MASTER EN SCIENCES INFORMATIQUES

Usage de la compilation conditionnelle dans le cadre de la modernisation de codes COBOL
vers plus de variabilité, de sécurité et un « design » plus contemporain (émulation OO)

BAUWENS, Joël

Award date:
2023

Awarding institution:
Universite de Namur

UNIVERSITÉ DE NAMUR
Faculté d'informatique
Année académique 2022–2023

Usage de la compilation conditionnelle dans le cadre de la modernisation de codes COBOL : Vers plus de variabilité, de sécurité et un « design » plus contemporain (émulation OO)

Joël BAUWENS

……...(Signature pour approbation du dépôt - REE art. 40)

Promoteur : Anthony CLEVE

Mémoire présenté en vue de l'obtention du grade de Master 60 en Sciences Informatiques

Faculté d'Informatique – Université de Namur
RUE GRANDGAGNAGE, 21 ● B-5000 NAMUR(BELGIUM)

Remerciements

C’est au professeur Anthony Cleve, mon promoteur, que je tiens à adresser mes premiers remerciements pour ses conseils, son soutien positif, ses questions et idées, et avant cela pour avoir accepté ce sujet de mémoire portant sur un aspect du COBOL.

Pour avoir mis de l’énergie, établi des contacts dans divers pays et avoir consacré du temps à faire en sorte que je puisse disposer durant 60 jours du compilateur COBOL version 1.1 pour Linux x86 édité par IBM, je tiens à remercier Me Nicole Trudeau (IBM Canada). Si ça n’a jamais fonctionné dans le délais de 60 jours, ça ne lui est certainement pas imputable.

Pour la licence Etudiant (1 an) de Micro Focus Cobol (V8.0) pour Visual Studio, je tiens à remercier Micro Focus et particulièrement M. Hassan Mouterfi qui fut efficace pour finaliser cette obtention.

Sans l’autorisation d’utiliser la nuit le mainframe UNISYS pour, avant tout autre compilateur, éprouver les idées, la réalisation de ce travail n’aurait pas été aussi fructueuse. Pour la confiance accordée et l’accord donné, je tiens à remercier l’ONEm et notamment M. Geert Dewaersegers, responsable de l’IT.

Pour m’avoir fourni le texte (dernier draft avant publication officielle) de la norme Fortran ISO/IEC 1539-3 je tiens à remercier M. André Madarasz de NBN.be.

Pour sa patience pendant toutes ces heures que je consacrais à ce travail, indisponible pour tout le reste, et où elle assumait seule tout ce qui fait vivre un foyer, je tiens particulièrement à remercier mon épouse Anne.

Table des matières

Liste des figures

Résumé

Le présent travail étudie la compilation conditionnelle en COBOL. Pour ce faire les notions de COBOL nécessaires et suffisantes à comprendre ce travail sont fournies. Les concepts utilisés de compilation conditionnelle ainsi que de ligne de produits logiciels sont précisés. L'apparition de la compilation conditionnelle pour les langages ALGOL, FORTRAN et COBOL sur mainframe sera suivie, les raisons de son apparition ne seront formulées que sous forme d'hypothèses. Quelques syntaxes de compilation conditionnelle disponibles en COBOL avant la norme 2002 sont décrites, analysées, comparées (cette partie contient déjà des exemples d'usages sophistiqués) et des lignes directrices pour passer/migrer de l'une à l'autre sont données. Il n'y a aucun but de lister de manière exhaustive (ce serait impossible) tous les usages de la compilation conditionnelle en COBOL, l'orientation a été prise de fournir de manière didactique et évolutive pour que le lecteur appréhende la technique, des codes répondant à l'objectif de factorisation, de sélection à la carte, de rassemblement des variables et paragraphes dans un même fichier en dehors de toute division. L'usage de la technique est poussé jusqu'à l'émulation partielle de l'OO (variables membres, variables de travail, méthodes, héritage).

