plusieurs zones d'informations en une seule zone ayant un nom. On définit cela en donnant des niveaux.

Exemple – Variable structurée

```
01 PERSONNE.
   03 NOM                PIC X(20).
   03 ADRESSE.
      05 RUE              PIC X(20).
      05 NUMERO           PIC 9(4).
   03 DATE-NAISSANCE.
      05 JOUR             PIC 99.
      05 MOIS             PIC 99.
      05 ANNEE            PIC 9999.
```

On a ici plusieurs structures imbriquées. L'allocation mémoire est la suivante

PERSONNE (52 octets)					
	ADRESSE (24 octets)		DATE-NAISSANCE (8 octets)		
NOM	RUE	NUMERO	JOUR	MOIS	ANNEE
20 octets	20 octets	4 octets	2 octets	2 octets	4 octets

- Le niveau supérieur doit porter le niveau 01.
- Les suivants sont compris entre 02 et 49. On peut sauter des numéros.
- 2 frères doivent avoir le même numéro. Ce n'est plus obligatoire pour les cousins (RUE et JOUR ne doivent pas avoir le même numéro).
- Seul les variables élémentaires (dernier niveau) peuvent posséder une clause PIC. Une variables structurée a implicitement une clause PIC X(n) où n est le nombre d'octets occupés par les éléments qui composent la structure.
- Une zone non structurée a le niveau 77 (01 est valable aussi mais non recommandé).
- Il arrive qu'une partie d'une structure ne sera pas utilisée. On peut alors utiliser le mot clé FILLER ou même omettre le nom. On utilisera beaucoup ce principe dans les fichiers d'entrées si certaines informations ne sont pas pertinentes pour notre cas ou encore dans les fichiers de sortie lorsque des séparateurs sont définis.

Exemple – Omission de FILLER

```
01 LIGNE-SORTIE.
   03         PIC XXX value " | ".
   03 nom     PIC X(20).
   03         PIC XXX value " | ".
   03 prenom  PIC X(20).
   03         PIC XXX value " | ".
```

13. NOMMER LES VARIABLES

13.1. NOMS QUALIFIÉS : IN OU OF

On peut, dans un programme, utiliser plusieurs fois le même nom pour désigner des variables pour autant qu'on puisse les distinguer par la structure dans lesquelles elles se trouvent.

Exemple – Noms qualifiés – cas simple

```
01 A.                  01 D.
   02 B.                  02 E.
      03 C.                  03 C.
```

Il existe plusieurs possibilités pour différencier les deux variables C.
C OF B
C OF B OF A (on peut indiquer des qualifications non nécessaires à la distinction)
C OF A (on peut sauter des niveaux dans la hiérarchie)
Les niveaux FD, … peuvent aussi intervenir

Exemple – Noms qualifiés – définition ambiguë

```
01 A.               77 B.
   02 B.
```

Pas valable car B est ambigu. (Multi-defined B).

Exemple – Noms qualifiés – définition ambiguë (2)

```
01 A.                  01 D.
   02 B.                  02 A.
      03 C.                  03 C.
```

Accepté mais C ne pourra pas être désigné par C OF A qui est ambigu (puisqu'on peut sauter des niveaux)

13.2. MODIFICATION DE RÉFÉRENCE

On peut extraire facilement une sous-chaîne

```
id(début:[longueur])
```

`début` indique le premier caractère à prendre (on compte à partir de 1).
`longueur` indique la longueur de la sous-chaîne (jusqu'au bout si non spécifié).
Ce sont 2 littéraux entiers > 0.
On trouve d'abord le nom qualifié puis la modification de référence

```
id OF structure (debut:longueur)
```

Attention: Une partie de variable est toujours considérée comme une chaîne (même si extrait d'un numérique)

14. CLAUSE PIC

Les caractères les plus simples ne posent pas de problèmes. Pour certains, il est parfois difficile (et cela peut varier d'un compilateur à l'autre) de déterminer exactement le résultat obtenu dans les cas les plus tordus.

Dans tous les cas, on peut répéter plusieurs fois un même caractère ou indiquer le nombre d'occurrences entre parenthèses

Exemple – facteur de répétition

```
PIC 999.99 et PIC 9(3).9(2) sont 2 notations équivalentes
```

14.1. CARACTÈRES SIMPLES

A	lettre ou þ
X	tout
9	chiffre 0 à 9
S	signe virtuel (pas d'octet réservé)
V	point décimal virtuel (pas d'octet réservé)

Le signe virtuel utilise le chiffre hexadécimal de poids fort du dernier octet pour représenter le signe, suivant la table suivante

Code	Positif	Négatif
EBCDIC	C	D
ASCII	4	5

Exemple – Signe et point virtuels

```
A PIC S999V9.
MOVE 100  TO A donne "0100"
MOVE 10.3 TO A donne "0103"
MOVE -101 TO A donne "100A"
ADD  1.2  TO A donne "099H"
```

14.2. CARACTÈRES D'ALLÉGEMENT DES 0

Z	Les 0 de tête sont remplacés par des þ. (*1) (*2)
*****	Les 0 de tête sont remplacés par des *. (*1) (*3)

- `PIC 9Z` n'a pas de sens → interdit.

(*1) *On s'arrête normalement au point décimal.*

- `A PIC ZZVZZZ. MOVE 0.003 TO A` donne þþ003
- `A PIC ZZ.ZZZ. MOVE 0.003 TO A` donne þþ.003

(*2) *Si que des Z et valeur 0 → que des espaces (point décimal supprimé)*

- `A PIC ZZVZZZ. MOVE 0 TO A` donne þþþþþ
- `A PIC ZZ.ZZZ. MOVE 0 TO A` donne þþþþþþ

(*3) Si que des * et valeur 0 → que des * (point décimal non supprimé)

- `A PIC **.***. MOVE 0 TO A` donne `**.***`

14.3. Caractères d'insertions

B	b
/	/ (*4)
0	0 (*4)
,	insère une virgule, souvent utilisé pour la virgule séparant les milliers (*4)
$	signe monétaire (en tête ou en fin)
.	point décimal
+	signe (en tête ou en fin)
-	signe – ou espace (en tête ou en fin)
CR	apparaît si nombre négatif, sinon espaces (uniquement en fin)
DB	apparaît si nombre négatif, sinon espaces (uniquement en fin)

- Les zones éditées peuvent juste être assignées (par MOVE ou équivalent); elles ne peuvent pas apparaître comme opérande d'un calcul.

(*4) *N'apparaît pas si le caractère à gauche est un espace.*

- `A PIC ZZ/ZZ. MOVE 13 TO A` donne `bbb13`
- `A PIC ZZ/ZZ. MOVE 123 TO A` donne `b1/23`
- `A PIC B/X. MOVE SPACE TO A` donne `bbb`

14.4. Caractères flottants

- ++…+ Z avec le signe placé le plus à droite possible
- --…- Z avec le signe placé le plus à droite possible (si négatif)
- $$…$ Z avec le signe monétaire placé le plus à droite possible

14.5. Exemples récapitulatifs et difficultés

PIC	Move	Contenu	Remarque
`+***.**`	`-2330`	`-330.00`	Alignement sur le point décimal
`+***.**`	`0`	`****.**`	+ remplacé par * car 0
`$9,99`	`300`	`$3,00`	La virgule n'est pas un point décimal. Pas d'alignement
`$$,99`	`0`	`bb$00`	La virgule n'apparaît pas car 0 non significatif à gauche. $ le plus à gauche possible (prend la place de , ce qui peut surprendre)
`++/++/++`	`-2330`	`bb-23/30`	La première / n'apparaît pas; le signe prend sa place.
`$+(3).++`	`-2330`	`$-30.00`	Le signe est flottant et vient se coller au nombre
`+$(3).$$`	`-2330`	`-$30.00`	La monnaie est flottante et vient se coller au nombre
`999DB`	`230`	`230bb`	DB est remplacé par `bb` si nombre positif
`999DB`	`-230`	`230DB`	id.
`999CR`	`230`	`230bb`	CR est remplacé par `bb` si nombre positif
`999CR`	`-230`	`230CR`	id.

15. CLAUSE VALUE

Cette clause permet de donner une valeur de départ à une variable

- Les variables ont une valeur par défaut mais il est plus prudent de ne pas en ternir compte
- Cette clause n'a aucun sens en FILE SECTION
- Pour une variable éditée, on doit assigner une chaîne de caractère qui sera recopiée telle quelle dans la zone mémoire assignée, sans formatage.

Exemple – Clause VALUE et variables éditées	
`PIC +ZZ9 VALUE "+ 21"`	*OK*
`PIC +ZZ9 VALUE 21`	*Non*
`PIC +ZZ9 VALUE "21"`	*Dépend du compilateur (Fujitsu Oui. MVS : Non)*

- La valeur assignée ne peut pas déborder de la zone assignée. Il n'y a pas de troncature effectuée

Exemple – Clause VALUE et dépassement de capacité	
`PIC 999 VALUE 1234`	*Erreur*
`PIC XXX VALUE "ABCD"`	*Erreur*

16. Clause USAGE

Cette clause permet notamment de définir le format interne de représentation d'une variable numérique; c'est ce que nous étudions ici. Elle est également utilisée pour la déclaration de données index (cf. tableaux).