Mots-clés : *cobol, compilation conditionnelle*

Abstract

This work studies conditional compilation in COBOL. The COBOL notions necessary and sufficient to understand this work are provided. The concepts of conditional compilation and software product lines are explained. The appearance of conditional compilation for the Algol, Fortran and COBOL languages on the mainframe will be followed, the reasons for its appearance will only be formulated in the form of hypotheses. Some conditional compilation syntaxes available in COBOL before the 2002 standard are described, analysed and compared (this part already contains examples of sophisticated uses) and guidelines for switching/migrating from one to another are given. There is no attempt to provide an exhaustive list of all the uses of conditional compilation in COBOL (that would be impossible), but the aim is to provide, in a didactic and progressive way so that the reader can grasp the technique, codes that meet the objectives of factorisation, à la carte selection, and the collection of variables and paragraphs in a single file without any division. The use of the technique is taken as far as partial emulation of the OO (member variables, work variables, methods, inheritance).
(Deepl Traduction)

Keywords : *cobol, conditional compilation*

Chapitre 1

Introduction

1.1 Objet

Le présent travail sur la compilation conditionnelle en COBOL a pour but d'en identifier les possibilités actuelles afin de trouver des bénéfices que pourrait apporter celle-ci dans les sources codées en COBOL n'exploitant pas encore les fonctionnalités « Object Oriented » telles que disponibles depuis la norme COBOL ISO/IEC 1989:2002 [65] . En exploitant la compilation conditionnelle, peut-on réorganiser ces vieilles sources pour obtenir un code mieux découpé, moins redondant, des fonctionnalités « à la carte » ? … ou, pourquoi pas, approcher un « look and feel » « orienté objet » préfigurant ce que pourrait être le programme converti vers un langage plus moderne ?

1.2 Mise en contexte

Pour certains vieux langages ayant leurs origines au temps où seuls les mainframes existaient, la [complexité | difficulté] de trouver des programmeurs pour prendre le relais de la maintenance des applications programmées dans ces langages (COBOL, PL/1, FORTRAN, ALGOL, …) est un défi rempli d'écueils à surmonter. Si trouver des programmeurs qui y ont acquis de l'expérience au début de leur carrière est possible, ils sont d'une moyenne d'âge dépassant la cinquantaine[1] et n'accompagneront donc plus de longues années les projets existants. Si trouver des programmeurs plus jeunes qui « veulent bien essayer » ne coule pas de source, il ne sera pas facile de faire comprendre la manière d'appréhender un fonctionnement sans risquer de les faire fuir. Et si ces deux premières étapes sont franchies, il faudra encore s'assurer de leur efficacité tant en termes de qualité du code que de rapidité de mise au point du dit code (développement/débogage). Certes, la programmation ne s'effectuant plus sur des terminaux mais sur des stations de travail, il a existé (Intertest de On Line Software, XPEDITER de Application Development System, …) et il existe toujours des outils tels que Eclipse[2], Microsoft Visual Studio[3], IBM

[1] http://techchannel.com/Enterprise/03/2021/business-systems-cobol (2021)
https://www.hackerrank.com/blog/the-inevitable-return-of-cobol/ (2020)
[2] Eclipse : https://www.eclipse.org/ide/
[3] Microsoft Visual Studio : https://visualstudio.microsoft.com/fr/vs/

Debug[4], TADS[5],… qui, en lieu et place de premiers populaires éditeurs (CANDE [45]; l'éditeur de SPF [28]) post-cartes perforées, facilitent le codage et le débogage de codes, mais la tâche ne s'arrête pas là : il faut programmer le plus proprement possible et de manière telle que ce puisse être compris et repris par d'autres développeurs qui à leur tour devront adapter, compléter ou corriger le code au cours de son existence … qui peut s'avérer très longue.

L'index TIOBE[6] calculant la popularité des langages de programmation ne positionne, en janvier 2023, le COBOL qu'en 31eme position avec un rating de 0,33%, 23eme au mois de mars 2023 avec un rating de 0,58% et 20eme au mois de juillet 2023 avec un rating de 0,86%. Calculé sur base du nombre de développeurs, du nombre de cours et du nombre d'éditeurs, est-il réellement représentatif de la popularité d'un langage ? Les compilateurs COBOL sont accompagnés de documentations claires, complètes… les cobolistes sont généralement vieux, expérimentés et préfèrent un échange entre collègues ou un « bon vieux manuel » à une recherche internet ; ils n'activent donc simplement pas les indicateurs pris en compte par TIOBE, est-ce pour cela que COBOL peut être qualifié de peu populaire ? En tout cas, l'index démontre à quel point une carence de développeurs s'annonce, sachant que les estimations[7] du nombre de lignes de code écrites en COBOL avoisineraient 800 milliards, et que le nombre d'entreprises se compte en millier ; qu'une estimation du nombre de transactions journalières supportées par COBOL atteint 3.000.000.000. Des migrations automatisées, des réécritures d'applications s'effectuent, mais ce sont des processus onéreux, risqués et qui prennent du temps, … et en attendant : l'existant doit satisfaire les exigences métiers.