- Pas pour les niveau 66 et 88
- Si donné pour un groupe, s'applique à tous les éléments du groupe (qui ne peuvent amener une contradiction)

16.1. DISPLAY (par défaut)

- La donnée est représentée sous forme de caractère, 1 par octet.
- Facilite l'écriture mais complique le calcul.
- Lors d'un calcul, la donnée est convertie en format interne, le calcul est effectué puis, si la variable est modifiée, on revient à la forme DISPLAY.
- Chaque chiffre d'un nombre sera représenté par un octet. Le signe sera représenté dans les 4 bits de poids fort du dernier caractère.
  - 0 à 9 : F0 à F9 (en EBCDIC)
  - + : C en poids fort, - : D en poids fort
  - exemple : +1234 est représenté : F1 F2 F3 C4
    (cf. table EBCDIC pour obtenir le caractère équivalent.)

16.2. BINARY

- La donnée numérique est représentée en complément à 2, virgule fixe.
- Le bit de poids fort est utilisé pour représenté le signe.
- Le nombre de bits dépend du nombre de chiffres déclarés dans la clause PIC

chiffres dans PIC	octets en mémoire
1 à 4	2
5 à 9	4
10 à 18	8

16.3. COMP(UTATIONAL)

- Représentation d'un numérique entier. Système dépendant (ici = BINARY)

16.4. COMP(UTATIONAL)-1

- Représentation d'un flottant en simple précision (4 octets)

16.5. COMP(UTATIONAL)-2

- Représentation d'un flottant en double précision (8 octets)

16.6. COMP(UTATIONAL)-3

- Système dépendant (ici = PACKED-DECIMAL)

16.7. COMP(UTATIONAL)-4

- Système dépendant (ici = BINARY)

16.8. PACKED-DECIMAL

- Comme DISPLAY mais on stocke 2 chiffres par octets (puisque 4 bits suffisent)
- Le signe est représenté dans la partie faible du dernier octet.

17. Clause REDEFINES

Permet de redéfinir une zone mémoire en en donnant une autre interprétation.

```
niveau id1 REDEFINES id2 …
```

Doit apparaître directement après le nom de la variable, avant les autres clauses
Indique que `id1` redéfinit la zone occupée par `id2` plutôt que de définir une nouvelle zone.
Doit suivre immédiatement la définition de `id2`.
Le niveau doit être le même.
`id1` ne peut pas contenir de clause `VALUE`.
La longueur de la nouvelle donnée ne peut excéder la zone attribuée pour la première.
Interdit en niveau 1 de la `FILE SECTION`. Inutile puisqu'il y a un `REDEFINES` explicite.

Exemple - Redefines

```
01 annee-x
   03 siecle              PIC 99.
   03 annee-dans-siecle   PIC 99.
01 annee REDFINES annee-x PIC 9999.
```

Ou encore

```
01 annee                  PIC 9999.
01 annee-x REDFINES annee
   03 siecle              PIC 99.
   03 annee-dans-siecle   PIC 99.
```

18. AUTRES CLAUSES

18.1. MISE A BLANC

La clause BLANK WHEN ZERO permet de mettre tout un champ à blanc si la valeur est nulle.

Exemple – Utilisation de BLANK WHEN ZERO

```
A PIC +99.9 BLANK WHEN ZERO.
MOVE 0 TO A (donne "     " et pas "+00.0")
```

18.2. LA CLAUSE JUST(IFIED)

Cette clause, valable pour les variables non numériques indique à cadrage à droite lors des MOVE.

```
JUST[IFIED] RIGHT
```

18.3. LA CLAUSE SYNC(HRONIZED)

Les variables binaires seront synchronisées sur une adresse paire ce qui est parfois nécessaire pour un accès direct via le processeur

```
SYNC { LEFT  }
       RIGHT
```

18.4. LA CLAUSE SIGN

Permet d'indiquer comment va être représenté le signe pour une variable numérique de type DISPLAY.

```
SIGN IS { LEADING }  [SEPARATE]
          TRAILING
```

Le signe sera stocké avec le premier ou le dernier chiffre ou séparément (en tête ou en queue).

19. NOMS DE CONDITIONS

19.1. DÉFINITION

Il est possible d'associer un nom à certaines valeurs d'une variable. Cela permet de clarifier le code.

```
88 nom-condition {VALUE IS|VALUES ARE} lit-1 [THRU lit2]
                                       [lit-3 [THRU lit-4]] …
```

- On peut avoir plusieurs clauses 88.
- Elles doivent suivre immédiatement la déclaration de la variable.
- On peut indiquer des intervalles mais on doit respecter lit-1<lit-2
- On peut définir des conditions aussi bien en file section qu'en working-storage.
- Cela peut être une sous-structure.

Exemple – Expliciter des valeurs

```
77 sexe PIC 9.                77 nationalite PIC X.
   88 MASCULIN VALUE 1.          88 belge    VALUE "B".
   88 FEMININ  VALUE 2.          88 francais VALUE "F".

77 age PIC 999.                    77 PIC X.
   88 junior VALUES  0 THRU 24.       88 eof VALUE 1.
   88 adulte VALUES 25 THRU 59.
   88 senior VALUES 60 THRU 999.
```

19.2. MODIFICATION

Un nom de condition se modifie via la commande `SET`.

```
SET nom-condition TO TRUE
```

- Modifie la variable associée afin que la condition soit vraie
- Si plusieurs valeurs sont associées à une condition, la première est choisie

Exemple – Utilisation du SET

```
READ fichier AT END SET eof TO TRUE END-READ
```

19.3. UTILISATION

Un nom de condition peut apparaître partout où une condition peut apparaître.

Exemple – Lecture d'un fichier

```
77 eof PIC 9.           perform paragraphe until eof=1
```

devient

```
77 PIC 9.               perform paragraphe until eof
   88 eof value 1.
```

Exemple – Simulation d'une variable booléenne

```
77 PIC X.                    SET not-fin TO TRUE
   88 fin     VALUE "1".          IF fin THEN …
   88 not-fin VALUE "0".
```

– Partie IV –
Procédure Division

20.ACCEPT

ACCEPT permet de lire sur le fichier standards d'entrée (stdin, sysin, …) ou, variante, de lire une variable système.

20.1. Lecture sur le fichier d'entrée standard

```
ACCEPT id
```

- Le fichier d'entrée ne doit pas être ouvert ou fermé.
- Chaque lecture concerne une ligne physique entière (considérée comme composée de 80 caractères).
- On indique une variable et pas un nom de ficher comme dans un `READ`.
- Un seul identificateur est permis (souvent un groupe)
- Pas de notion de fin de fichier; d'où la connaissance du nombre d'enregistrement ou l'utilisation d'une valeur sentinelle est nécessaire.
- Si la zone réceptrice est trop grande, il y a remplissage; si elle est trop petite, on tronque.

Exemple – ACCEPT – Padding et troncature

Si le fichier input contient	*le bout de code suivant*	*donne*
`ABCD`	`77 A PIC X(3).`	
`ABC`	`ACCEPT A.`	`A = "ABC"`
`AB`	`ACCEPT A.`	`A = "ABC"`
`A`	`ACCEPT A.`	`A = "AB "`
	`ACCEPT A.`	`A = "A  "`

- Dans la norme, il n'y a aucune conversion de données à la lecture

Exemple – ACCEPT – Pas de conversion de données

Si l' input contient	*le bout de code suivant*	*donne*
`04`	`77 A PIC 99.`	
`4`	`ACCEPT A.`	*A = 04 ce qui est correct*
	`ACCEPT A.`	*A = 4þ ce qui n'est pas un nombre*

20.2. MVS

Sur MVS, le fichier d'entrée standard est SYSIN, déclaré dans la carte JCL par

Exemple – JCL classique pour un SYSIN

```
//GO.SYSIN DD *
valeurs …
/*
```

20.3. Lecture de paramètres système

Cet ordre peut également être utilisé pour lire la date courante

```
ACCEPT id FROM DATE // AAMMJJ
ACCEPT id FROM DAY  // AAJJJ
ACCEPT id FROM TIME // HHMMSSCC
```

21.ADD

Addition

21.1. 1° FORMAT

```
ADD val1 [val2…] TO id3 [ROUNDED] [id4 [ROUNDED]…]
    [ON SIZE ERROR proc1]
    [NOT ON SIZE ERROR proc2]
[END-ADD]
```

- id3 ← id3 + val1 + val2
- id4 ← id4 + val1 + val2
- On peut mélanger les USAGE et les formats
- ROUNDED : ajoute 5 au premier chiffre qui sera perdu (pas forcement un arrondi entier donc!) .

Exemple - Arrondi

```
A PIC 99V9 VALUE 12.2.
ADD 12.46 TO A     (donne 246)
ADD 12.46 TO A ROUNDED  (donne 247)
```

- ON SIZE ERROR : lancé si dépassement de capacité *à gauche*. Les variables posant problème restent inchangées.

Exemple – Dépassement de capacité

```
A PIC 99V9 VALUE 12.2.
ADD 92.46 TO A ON SIZE ERROR PERFORM erreur END-ADD
```

lance la procédure `erreur`.

21.2. 2° FORMAT

```
ADD val1 [val2…] TO val3
    GIVING id4 [ROUNDED] [id5 [ROUNDED]…]
    [ON SIZE ERROR proc1]
    [NOT ON SIZE ERROR proc2]
[END-ADD]
```

- id4 ← val1 + val2 + val3
- id5 ← val1 + val2 + val3

ATTENTION : Sur MVS, les calculs avec des nombres flottants sont toujours arrondis même si la clause ROUNDED n'est pas spécifiée (cf. Réf MVS p186)

22.COMPUTE

Calcul général

```
COMPUTE id1 [ROUNDED] = expr
    [ON SIZE ERROR proc1]
    [NOT ON SIZE ERROR proc2]
[END-COMPUTE]
```

- Combinaison de : -, +, *, /, **

Cf. règles d'écriture d'une expression arithmétique.

- COBOL choisit la précision des variables intermédiaires ce qui peut surprendre

Exemple - Arrondis

```
77 A pic 999 value 1.
77 B pic 999 value 100.
77 C pic 999V999.
COMPUTE C = (A / B) * 100
DISPLAY C
```

donne 1 mais

```
77 A pic 999 value 1.
77 B pic 999 value 100.
77 C pic 999.
COMPUTE C = (A / B) * 100
DISPLAY C
```

donne 0 car le compilateur choisit la précision des calculs intermédiaires aussi en fonction de celle demandéed dans le résultat final.

23.DISPLAY

DISPLAY permet d'écrire sur le fichier standard de sortie.(stdout, sysout, …)

```
DISPLAY val1 [val2 …]
```

- On peut indiquer plusieurs identificateurs qui seront écrits un à la suite de l'autre.
- Si la zone réceptrice est trop grande, il y a padding; si elle est trop petite, on tronque.
- Chaque écriture concerne une ligne physique entière (considérée comme composée de 120 caractères).
- Dans la norme, il n'y a aucune conversion de données à l'écriture

Exemple – ACCEPT – Pas de conversion de données

```
77 A PIC S99 VALUE -15.
DISPLAY A.
```

va écrire

```
1N
```

car le signe est représenté avec le dernier chiffre

- Sur MVS, le fichier de sortie standard est SYSOUT, déclaré dans la carte JCL par

Exemple – JCL classique pour un SYSOUT

```
//GO.SYSOUT DD SYSOUT=Z
```

24.DIVIDE

Division

1.1. 1° FORMAT

```
DIVIDE val1 INTO id2 [ROUNDED] [id3 [ROUNDED]…]
    [ON SIZE ERROR proc1]
    [NOT ON SIZE ERROR proc2]
[END-DIVIDE]
```

- id2 ← id2 / val1
- id3 ← id3 / val1
- `ROUNDED, ON SIZE ERROR` : cf. ADD
- diviser par 0 équivaut à un dépassement de capacité

24.1. 2° FORMAT

```
DIVIDE val1 {INTO|BY} val2
    GIVING id4 [ROUNDED] [id5 [ROUNDED]…]
    [ON SIZE ERROR proc1]
    [NOT ON SIZE ERROR proc2]
[END-DIVIDE]
```

- `INTO` : id4 ← val2 / val1
- `BY` : id4 ← val1 / val2

24.2. 3° FORMAT

```
DIVIDE val1 {INTO|BY} val2
    GIVING id4 [ROUNDED]
    REMAINDER id5
    [ON SIZE ERROR proc1]
    [NOT ON SIZE ERROR proc2]
[END-DIVIDE]
```

- id5 reçoit le reste de la division (dépend de la déclaration, pas forcément entier)

25. EVALUATE

Opérateur à la syntaxe complexe permettant le choix entre plusieurs possibilités

25.1. Utilisation simple

```
EVALUATE id
    WHEN lit2
        proc1
    …
    [WHEN OTHER
        proc2]
[END-EVALUATE]
```

- Le plus proche d'un switch en C.
- Attention à l'ordre. Une fois une condition remplie, on ne teste pas les autres.
- Pas de mécanisme de break comme en C car on ne passe pas au cas suivant
- Si plusieurs cas sont vérifiés, on ne retient que le premier.
- Si aucun cas n'est vérifié, on effectue la clause `OTHER` (ou rien si absente)

25.2. Généralisation pour les expressions numériques et alphanumériques

```
EVALUATE expr1
    WHEN [NOT] {expr2} [THRU {expr3}]
        proc1
    …
    [WHEN OTHER
        proc2]
[END-EVALUATE]
```

- Sans `THRU`, le cas est sélectionné si `expr1 = expr2`
- Il faut que les 2 expressions soient compatibles (numériques ou alphanumériques)
- avec `THRU`, le cas est sélectionné si `expr2<=expr1<=expr3`
- `NOT`, inverse le test.

25.3. Généralisation pour les conditions

```
EVALUATE {cond1|TRUE|FALSE}
    WHEN {cond2|TRUE|FALSE}
        proc1
    …
    [WHEN OTHER
        proc2]
[END-EVALUATE]
```

- Le cas est sélectionné si les 2 éléments ont la même valeur de vérité.
- on peut utiliser des noms de conditions.

25.4. ASSOCIATION DE CAS

```
EVALUATE el1
    WHEN el2
    WHEN el3
        proc1
    …
[END-EVALUATE]
```

- où `el` représente tout ce qui est permis là, à savoir une expression, une condition, `TRUE` ou `FALSE`.
- On peut associer plusieurs clauses `WHEN` à un paragraphe, il sera exécuté si la comparaison se vérifie pour au moins une des clauses, à savoir `el1=el2` ou `el1=el3`.

25.5. `ALSO` : PLUSIEURS CONDITIONS ENSEMBLES

```
EVALUATE el1 ALSO el2
    WHEN el3 ALSO el4
    …
    [WHEN OTHER proc2]
[END-EVALUATE]
```

- Le cas est vérifié si `el1=el3` et `el2=el4`
- On peut mélanger les expressions arithmétiques et les conditions

25.6. `ANY` : ELÉMENT UNIVERSEL

- `ANY` peut remplacer tout élément.
- Une comparaison avec `ANY` est toujours vérifiée.

Exemple récapitulatif

```
EVALUATE (A+1)     ALSO 0      ALSO B>C
    WHEN 0         ALSO D      ALSO TRUE
    WHEN 1         ALSO ANY    ALSO TRUE
        proc1
    WHEN ANY       ALSO NOT D ALSO FALSE
        proc2
    WHEN 0 THRU 4 ALSO D      ALSO B>A
        proc3
END-EVALUATE
```

- *proc1 est exécuté si (A+1)=0, D=0 et B>C ou (A+1)=1 et B>C*
- *proc2 est exécuté si D<>0 et B<=C*
- *proc3 est exécuté si 0<=(A+1)<=4 , D=0 et les conditions (B>C et B>A) ont la même valeur de vérité.*
- *Dans tous les autres cas, rien n'est vérifié*

26.IF

Branchement conditionnel.

```
IF cond THEN
   proc1
[ELSE
   proc2]
[END-IF]
```

- Attention à l'utilisation judicieuse du point
- Dans le cas de choix imbriqués, un `ELSE` se rapporte toujours au dernier `IF` ouvert.

27.INSPECT

Cet ordre permet
- de compter des caractères dans une chaîne
- de remplacer des parties de chaîne.

27.1. COMPTER

```
INSPECT var TALLYING id
    FOR {CHARACTERS    } [{BEFORE} [INITIAL] val2]…]
        {ALL val1      }  {AFTER }
        {LEADING val1 }
    …
```

- Analyse la variable `var`
- `id` est *incrémenté* du nombre de valeurs trouvées.
- `CHARACTERS` compte les caractères (quels qu'ils soient)
- `ALL val1` compte les occurrences de `val1` (une chaîne quelconque). Pas de recouvrement des chaînes

Exemple – INSPECT – Compter l'occurrence d'une sous-chaîne

```
INSPECT chaine TALLYING compteur FOR ALL "*" END-INSPECT
```
Compte le nombre d' dans la chaine*
```
MOVE "ABABABAB" TO chaine
INSPECT chaine TALLYING compteur FOR ALL "ABA" END-INSPECT
```
incrémente compteur de 2 unités

- `LEADING val1` compte les occurrences de val1 en début de chaîne.
- `AFTER INITIAL val2` ne commence à compter qu'après la rencontre de `val2`
- `BEFORE INITIAL val2` s'arrête de compter à la rencontre de `val2`.

27.2. REMPLACER

```
INSPECT var REPLACING
    {CHARACTERS BY val2}
    {ALL|LEADING|FIRST} val1 BY val2}
        [{BEFORE} [INITIAL] val3] …
         {AFTER }
    …
```

- Analyse la variable `var` afin d'y remplacer des parties de chaînes.
- `id` est *incrémenté* du nombre de valeurs trouvées.
- `CHARACTERS` : on remplace tous les caractères par `val2`
- `ALL val1 BY val2` : On remplace `val1` par `val2`.
- `FIRST val1 BY val2` : On remplace la première occurrence de `val1` par `val2`.
- `LEADING val1 BY val2` : On remplace `val1` par `val2` pour tous les `val1` en tête de chaîne.

Exemple – INSPECT – Remplacer une sous-chaîne

A faire

- Attention : les chaînes de remplacement doivent être de même longueur que les chaînes remplacées.

27.3. Compter et remplacer

Les 2 versions précédentes pour être combinées dans un même ordre. On compte avant de remplacer.

27.4. Remplacement de caractères

Si on veut remplacer une série de caractères, il existe une version plus compacte de l'ordre qui est

```
INSPECT var CONVERTING val1 TO val2
        [{BEFORE} [INITIAL] val3] …
         {AFTER }
```

- `val1` et `val2` ont la même taille
- Tout caractère de `val1` est remplacés dans `var` par le caractère en même position dans `val2`.

Exemple – INSPECT – Converting

```
INSPECT chaine CONVERTING "ABC" TO "EDF"
```

Remplace dans chaine, tous les A en E, tous les B en D et tous les C en F

28. MERGE

Fusion de fichiers.