Obtenir un <regroupement des lignes de code> et présenter <des fichiers sources> plus proches de ce que connaissent les récentes générations de programmeurs parait donc intéressant. Il s'agit là du but premier, mais des usages autres pourront être découverts au cours des lectures et seront dans ce cas, à minima mentionnés comme existants et éventuellement transposés en COBOL.

1.3 Méthode

L'état de l'art basé sur la recherche `"conditional compilation" AND cobol` fixera les concepts, clarifiera l'état actuel de l'usage, présentera un historique ciblé de l'évolution et servira de de point de départ à la recherche.

Puisque tous les éditeurs n'ont pas encore opté pour la norme COBOL ISO/IEC 1989:2002 des dialectes (implémentations de la compilation conditionnelle propre par les divers compilateurs cités dans l'historique ciblé) COBOL seront comparés.

La méthodologie utilisée est expérimentale : peut-on, avec un compilateur implémentant la norme COBOL ISO/IEC 1989:2002 proposer des codes mieux factorisés (voire même presque « encapsulés ») grâce à cette première normalisation de la compilation conditionnelle introduite en COBOL ?

MicroFocus Visual Cobol 8.0 qui implémente la norme COBOL ISO/IEC 1989:2002 [65] sera utilisé pour élaborer et proposer des exemples concrets répondant à l'objectif de la recherche.

Les traductions des quelques citations sont les résultats de traductions automatiques avec www.DeepL.com/Translator (version gratuite) confrontées à celles de Microsoft Translator (version Microsoft Office payant) et relues).

[4] IBM Debug for z/OS https://www.ibm.com/products/debug-for-zos
[5] TADS : Test And Debug System, livré avec les compilateurs ALGOL, COBOL, sur mainframe Unisys
[6] https://www.tiobe.com/tiobe-index/
[7] https://www.microfocus.com/en-us/press-room/press-releases/2022/cobol-market-shown-to-be-three-times-larger-than-previously-estimated-in-new-independent-survey

1.4 Cheminement

Pour l'état de l'art il sera recherché et analysé dans la littérature (dl.acm.org, ieee.org, springer.com, sciencedirect.com) les usages du critère **`"conditional compilation" AND cobol`**. Des recherches plus générales (StackOverflow.com, Community.ibm.com, Github.com) de ce même critère seront effectuées pour relever des usages concrets existants exposés par des cobolistes.

Avant d'aborder le sujet de recherche concernant les possibilités d'usage dans l'un : le COBOL, de l'autre : la compilation conditionnelle, quelques notions minimums de COBOL seront mentionnées, et des motivations de l'existence de la compilation conditionnelle pour des langages qui l'utilisent seront recherchées. Ainsi s'en suivront :

- un chapitre de présentation de base du langage COBOL, normalement suffisante pour la compréhension des codes qui seront fournis.
- Un chapitre pour déterminer raisonnablement comment la littérature défini la compilation conditionnelle de manière générale (concepts, finalités, avantages/inconvénients).
  Cela servira de source de réflexion pour la proposition d'usages nouveaux ou repris d'autres langages*, s'ils sont bénéfiquement transposables en COBOL,* de la compilation conditionnelle .
  Ce chapitre sera terminé par un historique basé sur la lecture de publications d'éditeurs de compilateurs de « vieux langages sur mainframe » (ALGOL, FORTRAN, COBOL) et évoquera l'évolution des possibilités de compilations conditionnelles proposées par ces trois langages, ainsi que*, pour COBOL et FORTRAN*, éventuellement définies par une norme. Quelques dialectes COBOL incluant leur, toujours supportée, compilation conditionnelle « spécifique » (comprendre « en dehors de la définition de la norme COBOL ISO/IEC 1989:2002 ») seront présentés.