28.1. Syntaxe simple

```
MERGE wrk-file
    {ASCENDING|DESCENDING} cle1 [clé2…]
    …
    USING file2 [file3…]
    GIVING file4
```

- Le fichier intermédiaire est traité comme avec le SORT
- Fusionne file2, file3, … et met le contenu dans file4
- Les fichiers sont triés sur une ou plusieurs clé
- Tous décrits en FD avec la même structure

28.2. Post-traitement

Ici aussi, on peut effectuer un post-traitement sur le résultat

```
MERGE wrk-file
    {ASCENDING|DESCENDING} cle1 [clé2…]
    …
    USING file2 [file3…]
    OUTPUT PROCEDURE nom-section
```

- La lecture dans le fichier intermédiaire ne se fait pas via un READ mais via un RETURN dont la syntaxe est identique à part cela.

29.MOVE

Transfert de données entre zones mémoires

```
MOVE val1 TO id1 …
```

Le type de transfert va dépendre de la variable réceptrice (id1)

- Alphabétique, alphanumérique (édité) : cadrage à gauche avec remplissage d'espaces ou troncature. Si val1 est signé, le signe est perdu.
- Numérique (édité) : cadrage sur le point décimal avec remplissage de 0. Le signe est gardé. Si val1 est alphanumérique, il ne peut contenir que des chiffres

Types de MOVE valides

source\récep.	αβ	αN	αN-éd	N	N-éd
αβ + SPACE	O	O	O	N	N
αN + litt. ¬N	O	O	O	O	O
αN-éd	O	O	O	N	N
N + litt. N + ZERO	N	O	O	O	O
N-éd (A VERIFIER)	N	O	O	O	O

30.MOVE : CLAUSE CORRESPONDING

Cette clause, présente dans les ordres MOVE, ADD et SUBTRACT permet d'effectuer en une seul ordre une opération sur plusieurs champs d'une structure.

30.1. PRINCIPE DE BASE : LE MOVE

```
MOVE CORRESPONDING struc1 TO struc2
```

- S'il existe un champ qui possède le même nom dans les 2 structures, on effectue l'opération, ici le MOVE, sur ces 2 champs. Quand on dit que le nom est le même, il s'agit du nom qualifié total (relativement à la structure)
- Le champ doit être élémentaire dans au moins 1 des 2 structures.
- Les champs non concernés ne sont pas modifiés
- Les numéros de niveau ne doivent pas forcément être identiques
- L'ordre d'apparition du champ dans la structure n'a pas d'importance

Exemple de MOVE CORRESPONDING

```
01 A.                    01 B.
   03 NOM.                  04 DATE-CR
   03 DATE-CR                  08 JJ
      06 JJ                    08 MM
      06 MM                    08 AA
      06 AA                 04 ZONES
   03 TRUC                     08 VAL
   03 ZONE                  04 NOM
      06 VAL
      06 VAL2

MOVE CORRESPONDING A TO B est équivalent à
MOVE NOM OF A TO NOM OF B
MOVE JJ OD DATE-CR OF A TO JJ OF DATE-CR OF B (idem MM et AA)
(pas de MOVE entre les 2 VAL qui n'ont pas le même nom au niveau supérieur)
(Si un des DATE-CR avait été élémentaire, ça fonctionne toujours)
```

Exemple de MOVE CORRESPONDING (2)

```
01 A.                    01 B.
   03 NOM.                  02 T.
   03 DATE-CR                  04 DATE-CR
      06 JJ                       08 JJ
      06 MM                       08 MM
      06 AA                       08 AA
   03 TRUC                     04 ZONES
   03 ZONE                        08 VAL
      06 VAL                   04 NOM
      06 VAL2

MOVE CORRESPONDING A TO B ne fait rien mais
MOVE CORRESPONDING A TO T a le même effet que l'exemple précédent.
```

30.2. OPÉRATIONS

La clause peut également apparaître dans des formes simples de ADD et SUBTRACT

```
ADD CORRESPONDING struc1 TO struc2
SUBTRACT CORRESPONDING struc1 FROM struc2
```

31.MULTIPLY

Multiplication

31.1. 1° FORMAT

```
MULTIPLY val1 BY id2 [ROUNDED] [id3 [ROUNDED]…]
    [ON SIZE ERROR proc1]
    [NOT ON SIZE ERROR proc2]
[END-MULTIPLY]
```

- id2 *← val1
- id3 *← val1
- `ROUNDED, ON SIZE ERROR` : cf. ADD

31.2. 2° FORMAT

```
MULTIPLY val1 BY val2
    GIVING id4 [ROUNDED] [id5 [ROUNDED]…]
    [ON SIZE ERROR proc1]
    [NOT ON SIZE ERROR proc2]
[END-MULTIPLY]
```

- id4 ← id5 ← val1 * val2

32.PERFORM

Répétition contrôlée d'une partie de code. Une fois le PERFORM terminé, on passe à l'instruction suivante.

32.1. Appel simple

```
PERFORM par1 [ THRU par2 ]
```

- On exécute `par1` (jusque `par2`) puis on revient.

32.2. `TIMES` : Itération nombre fixe de fois

```
PERFORM par1 [ THRU par2 ] {val} TIMES
```

- Si val <=0, on passe directement à l'instruction suivante
- Modifier `val` dans la procédure n'a plus d'influence

32.3. `UNTIL` : Tant que

```
PERFORM par1 [ THRU par2 ] UNTIL cond
```

- le test est effectué avant

32.4. `VARYING` : Boucle

```
PERFORM par1 [ THRU par2 ]
        VARYING id1 FROM {val2} BY {val3} UNTIL cond1
        [AFTER  id4 FROM {val5} BY {val6} UNTIL cond2 …]
```

- modification des variables prise en compte.
- Le cas simple correspond à (en C) : `for(id1=val2;!cond1;id1+=val3) {par1}`
- `AFTER` introduit les boucles imbriquées
- Le cas double correspond à
  `for(id1=val2,id4=val5;!cond1;id1+=val3,id4=val5)`
  `    for(;!cond2;id4+=val6)`

32.5. Clause `WITH TEST` : Répéter

```
PERFORM par1 WITH TEST {BEFORE|AFTER}
```

- Permet de forcer un répéter à la place d'un tant que
- Ne peut apparaître que là où il y a un `UNTIL`.
- La clause apparaît avant le reste (`UNTIL` ou `VARYING`)

32.6. Utilisation *en ligne*

```
PERFORM …
   proc
END-PERFORM
```

- Valable pour tout type de `PERFORM` (sauf clause `AFTER`). On indique le code plutôt que d'appeler un paragraphe. Seul cas où peut apparaître la clause `END-PERFORM`.

33.SEARCH

Permet une recherche dans une table possédant un index.

33.1. Recherche séquentielle

```
01 table.
    03 element OCCURS …

SEARCH element [VARYING {idx|don-idx|ind}]
    [AT END proc]
    [WHEN cond1 {proc1|CONTINUE}]
    …
[END-SEARCH]
```

- VARYING : Par défaut, on utilise le premier index associé au tableau mais on
  - indiquer un autre index du tableau qui sera utilisé à la place
  - indiquer un index d'un autre tableau qui suivra l'index principal du tableau
  - indiquer une donnée index qui suivra l'évolution de l'index principal du tableau
  - indiquer un indice qui suivra l'index principal du tableau
- On commence par l'élément pointé par l'index utilisé pour la recherche
- Lorsqu'une condition est vérifiée, on exécute la procédure correspondante puis on sort du `SEARCH`. La condition est quelconque.
- La recherche est faite sur tout le tableau pas uniquement la partie ayant un sens (sauf si table de longueur variable).
- Rien n'est prévu pour une recherche dans une table à plus d'une dimension; utilisation de boucles pour les niveaux supérieurs et d'un `SEARCH` pour le dernier niveau

33.2. Recherche dichotomique

Lorsque les éléments de la table sont triés, on peut effectuer une recherche plus rapide
Il faut indiquer le tri lors de la déclaration de la table.

```
01 table.
    03 element OCCURS
              [ASCENDING|DESCENDING KEY IS  id1 id2… ] …
```

- Indique que la table est triée en majeur sur id1 puis en mineur sur id2.
- id1 et id2 sont des parties des éléments de la table (ou tout l'élément)
- Il incombe au programmeur de respecter son ordre.
- Un index doit être associé à la table

```
SEARCH ALL element
    [WHEN CLE(idx) = ARG {proc1|CONTINUE}]
    …
[END-SEARCH]
```

- Recherche dans toute la table sauf si table variable → uniquement partie utile.
- Les conditions sont limitées à des égalités testées sur la clé principale au moins.

34. SORT

Tri d'un fichier.

34.1. Fichier intermédiaire

Le tri se fait via un fichier intermédiaire

```
SELECT wrk-file ASSIGN TO nom-assigné
…
SD wrk-file
01 nom-record
…
```

- Le fichier ne doit pas exister sur le système. Il sera crée.
- Pas de carte JCL pour ce type de fichier
- Déclaration en FILE SECTION par SD (sort descriptor) au lieu de FD.

34.2. Syntaxe simple

```
SORT wrk-file
    {ASCENDING|DESCENDING} cle1 [clé2…]
    …
    USING file2
    GIVING file3
```

- description des clés de tri dans l'ordre d'importance.
- file2 : fichier à trier, séquentiel.
- file3 : fichier trié, séquentiel, indexé (trié en ascending sur la clé primaire) ou relatif
- Tous les fichiers doivent être fermés. SORT se charge de les ouvrir et de les refermer à la fin.
- Tous décrits en FD avec la même structure

34.3. Pré-traitement

On peut effectuer un pré-traitement sur les enregistrements (sélection d'enregistrements ou de champs)

```
SORT wrk-file
    {ASCENDING|DESCENDING} cle1 [clé2…] …
    INPUT PROCEDURE IS nom-section
    GIVING file3
```

- Dans la section, on a la charge d'ouvrir les fichiers, lire les enregistrements, placer ce qu'il faut dans le fichier intermédiaire et refermer le fichier.
- L'écriture dans le fichier intermédiaire ne se fait pas via un WRITE mais via un RELEASE dont la syntaxe est identique à part cela.

34.4. Post-traitement

De la même façon, on peut effectuer un post-traitement sur le résultat (typiquement pour une impression)

```
SORT wrk-file
    {ASCENDING|DESCENDING} cle1 [clé2…] …
    USING file2
    OUTPUT PROCEDURE nom-section
```

- La lecture dans le fichier intermédiaire ne se fait pas via un READ mais via un RETURN dont la syntaxe est identique à part cela.

35.STRING

Concaténation de chaînes

```
STRING
    val1 [val2…] DELIMITED BY {val3|SIZE}
    […]
    INTO id
    [WITH POINTER id2]
    [ON OVERFLOW proc1]
    [NOT ON OVERFLOW proc2]
[END-STRING]
```

- Concatène simplement si délimité par SIZE.
- Sinon, on tronque val1 et val2 à la première occurrence de val3.
- si id est trop court, il y a troncature
- Si on ne remplit pas `id`, il n'y a pas de complétion avec des blancs

35.1. CLAUSE WITH POINTER

- indique où on commence dans le résultat et où on termine

35.2. CLAUSE OVERFLOW

- Lancé si on dépasse la zone réceptrice. (lié aussi à la valeur initiale du pointeur)

36.SUBTRACT

Soustraction

36.1. 1° FORMAT

```
SUBTRACT val1 [val2…] FROM id3 [ROUNDED] [id4 [ROUNDED]…]
    [ON SIZE ERROR proc1]
    [NOT ON SIZE ERROR proc2]
[END-SUBTRACT]
```

- id3 ← id3 – (val1 + val2)
- id4 ← id4 – (val1 + val2)
- `ROUNDED, ON SIZE ERROR` : cf. ADD
- Si la zone réceptrice n'est pas signée, elle reçoit la valeur absolue.

36.2. 2° FORMAT

```
SUBTRACT val1 [val2…] FROM val3
    GIVING id4 [ROUNDED] [id5 [ROUNDED]…]
    [ON SIZE ERROR proc1]
    [NOT ON SIZE ERROR proc2]
[END-SUBTRACT]
```

- id4 ← val3 – (val1 + val2)
- id5 ← val3 – (val1 + val2)

37.UNSTRING

Opération inverse de STRING. Décompose une chaîne

37.1. 1° FORMAT

```
UNSTRING id1
    DELIMITED BY [ALL] val1 [OR [ALL] val2…]
    INTO  id2 [DELIMITER IN id3] [COUNT IN id4]
          [id5 [DELIMITER IN id6] [COUNT IN id7]…]
    [WITH POINTER id8]
    [TALLYING IN id9]
    [ON OVERFLOW proc1]
    [NOT ON OVERFLOW proc2]
[END-UNSTRING]
```

- id1 : chaîne à décomposer
- val1, val2 : délimiteurs pour séparer les éléments (examinés dans l'ordre)
- ALL indique que plusieurs occurrences successives sont prises comme une seule (ALL SPACE <> SPACE)
- Si délimiteurs contigus (sans ALL) → mot vide
- id2, id5 : reçoivent les parties de id1 (via un MOVE -> padding et troncature)
- id3, id6 : reçoit le délimiteur trouvé pour cette partie là. (les délimiteurs ne sont pas compris dans les parties de chaîne)
- id4, id7 : nb de caractères transférés dans id2 ou id5
- id8 : pointeur pour indiquer où on commence à décomposer.
- id9 : nb de mots transférés (mots vides compris)
- overflow si on commence en dehors de la chaîne (mauvaise valeur du pointeur) où pas assez de zones réceptrices.

37.2. 2° FORMAT

```
UNSTRING id1
    INTO  id2 [id3…]
    [WITH POINTER id8]
    [TALLYING IN id9]
    [ON OVERFLOW proc1]
    [NOT ON OVERFLOW proc2]
[END-UNSTRING]
```

- On décompose en fonction de la taille des zones réceptrices.

– Partie V –
Table

Une table est une collection d'éléments de même nature qui sont liés sémantiquement. On présente ici tout ce qui les concerne : déclaration, initialisation, utilisation, …

38. Table : Déclaration et indice

38.1. Indice

- Entier > 0 déclaré WORKING-STORAGE section
- Pour désigner un élément, seul *id +/- littéral* est permis
- Le premier élément a le numéro 1 (pas 0 comme en C). Pas de vérification des bornes

38.2. Déclaration

```
01 table.
    03 element OCCURS lit TIMES …
```

- La clause ne peut pas apparaître aux niveaux 01 ou 77
- Permis dans toute la DATA division
- Référence à un élément : `element( indice )`
- Si nom qualifié, d'abord indiquer le qualificatif : `element of … (indice)`
- L'indice peut être lui-même un nom qualifié.
- L'indice peut-être un élément de tableau

38.3. Un élément de table peut-être une structure

```
01 table.
    03 element OCCURS lit TIMES
       05 a1 …
       05 a2 …
```

- Référence à un champ de l'élément : `a1( indice )`

38.4. Tables imbriquées

```
01 table.
    03 niv1 OCCURS lit1 TIMES
       05 niv2 OCCURS lit2 TIMES
```

- Permis : niv1(ind1) et niv2(ind1 ind2).
- Pas permis : niv1(ind1 ind2) ou niv2(ind1)
- Différent d'une structure contenant 2 tableaux séparés (si niv2 est au niveau 3 également)

38.5. Table de longueur variable

```
01 table.
    03 element OCCURS lit1 TO lit2 TIMES DEPENDING ON id
```

- `id` indique la taille logique de la table mais l'espace maximal est réservé et rien n'empêche d'y aller.
- Suivant la norme, on a 0 < lit1 < lit2 et la table doit être unique dans une structure et apparaître à la fin de celle-ci. Sur notre compilateur, les règles sont plus souples : 0 <= lit1 <= lit2 et on peut en avoir plusieurs.
- Toutefois, s'il s'agit d'un enregistrement écrit sur fichier, on écrira la table avec sa longueur courante, économisant ainsi de l'espace disque.

39. TABLE : INITIALISATION

Il existe plusieurs façons d'initialiser une table, en fonction de ce qu'on veut y mettre et du compilateur car certaines méthodes sont récentes.

39.1. LORS DE LA DÉCLARATION

Assignation générale à la table

Au niveau de la table, on indique la valeur d'un élément (ils auront tous la même valeur). Si l'élément est un groupe, on indique cette valeur en alphanumérique.

Exemple – Assignation générale à une table

```
01 T.
   03 el occurs 20 value "12".
      05 A pic 9.
      05 B pic 9.
```

Tous les A sont initialisés à 1 et les B à 2

Assignation particulière à la table

Si les éléments de la table sont des groupes, on peut indiquer individuellement la valeur de chaque partie.

Exemple – Assignation particulière à une table

```
01 T.
   03 el occurs 20.
      05 A pic 9 value 1.
      05 B pic 9 value 2.
```

Tous les A sont initialisés à 1 et les B à 2

Via un REDEFINES

Utile pour assigner des valeurs différentes à chaque élément de la table.

Exemple – Assignation via un REDEFINES

```
01 T-init.
   03 pic X(20) value "Janvier".
   …
   03 pic X(20) value "Décembre".
01 T REDEFINES T-init.
   03 mois PIC X(20) occurs 12.
```

mois(3) contiendra "Mars"
Attention : pas dans l'autre sens car le VALUE doit être dans la partie redéfinie pas dans l'autre.

39.2. DANS LA PROCEDURE DIVISION

Par des MOVE explicites ou par lecture d'un fichier.

40. TABLE : INDEX

40.1. INDEX

```
01 table.
    03 element OCCURS lit TIMES INDEXED BY idx1 [idx2 …]
```

- Un index est associé à un et un seul vecteur.
- Ne doit pas être déclaré ailleurs
- Il possède une vue externe (comme un indice) et une vue interne (déplacement en octets dans le vecteur). Permet un accès plus rapide.
- Si un index existe, on peut utiliser des indices aussi mais on ne peut pas mélanger index et indices dans un même accès d'élément (cas des tableaux multi-dimensionnels)

40.2. DONNÉES INDEX

```
77 idx USAGE IS INDEX
```

- Pas de PIC. Format géré par le compilateur.
- Pas de valeur externe (puisque pas associé à un tableau). Sert de zone de stockage.

40.3. SET : MODIFIER LA VALEUR D'UN INDEX

Un index ne peut être manipulé par les mêmes ordres qu'une variable classique. On a

```
SET idx TO {val|idx}
SET idx {UP|DOWN} BY val
```

- On peut combiner indices et index si cela a un sens.

40.4. VARYING : INDEX COMME ITÉRATEUR D'UNE BOUCLE

- Le seul autre endroit où un index peut être utilisé, c'est comme itérateur d'une boucle (clauses VARYING et FROM).