Le quatrième chapitre, consacré au développement de la recherche, présentera la question de recherche, la méthodologie de recherche, et après avoir comparé des dialectes de compilations conditionnelles antérieures à la norme et proposé quelques usages, il présentera de manière didactique (évolutive et avec une pertinence croissante) l'intérêt pour les programmeurs COBOL, expérimentés ou débutants, d'appréhender et de faire leur l'usage de la compilation conditionnelle. Des codes concrets seront proposés comme fil conducteur. Ils auront été éprouvés avec le MicroFocus Visual Cobol 8.0 pour Visual Studio (licence étudiant, gratuite 1 an), pour démontrer des usages répondant à l'objectif de ce travail.

Chapitre 2

Etat de l'art

La recherche s'est basée sur le critère relativement large **"conditional compilation" & cobol** ne limitant en rien le sujet de recherche à un dialecte de COBOL, à une version précise de la norme ou à un domaine d'application spécifique.

La première source d'articles fut « The ACM Guide to Computing Literature » https://dl.acm.org/ paramètre &expand=all en fin d'URL). Elle a fourni 33 résultats, dont la grande majorité fut écartée car non pertinente : simple évocation du langage COBOL, parfois même seulement dans la mention d'une référence, ou pour nommer un vieux langage, ou l'utiliser dans des comparaisons.

La deuxieme source https://www.ieee.org/ n'a donné qu'un seul résultat où les deux concepts n'étaient présents simultanément que parce que sujets distincts de différents workshops d'un même congrès.

La troisième source https://link.springer.com/search a fourni 27 résultats qui, comme pour dl.acm.org, ne furent, pour la quasi-totalité, pas probants.

Les 15 résultats fournis en interrogeant https://www.sciencedirect.com/ recouvraient certains obtenus précédemment, ou n'étaient pas pertinents.

Le site d'entraide https://stackoverflow.com interrogé pour **"conditional compilation" cobol** a fourni trois résultats dont un non pertinent, un deuxième faisant référence au travail du préprocesseur lorsqu'il traite la directive COPY REPLACING… Seule la troisième[8] de mars 2022 répondait à une question de syntaxe pour l'usage de la compilation conditionnelle en COBOL afin qu'une même source puisse être compilée sur un système z/OS et un IBM i exploitant aussi la compilation conditionnelle pour établir une Software Product Line spécialisée « plateforme ».

Sur le site https://github.com les quelques issues (41) et discussions (7) répondant à la recherche **"conditional compilation" cobol** concernent IBM et émanent principalement d'une

[8] https://stackoverflow.com/questions/71692473/single-source-for-ibm-i-and-z-os/71693293#71693293

même personne (Denis FALLAI), et c'est lui toujours qui sur le site https://community.ibm.com/ interroge la communauté coboliste[9] IBM pour connaitre les avis sur l'usage de la compilation conditionnelle en COBOL, Il n'a reçu qu'une seule réponse, mais enthousiaste, sur l'usage de cette technique.

La première conclusion qui ressort est le peu d'études/discussions dédiées à la combinaison des sujets 'compilation conditionnelle' et 'cobol'. La non-popularité du COBOL, l'image qu'il véhicule (E.W. Dijkstra [36]), l'apparition tardive de la compilation conditionnelle dans la norme du langage et l'implémentation de celle-ci bien plus tard encore dans les compilateurs COBOL, n'y sont certainement pas étrangers. En 2012, N. Volanschi [89] rapportait *qu'en COBOL il n'y a pas de facilités de compilation conditionnelle*. Le livre « Beginning COBOL for Programmers » de M. Coughlan [94] paru en 2014 développe toutes les fonctionnalités Orienté Objet apparues en COBOL depuis la norme 2002, mais n'évoque pas la compilation conditionnelle. La *norme existe pourtant depuis 2002*, mais ses fonctionnalités <compilation conditionnelle> *ne furent disponibles dans des compilateurs que bien plus tard* (IBM Enterprise Cobol for z/OS 6.2 en 2017[10] , Micro Focus Visual Cobol 5.0 en 2019[11]). Tel que ce sera évoqué plus loin, des compilateurs COBOL supportant de la compilation conditionnelle existaient pourtant bel et bien, mais il s'agissait d'implémentations qui leur étaient propres. Ces compilations conditionnelles antérieures à la norme semblent n'avoir jamais suscité d'intérêt de recherche liée à leur usage en COBOL.