– Partie VI –
Fichier

41. Types de fichiers

41.1. Fichiers séquentiels

La difficulté peut venir des fichiers à enregistrements de longueur variable

41.2. Fichiers séquentiels à enregistrements de longueur variable

Un fichier séquentiel peut avoir des enregistrements de longueur variable pour 2 raisons

- Le fichier contient des enregistrements de type différent
- Il contient des enregistrements avec des tables de longueur variable.
- Au niveau physique, chaque enregistrement est précédé d'un *Record Word Descriptor*, un nombre sur 4 octets indiquant la taille de l'enregistrement qui suit.

Lecture

- Au niveau de COBOL, on ne peut pas connaître la taille de l'enregistrement que l'on vient de lire. Il faut donc gérer cela au niveau du programme

Écriture

- L'ordre d'écriture va indiquer la taille de l'enregistrement à écrire puisqu'on indique le masque à utiliser.
- Si une table de longueur variables est présente, on ne sauvera que les éléments effectivement présents dans cette table.

41.3. Fichiers indexés

- Les enregistrements pourront être accédés via une clé primaire ou des clés secondaires.

41.4. Fichiers relatifs

- Ensemble d'enregistrements de même longueur.
- Chaque enregistrement est caractérisé par un numéro (commence à 1)
- Il peut être libre ou occupé. S'il est libre, il ne sera pas considéré par une lecture séquentielle et une lecture random lancera la clause d'erreur.
- Tout accès à un enregistrement modifie la variable représentant le numéro de l'enregistrement pour refléter celui qui a été traité (par une lecture par exemple).

41.5. Tableau récapitulatif

- Récapitulatif des ordres possibles en fonction du type de fichier, du mode d'accès et du mode d'ouverture.

	Séquentiel				**Random**		
	INPUT	**OUTPUT**	**EXTEND**	**I-O**	**INPUT**	**OUTPUT**	**I-O**
READ	S,R,I			S,R,I	R,I		R,I
WRITE		S,R,I	S			R,I	R,I
REWRITE				S,R,I			R,I
START	R,I			R,I			
DELETE				R,I			R,I

- S (séquentiel), R (relatif), I (indexé)
- Pour un mode d'accès dynamique, on combine les 2.

42.SELECT

```
SELECT nom-cobol
       ASSIGN TO nom-externe
       [ORGANIZATION IS SEQUENTIAL|INDEXED|RELATIVE]
       [ACCESS MODE  IS SEQUENTIAL|RANDOM|DYNAMIC]
       [RELATIVE KEY IS clé]
       [RECORD KEY IS clé]
       [ALTERNATE RECORD KEY IS clé2 [WITH DUPLICATES]] …
       [FILE STATUS  IS id]
       .
```

- nom-cobol peut être identique à nom-externe
- Cf fiches sur le JCL et AS/400
- `ORGANIZATION` indique l'organisation physique du fichier (`SEQUENTIAL` par défaut)
- `ACCESS MODE` indique la manière d'y accéder
  - `SEQUENTIAL` (défaut) : purement séquentiel
  - `RANDOM` : purement à la demande
  - `DYNAMIC` : combinaison des deux (on passe à volonté du mode séquentiel au mode random)
  - Ces 2 derniers modes ne sont pas valables pour un fichier à organisation séquentielle
- `RELATIVE KEY` (fichier relatif uniquement) indique la variable qui donnera le numéro de l'enregistrement courant.
  Elle sera décrite comme numérique dans la `WORKING-STORAGE`.
- `RECORD KEY` (fichier indexé uniquement) donne le champ qui servira de clé primaire
- `ALTERNATE RECORD KEY` permet de désigner des clés secondaires (avec ou sans duplication possible)
- `FILE STATUS` est décrit dans une autre fiche.

clause ASCENDING / DESCENDING ?

- ALTERNATE RECORD définit une clé secondaire qui doit faire partie de l'enregistrement. (remarque sur le JCL)

43. CLAUSE : FILE STATUS

```
SELECT nom-cobol …
       [FILE STATUS  IS id]
       .
…
77 id PIC XX.
```

- La variable reprenant le file status doit être déclarée au WORKING-STORAGE comme un XX.
- A chaque opération sur le fichier, il reçoit un code indiquant comment l'opération s'est passée

Liste non exhaustive des code renvoyés après une opération sur un fichier	
00	OK
10	fin de fichier
23	pas d'enregistrement ayant cette clé
30	erreur matérielle
35	fichier inexistant
41	fichier déjà ouvert
42	fichier non ouvert

44.OPEN / CLOSE

44.1. OPEN

```
OPEN {INPUT|OUTPUT|EXTEND|I-O} file1 [file2…]  …
```

- INPUT : fichier existe, positionné au début, lecture seule
- OUTPUT : fichier n'existe pas, crée (saufs certains fichiers temporaires qui seront écrasés). Écriture seule
- EXTEND : fichier existe, positionné à la fin. Écriture seule
- I-O : fichier existe, positionné au début. On pourra réécrire l'enregistrement qui vient d'être lu.

44.2. CLOSE

```
CLOSE file1 [file2…]
```

- Les fichiers qui ne sont pas explicitement fermés, le sont par l'instruction STOP RUN.
- Après fermeture du fichier, le contenu du buffer (associé au fichier dans la `FILE-SECTION`) n'est plus garanti.

Exemple – Fermeture d'un fichier

```
FD FICHIER.
01 ENREG.
   03 CONTENU PIC X(10).

OPEN INPUT FICHIER
READ FICHIER
DISPLAY CONTENU         (OK)
CLOSE FICHIER
DISPLAY CONTENU         (Peut contenir n'importe quoi!)
```

45.READ

L'ordre diffère en fonction du mode d'accès au fichier. Dans tous les cas,

- Le fichier doit être ouvert en mode `INPUT` ou `I-O`.

45.1. Accès séquentiel

```
READ file [INTO id]
    [AT END proc1]
    [NOT AT END proc2]
[END-READ]
```

- Lit l'enregistrement courant sur le fichier.
- Avec `INTO`, un `MOVE` du buffer vers la variable est effectué après la lecture
- La clause `AT END` est lancée si on essaie de lire au delà de la fin du fichier.

45.2. Accès random

```
READ file [INTO id] [KEY …]
    [INVALID KEY proc1]
    [NOT INVALID KEY proc2]
[END-READ]
```

- Lit l'enregistrement indiqué par son numéro (fichier relatif) ou sa clé (fichier indexé)
- `KEY` : (valable uniquement pour les fichiers indexés) permet de spécifier explicitement la clé (primaire:défaut ou secondaire) utilisée.
- La clause `INVALID KEY` est lancée si l'enregistrement demandé n'existe pas.

45.3. Accès dynamic

- L'ordre de lecture random ne change pas.
- L'ordre de lecture séquentielle devient

```
READ file NEXT [INTO id]
    [AT END proc1]
    [NOT AT END proc2]
[END-READ]
```

46.WRITE / REWRITE / DELETE

- L'ordre pour les fichiers d'impression est décrit dans une fiche séparée.

46.1. Fichier séqentiel

```
WRITE enregistrement [FROM id]
[END-WRITE]
```

- Avec `FROM`, un `MOVE` est effectué de la variable vers le buffer avant l'écriture

```
REWRITE enregistrement [FROM id]
[END-REWRITE]
```

- Ne peut être utilisé que pour réécrire le dernier enregistrement lu par un ordre READ (réussi!)
- L'enregistrement doit être de même longueur.

46.2. Fichier indexé ou relatif

```
WRITE enregistrement [FROM id]
    [INVALID KEY proc1]
    [NOT INVALID KEY proc2]
[END-WRITE]
```

- On donne une valeur à la clé ou au numéro d'enregistrement et on lance l'écriture.
- L'enregistrement ne peut pas déjà exister.
- Si le fichier est ouvert en OUTPUT pour un accès séquentiel (typique lors de la création), les enregistrements doivent être écrits dans l'ordre de la clé (cas indexé) ou seront écrits dans l'ordre croissant sans tenir compte du numéro donné (cas relatif)
- La clause INVALID KEY est lancée si on tente d'écrire un enregistrement qui existe déjà ou qu'on ne respecte pas la contrainte pour un fichier OUTPUT en mode séquentiel.

```
REWRITE enregistrement [FROM id]
    [INVALID KEY proc1]
    [NOT INVALID KEY proc2]
[END-REWRITE]
```

- En accès séquentiel, remplace celui qui vient d'être lu. Doit avoir la même taille.
- En accès random ou dynamique, on récrit en fonction de la clé.
- L'enregistrement peut avoir une longueur différente.
- La clause INVALID KEY est lancée si on tente d'écrire un enregistrement qui n'existe pas

```
DELETE file
    [INVALID KEY proc1]
    [NOT INVALID KEY proc2]
[END-DELETE]
```

- Accès séquentiel : supprime le dernier enregistrement lu. (pas de clause `INVALID KEY`)
- Autres accès : supprime l'enregistrement donné par la clé.

47.START

Permet de se positionner avant une lecture séquentielle

47.1. START

```
START file
    [KEY {=|<|>|<=|NOT >|…} clé]
    [INVALID KEY proc]
    [NOT INVALID KEY proc]
[END-START]
```

- Valide pour un accès séquentiel ou dynamic.
- Se positionne sur l'enregistrement donné par la clé (sans lecture)
- KEY indique à la fois la clé de référence et le test. (défaut: clé primaire et égalité)
- clé est soit la clé primaire ou une clé secondaire (fichier indexé) soit la varibale utilisée pour référencé le numéro d'enregistrement (fichier relatif).
- exemple : `MOVE 32 TO clé. START file KEY NOT < clé`
  se positionne sur le premier enregistrement dont la clé est supérieure ou égale à 32.
- La clause `INVALID KEY` est lancée si le positionnement ne peut se faire.