[9] https://community.ibm.com/community/user/ibmz-and-linuxone/discussion/cobol-conditional-compilation-when-to-use-when-not-to-use

[10] https://www.ibm.com/resources/publications/OutputPubsDetails?PubID=SC27871301

[11] https://www.microfocus.com/documentation/visual-cobol/vc50/VS2019/GUID-2C559AC9-6421-4011-9EA5-1751CB768548.html

Chapitre 3

Le Cobol en bref

Avant de l'utiliser dans le chapitre de recherche, il est important d'évoquer quelques notions de COBOL. Ces notions sont la synthèse du cours de M. Prévost, suivi à l'ISI en 1987-1988, maintenues à jours par la lecture de manuels de référence du langage accompagnant les releases de compilateurs (IBM [111], UNISYS [46], MicroFocus[en ligne]) ainsi que du livre « Beginning COBOL for Programmers » [94]. Sachant qu'en 2001 il est fait mention de plus de 300 dialectes COBOL [64] les notions présentées restent toutefois dans la norme du COBOL.
Il s'agit d'une description du COBOL sans usage de l'orienté objet (- *la possibilité de l'OO en COBOL a été introduit par la norme 2002 [65]* -) tel qu'on le trouve dans les vieux programmes « **legacy** » (hérités) et que connaissent les vieux programmeurs. « *Les systèmes logiciels hérités sont des programmes essentiels au fonctionnement des entreprises, mais qui ont été développés il y a des années en utilisant de premiers langages de programmation tels que Cobol et Fortran. Ces programmes ont été maintenus pendant de nombreuses années par des centaines de programmeurs, et bien que de nombreux changements aient été apportés au logiciel, la documentation qui les accompagne peut ne pas être à jour*. » [58].

3.1 La structure d'un programme

Un programme COBOL est composé de quatre divisions ordonnées.

- L'**`IDENTIFICATION DIVISION.`**
  Cette division obligatoire et au nom suffisamment parlant, doit contenir au minimum l'entrée **`PROGRAM-ID. Nom-Du-Programme.`**
  D'autres entrées telles que **`AUTHOR. Nom-du-programmeur.`** sont facultatives.
- L'**`ENVIRONMENT DIVISION.`**
  se compose de deux sections optionnelles :
  La **`CONFIGUARTION SECTION`**. permet quelques configurations du compilateur. Si le **`COMPUTER-NAME`** est sans effet, l'option associée **`WITH DEBUGGING MODE`** spécifie la nécessité de compiler les lignes préfixées d'un **`D`** en colonne 7. D'autres options telles que la spécification du symbole décimal, du symbole monétaire, d'un alphabet spécifique, … existent, et bien

qu'il s'agisse de « directives de compilations » ne sont pas étudiées ici en tant que telles puisqu'elles n'agissent que sur un ou des caractères en tant que « données ».

L'**INPUT-OUTPUT SECTION.** peut contenir deux paragraphes :

- le **FILE-CONTROL.** qui établira le lien entre la vision externe d'un fichier (son nom connu en dehors du programme, ses caractéristiques,…) et sa connaissance à l'intérieur du programme (son nom symbolique utilisé dans le programme).
- L'**I-O-CONTROL** décrit au système la fréquence des points des « checkpoint » lorsque le programme traite des fichiers.

- La **DATA DIVISION.**

  Il s'agit ici d'une division optionnelle qui sert à localiser les données : tout nom symbolique identifiant une adresse mémoire où localiser une donnée doit avoir été porté à la connaissance du compilateur dans cette division. Elle peut contenir diverses sections, toutes optionnelles, qui caractérisent les localisations (buffer, externe, interne,…).

  La **FILE SECTION.** qui va, pour chaque fichier, contenir la description du ou des formats d'enregistrement que ce fichier peut contenir. On pourra ainsi interpréter l'enregistrement lu, alors qu'il sera toujours positionné dans le buffer, ou y préparer l'enregistrement qui sera envoyé vers le support.

  La **WORKING-STORAGE SECTION.** sert à définir (en type, en taille, en occurrence) toutes les variables et leurs caractéristiques (format, valeur initiale), qu'elles soient élémentaires, structurées, en tableau (ordonné ou non), qui seront propres à l'exécutable produit ; le contenu est initialisé au chargement en mémoire de l'exécutable, est persistant, et peut être partagé entre plusieurs threads.

  La **LOCAL-STORAGE SECTION.** est semblable à la WORKING-STORAGE SECTION, sauf que chaque sollicitation de l'exécutable, même s'il est déjà résidant en mémoire, recharge et initialise une copie des déclarations de cette section, et que chaque threads, s'il y en a plusieurs, dispose de sa propre copie.

  La **LINKAGE SECTION.** décrit les éventuels paramètres échangés entre le système/programme appelant et l'exécutable lorsqu'il est sollicité.

  La **COMMUNICATION SECTION.** permet de décrire, si utilisée, un moyen de communication basé sur des messages échangés entre deux programmes.

  La **REPORT SECTION.** permet de spécifier les caractéristiques d'un rapport (nombre de lignes par page, page d'en-tête ou de fin de rapport, haut et pied de page, critères et présentation de ruptures, totaux intermédiaires,…) que COBOL prendra en charge, débarrassant le programmeur de devoir cumuler des valeurs, évaluer des changements, compter ses lignes écrites,…

- La **PROCEDURE DIVISION.**

  Cette division contient toutes les instructions programmées pour réaliser les fonctionnalités du programme, y inclus d'éventuelles groupes d'instructions aidant au débogage du programme.

  S'il existe, on trouvera le bloc de **DECLARATIVES** localisé tout au début de la procédure division. Une déclarative est une section dont l'exécution n'est pas explicitement demandée mais qui est automatiquement exécutée lorsque survient l'évènement pour lequel elle a été programmée, évènement qui est spécifié dans la clause **USE** de ladite section.

  A la suite des déclaratives, on retrouve, éventuellement regroupés en **SECTION** le ou les paragraphe(s) pouvant être exécuté(s).

  Un paragraphe est le plus petit groupe d'instruction(s) que l'on peut demander d'exécuter ; il est constitué d'au moins une **SENTENCE**, elle-même constituée d'au moins une **STATEMENT**, c'est-à-dire d'une action identifiable par un « verbe » (le mot réservé IF est aussi considéré comme un verbe).

Il y a diverses manières d'obtenir l'exécution d'un paragraphe :

+ soit parce qu'il est le tout premier de la **PROCEDURE DIVISION** et non inclus dans une déclarative
+ soit parce qu'il est le tout premier suivant un point d'entrée alternatif
+ soit par invocation explicite **PERFORM** nom-du-paragraphe
+ soit par branchement **GO TO** nom-du-paragraphe
+ soit parce qu'on y arrive en cascade :
  - il appartient à la section dont on a demandé l'exécution,
  - ou il est dans la séquence des paragraphes allant de G à K inclus apparaissant dans une instruction **PERFORM G THRU K** où G et K sont des noms de paragraphes et G précède dans le code du programme

  Dans une même **PROCEDURE DIVISION**, plusieurs paragraphes peuvent porter le même nom pour autant qu'ils appartiennent à des sections différentes ; pour l'invocation de l'un d'entre eux il faudra donc le qualifier en spécifiant la section à laquelle il appartient.

Dans la globalité du code (de la première à la dernière ligne), on peut insérer :

- Des lignes de commentaires, identifiables par la présence d'un caractère ***** en colonne 7 ;
- des directives de compilations propres à l'éditeur du compilateur, ainsi que des directives de compilations conditionnelles telles qu'évoquées précédemment et supportées par l'éditeur.