– Partie VII –
Fonction et sous-programme

On traite ici les fonctions intrinsèques et la découpe d'une application en différents programmes ce qui permet de simuler le concept de fonction et facilite la réutilisation de code.

48. FONCTION INTRINSÈQUE

Attention : Ce mécanisme est une extension au COBOL-85 présente sur le mainframe mais pas sous AS/400 pour l'instant.

Il existe un certain nombre de fonctions prédéfinies qui peuvent être utilisées pour les calculs.

```
FUNCTION nom-fonction[(arg1 arg2 …)]
```

On voit que les parenthèses sont absentes si il n'y a pas d'argument.
Voici ces fonctions groupées par contexte

Nom	Arguments	Retour	Sens
Conversions			
CHAR	I1	X	caractère de code I1
INTEGER	N1 : réel	entier	plus grand entier <= N1
INTEGER-PART	N1 : réel	entier	partie entière (<> de INTEGER pour les nombres négatifs)
NUMVAL	X1 : chaîne	réel	X1 représente un nombre; retourne sa valeur
NUMVAL-C	X1, X2 : chaîne	réel	idem mais X2 donne le signe monétaire à négliger (les virgules le seront aussi)
ORD	X1 : un caractère	entier	code du caractère
Fonctions mathématiques			
ACOS	N1 : réel [-1..1]	réel	arccos en radian
ASIN	N1 : réel [-1..1]	réel	arcsin en radian
ATAN	N1 : réel [-1..1]	réel	arctg en radian
COS	N1 : réel en radian	réel	cosinus
FACTORIAL	1 : entier >=0	entier	factorielle
LOG	N1 : réel > 0	réel	ln, logarithme népérien
LOG10	N1 : réel > 0	réel	log, logarithme en base 10
MOD	I1, I2<>0 : entiers	entier	I1 modulo I2 (utilise INTEGER)
REM	N1, N2<>0 : entiers	entier	reste de N1/N2 (utilise INTEGER-PART)
SIN	N1 : réel en radian	réel	sinus
SQRT	N1 : réel > 0	réel	racine carrée
TAN	N1 : réel en radian	réel	tangente
Fonctions statistiques			
MAX	N1..Nn même type T	T	valeur maximale
MEAN	N1..Nn réels	réel	moyenne arithmétique
MEDIAN	N1..Nn réels	réel	valeur médiane
MIDRANGE	N1..Nn réels	réel	moyenne entre min et max
MIN	N1..Nn même type T	T	valeur minimale
ORD-MAX	N1..Nn même type T	entier	position du maximum
ORD-MIN	N1..Nn même type T	entier	position du minimum
RANGE	N1..Nn entier ou réel	idem	différence entre min et max
STANDARD-DEVIATION	N1..Nn réels	réel	écart type
SUM	N1..Nn entier ou réel	idem	somme des arguments
VARIANCE	N1..Nn réels	réel	variance
Fonctions économiques			
ANNUITY	N1 >=0, I2 >0	réel	annuité
PRESENT-VALUE	N1, N2	réel	???

Manipulation de dates			
CURRENT-DATE	aucun	chaîne	date et heure courante AAAAMMJJHHSS….
DATE-OF-INTEGER	I1 > 0	I	I1 représente le nombre de jours écoulés depuis le 1/1/1600 du calendrier grégorien. Remet cette date sous la forme AAAAMMJJ
DAY-OF-INTEGER	I1 > 0	I	idem ci-dessus mais retourne AAAAJJJ
INTEGER-OF-DATE	I1 : AAAAMMJJ	I	nb de jours depuis le 1/1/1600
INTEGER-OF-DAY	I1 : AAAAJJJ	I	nb de jours depuis le 1/1/1600
WHEN-COMPILED		I	date et heure de compilation
Manipulation de chaînes			
LENGTH	chaîne ou nombre	entier	longueur de la zone mémoire
LOWER-CASE	1 : chaîne	chaîne	majuscules -> minuscules
REVERSE	1 : chaîne	chaîne	inverse la chaîne
UPPER-CASE	1 : chaîne	chaîne	minuscules -> majuscules
Aléatoire			
RANDOM	1 : entier	réel	nombre aléatoire dans [0 1[. L' argument débute la série (option).

Exemple –fonctions intrinsèques

```
FUNCTION MAX(-12.3, 1.2, 45)                    45
FUNCTION MIN("Hello", "Bonjour", "brol")        Bonjour
FUNCTION INTEGER(-2.1)                          -3
FUNCTION INTEGER-PART(-2.1)                     -2
FUNCTION MEDIAN(1 3 6 9)                        4.5
FUNCTION NUMVAL( "123" )                        123
FUNCTION NUMVAL-C( "$1,123", "$" )              1123
```

- Une fonction peut apparaître partout où une expression le peut.
- L'ordre MOVE est un cas particulier; il accepte les fonctions non numériques mais pas les fonctions renvoyant un numérique pour lesquelles il faut utiliser un COMPUTE.

Exemple – utilisation des fonctions intrinsèques

```
77 A PIC X(30).
77 B PIC 9.9(6).

MOVE FUNCTION CURRENT-DATE TO A
MOVE FUNCTION REVERSE(A) TO A
COMPUTE B = FUNCTION RANDOM
```

49. PROGRAMMES IMBRIQUÉS

Jusqu'à présent, nous avons vu des programmes COBOL uns et indivisibles. Il est possible de définir des exécutables qui sont formés de plusieurs *programmes* COBOL. Ces *programmes* peuvent être : imbriquées, un à la suite de l'autre, dans des unités de compilation différentes.

49.1. END PROGRAM

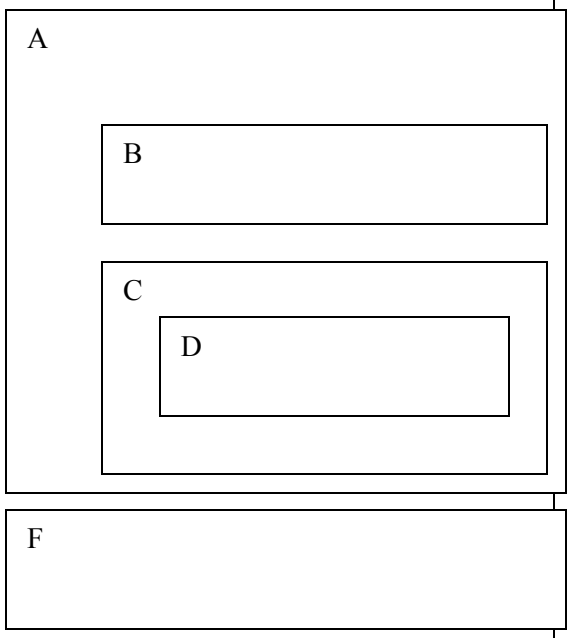
Cette instruction indique la fin d'un programme COBOL. Elle termine donc la `PROCEDURE DIVISION`. On doit indiquer le nom donné au paragraphe `PROGRAM-ID`. Nous ne l'avons pas encore rencontrée parce qu'elle est facultative. Cela permet d'avoir plusieurs programmes dans le même source. Le programme imbriqué se trouvera dans la `PROCEDURE DIVISION`.

Exemple – Plusieurs programmes dans le même source

Avec le code suivant

```
IDENTIFICATION DIVISION.
PROGRAM-ID. a.
…
IDENTIFICATION DIVISION.
PROGRAM-ID. b.
…
END PROGRAM b.
IDENTIFICATION DIVISION.
PROGRAM-ID. c.
…
IDENTIFICATION DIVISION.
PROGRAM-ID. d.
…
END PROGRAM d.
END PROGRAM c.
END PROGRAM a.
IDENTIFICATION DIVISION.
PROGRAM-ID f.
…
END PROGRAM f.
```

On a



B et C sont directement imbriqués dans A.
D est directement imbriqué dans C et indirectement imbriqués dans A.
A et F sont au même niveau. Il est en de même de B et C.

49.2. VISIBILITÉ

La structure donnée aux programmes va indiquer leur visibilité. Un programme ne pourra utiliser qu'un programme directement imbriqué ou niveau supérieur d'un programme séparé. En terme de relations familiales, on dira qu'un programme ne peut utiliser que ses filles directs ou les patriarches. En aucun cas, il ne peut y avoir de récursion (A appelle F qui appelle A)

Exemple – Appels de programmes

En reprenant l'exemple précédent, on a

A peut appeler B, C et F
B peut appeler F
C peut appeler D et F
D peut appeler F
F peut appeler A

49.3. COMMON

Cet attribut d'un programme modifie sa visibilité.

```
PROGRAM-ID. Nom COMMON.
```

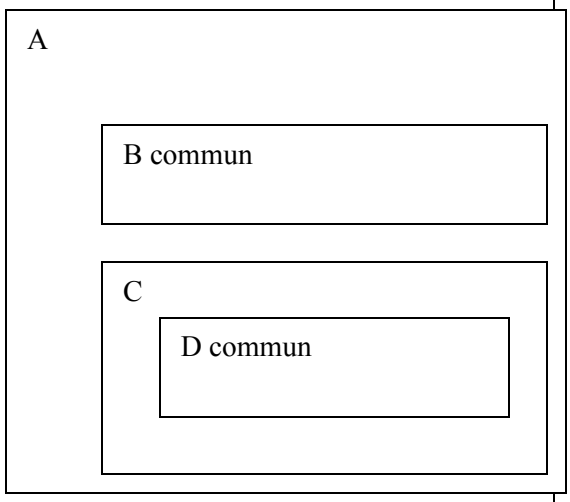
Un tel programme sera également visible par tous les programmes imbriqués (même indirectement) par le programme qui contient ce programme commun. En terme de liens de parenté, et pour être plus clair, il peut-être référencé par les frères et tous leurs descendants.

Exemple – Appel de programmes COMMON

Avec le code suivant

```
IDENTIFICATION DIVISION.
PROGRAM-ID. a.
…
IDENTIFICATION DIVISION.
PROGRAM-ID. b COMMON.
…
END PROGRAM b.
IDENTIFICATION DIVISION.
PROGRAM-ID. c.
…
IDENTIFICATION DIVISION.
PROGRAM-ID. d COMMON.
…
END PROGRAM d.
END PROGRAM c.
END PROGRAM a.
```

On a



En plus des appels déjà vus, on a que C et D peuvent appeler B
B ne peut pas appeler C.

50. Appel de sous-programmes

Il est possible dans un programme COBOL d'appeler un autre programme COBOL et même un programme écrit dans un autre langage; de même, un programme COBOL peut être appelé à partir d'un programme écrit dans un autre langage. Ici, nous nous intéressons au cas COBOL -> COBOL.
Appelons PP le programme appelant et SP le sous-programme appelé.

50.1. Appel simple

```
PP: CALL val1

SP:
PROGRAM-ID. val1.
```

- `val1` est une chaîne donnant le nom du programme SP; celui-ci est le nom donné par le paragraphe `PROGRAM-ID`. Si les 2 programmes appartiennent à des sources différents, cela correspond normalement au nom donné pour l'objet mais pas toujours.
- Si les 2 programmes appartiennent à des sources différents, ils seront regroupés en un seul exécutable à l'édition des liens.
- SP doit se terminer d'une certaine manière pour que le contrôle revienne au PP pour continuer après l'instruction `CALL`.
- Un SP peut lui-même effectuer un `CALL` et être donc un PP pour un autre SP.

50.2. Terminer un (sous) programme

Il y a 3 façon de terminer un programme. L'effet est différent s'il s'agit d'un PP ou d'un SP

	PP	SP
STOP RUN	Termine le processus	
GOBACK	Termine le processus	Retour au PP
EXIT PROGRAM	Sans effet	Retour au PP

- On voit que GOBACK convient dans tous les cas et permet à un programme d'être utilisé dans les 2 situations.
- Si on n'indique aucun de ces ordres, un `GOBACK` implicite est exécuté en fin de programme.

51. Passage d'arguments

Il y a plusieurs façons pour des programmes que communiquer des informations. Évidemment, on recommandera de toujours utiliser le passage d'arguments mais on peut également définir des variables globales.

51.1. Variables globales

Une variable de niveau 01 peut se voir attribuer l'attribut `GLOBAL`. Elle sera alors connue et utilisable directement dans tous les programmes imbriqués (même indirectement), càd dans tous les descendants. Un programme imbriqué peut définir une variable de même nom (éventuellement globale, elle aussi), auquel cas la variable globale n'est plus accessible.

51.2. Variables externes

Une variable de niveau 01 peut se voir attribuer l'attribut `EXTERNAL`. Elle est alors commune aux autres programmes séparés qui la déclare également comme externe. La clause `VALUE` n'est pas compatible. Elle doivent porter le même nom et avoir la même définition.

Exemple – Variables externes

```
IDENTIFICATION DIVISION.      IDENTIFICATION DIVISION.
PROGRAM-ID a.                 PROGRAM-ID b.
DATA DIVISION.                DATA DIVISION.
WORKING-STORAGE SECTION.      WORKING-STORAGE SECTION.
01 n PIC 9 EXTERNAL.          01 n PIC 999 EXTERNAL.
PROCEDURE DIVISION.           PROCEDURE DIVISION.
prog.                         prog.
    move 1 TO n.                  add 1 TO n.
    call "B".                 end program b.
    display n. (Affiche 2)
end program a.
```

51.3. Passage d'arguments

- Au niveau du PP, on donne la liste des arguments à passer lors du `CALL`.
- Ces arguments sont des variables (on ne peut pas passer un littéral !)

```
CALL sp USING arg1 arg2 … argn
```

- Dans le SP, il faut déclarer ces arguments dans une section à part et indiquer la liste au niveau de la `PROCEDURE DIVISION`.

```
DATA DIVISION.
LINKAGE SECTION.
77 arg1 PIC …
…
77 argn PIC …

PROCEDURE DIVISION USING arg1 arg2 … argn.
```

- Dans la LINKAGE SECTION, on interdit la clause VALUE qui n'a pas de sens.
- Les variables de cette section doivent être de niveau 01 ou 77 uniquement.
- Les listes des arguments dans PP et SP doivent correspondre exactement en nombre, en ordre et en type (les noms peuvent être différents)
- La concordance en type implique qu'il n'y a aucune conversion, même en ce qui concerne la taille. (*pas d'erreur générée ni à la compilation ni à l'exécution mais comportement erroné !)*

Exemple – Passage de paramètre

```
PP:
A PIC 99 VALUE 12.
CALL "SP" USING A.

SP:
LINKAGE SECTION.
A PIC 999.
PROCEDURE DIVISION USING A.
DISPLAY A.      (affichera 12 + l'octet mémoire suivant; imprévisible !!!)
```

51.4. Types de passage d'arguments

Par défaut, le passage d'argument se fait par référence mais on peut spécifier un passage par valeur. On remarque que cela est spécifié à l'appel et pas du tout à la définition.

```
CALL val USING {[BY REFERENCE id …]
                BY CONTENT   id … } …
```

- Dans un passage par valeur, il y a deux zones indépendantes pour stocker les arguments; la zone du PP et celle du SP. A l'appel, il y a recopie de la zone PP vers la zone SP. Dans ce cas, pour certains compilateurs (comme l'AS/400) et pour certains types de variables (pas de numérique), un littéral peut-être passé.
- Dans un passage par argument, le PP et le SP partagent la même zone. A l'appel, une modification de variable par le SP est répercutée dans le PP.

– Partie VIII –
Annexes

52. MVS ET LE TRAITEMENT BATCH

52.1. UN JOB TYPIQUE

Un job COBOL aura typiquement l'aspect suivant :

Exemple – Un JCL typique

```
//* ================================
//* Job typique pour un programme COBOL
//* ================================
//ANDR010 JOB ANDR010,'Codutti',CLASS=P,MSGCLASS=Z,NOTIFY=ROSEALN
//* 010     : représente le numéro d'utilisateur
//* Codutti : Nom de l'utilisateur
//* ALN     : représente la librairie personnelle
//CO1 EXEC IGYWCLG,PARM='paramètres de compilation'
//* IGYWCLG est le compilateur que nous utilisons
//* CLG pour Compile, Link and Go
//* En général, pas de paramètre -> option non mise
//COBOL.SYSIN DD *
     programme COBOL
/*
//GO.SYSOUT  DD SYSOUT=Z,OUTLIM=100
//* On limite la sortie à 100 lignes
//GO.FICHIER DD DSN=ANDR…,DISP=SHR
```

Note : Les lignes débutant par `//*` sont des commentaires et ne sont pas nécessaires.

52.2. FICHIERS PRODUITS

Les fichiers produits par l'exécution de ce job sont :

- JES2.JESMSGLG : Résumé du job. Chaque étape y est reprise avec son code de terminaison. Permet de savoir rapidement où a eu lieu l'erreur (syntaxe JCL, compilation, édition des liens, exécution, …)
- JES2.JESJCL : Le job est repris et agrémenté d'infos indiquant comment il a été compris par la machine (rarement utile)
- JES2.JESYSMSG : Tous les messages produits par le module de prise en charge du job. Les erreurs de syntaxe dans le job y apparaîtront explicitement ainsi que des problèmes avec les fichiers. très peu lisible mais on est parfois obligé de s'y plonger.
- CBLEX1.COBOL.SYSPRINT : Le code COBOL compilé. C'est ici qu'on trouve les erreurs de compilation. (note: CBLEX1 est le nom donné à l'étape COBOL dans le JCL; peut-être différent pour vous)
- CBLEX1.LKED.SYSPRINT : Le résultat de l'édition des liens. On verra les erreurs liées à des fichiers inexistants.
- CBLEX1.GO.SYSOUT : Fichier produit à l'exécution. On y trouvera les résultats des ordres DISPLAY mais aussi les messages d'erreurs produits à l'exécution (division par 0, …)
- CBLEX1.GO.LISTE : Fichier produit par le programme (résultats, …)

53. EXTENSIONS PROPRES À MVS

53.1. RECORDING MODE

Clause accompagnant la description des enregistrements d'un fichier et indiquant le type d'enregistrement.

```
FD nom-fichier [RECORDING MODE IS {F V U}].
01 enr1.
    …
```

- F : enregistrements de taille fixe
- V : taille variable
- U : taille indéfinie
- Si cette clause n'est pas présente, le compilateur la déduit de la structure du fichier et émet un message d'information pour indiquer son choix.

54. Extensions propres à AS/400

54.1. Type booléen

A écrire

54.2. Clause LIKE

A écrire

54.3. Utilisation de la DDS

A écrire

55. TABLE EBCDIC

Extraits

Code déc.	Code hexa	Caractère
64	40	espace
78	4E	+
96	60	-
192	C0	{
193	C1	A
…		
201	C9	I
…		
208	D0	}
209	D1	J
…		
217	D9	R
…		
240	F0	0
…		
249	F9	9