3.2 Les déclarations de variables

Une autre caractéristique du COBOL est son système de réservation mémoire basé sur des n° de niveaux. On peut déclarer des variables non-structurées, de niveau 01 ou 77, décrites chacune par un format 'picture' déterminant ce que la zone va contenir (un ou des caractères, un nombre entier ou décimal détaillant sa précision et sa représentation interne, …) . On peut déclarer des structures basées sur des imbrications de niveaux, le premier étant obligatoirement 01 et le plus bas 49. Quel que soit le niveau, une déclaration de variable non-décomposée en sous-niveaux doit disposer du 'picture' la décrivant. Si une déclaration à un niveau n ($0 < n <= 48$) est décomposée en sous-déclarations, elles seront toutes d'un (obligatoirement) même n° de sous-niveau $n+j$ où $0 < j <= 49-n$; cette déclaration de niveau n ne peut pas contenir de picture la décrivant, elle sera automatiquement considérée comme une chaine de caractères concaténant tous les octets des sous-déclarations la composant.

3.3 La directive COPY

Bien qu'usuellement présentée comme une « instruction », et clairement définie dans la norme COBOL, **COPY** est à destination du compilateur une directive de manipulation de la source avant « traduction » en exécutable. Elle lui spécifie d'aller chercher et, *avec* ou *sans manipulation (clause* **replacing***),* d'inclure/incorporer au texte source, en lieu et place de la directive, le contenu du fichier nommé. Étant donné que le texte résultant sera passé au compilateur, il est impératif que le résultat après incorporation respecte, à la place où il est trouvé, la syntaxe attendue par le compilateur. La norme 2002 définit de nombreuses règles liées à cette directive dont deux sont à rappeler pour ce travail :

- Il faut supporter au minimum 5 niveaux de **COPY** imbriqués ;
- Il ne peut pas y avoir de récursivité directe ou indirecte de **COPY**.

3.4 Un squelette de programme

La Fig. 1 ci-dessous montre un squelette de programme, compilable et exécutable. Ce squelette aurait pu être plus épuré encore puisque, mais ce n'est normalement pas le but d'un programme, on ne lui demande rien à faire.

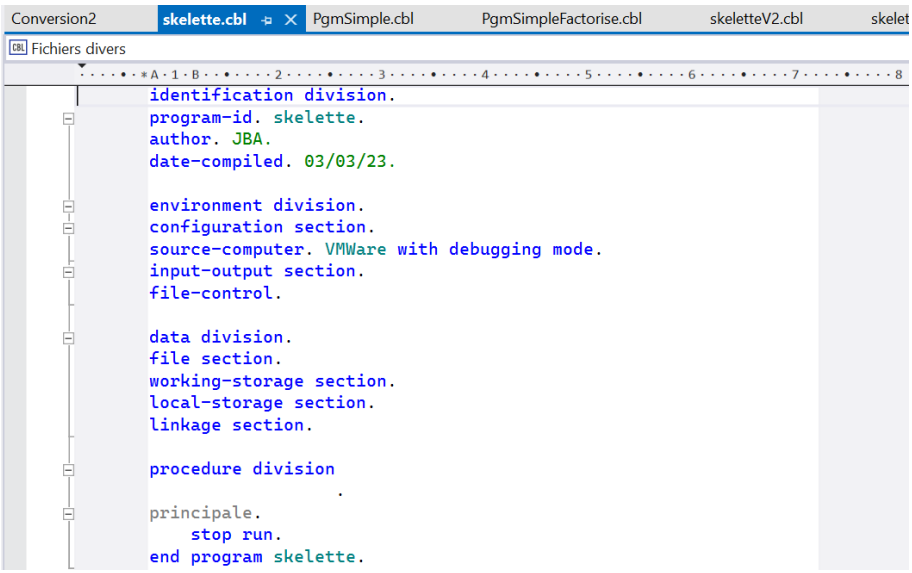


Fig. 1 Squelette de programme Cobol

La Fig. 2 suivante montre un programme très simple (et sans intérêt autre qu'illustratif) : pour un fichier donné : on doit passer un certain nombre de lignes, puis compter le nombre de lignes restante s'il en reste.

On constate la répartition/ventilation dans trois divisions d'informations « stables » concernant le fichier : - l'association entre son nom symbolique interne au programme et son nom physique - , - la description de sa structure - , - son processus de lecture -. Si plusieurs programmes ont à traiter ce même fichier pour des fonctionnalités différentes, ces informations « stables » devront être répétées dans chacun des programmes. Outre que c'est fastidieux, c'est constitutif d'une dette technique car si la structure du fichier change, ou qu'au lieu d'être séquentiel il devient indexé, chaque programme sera à modifier.

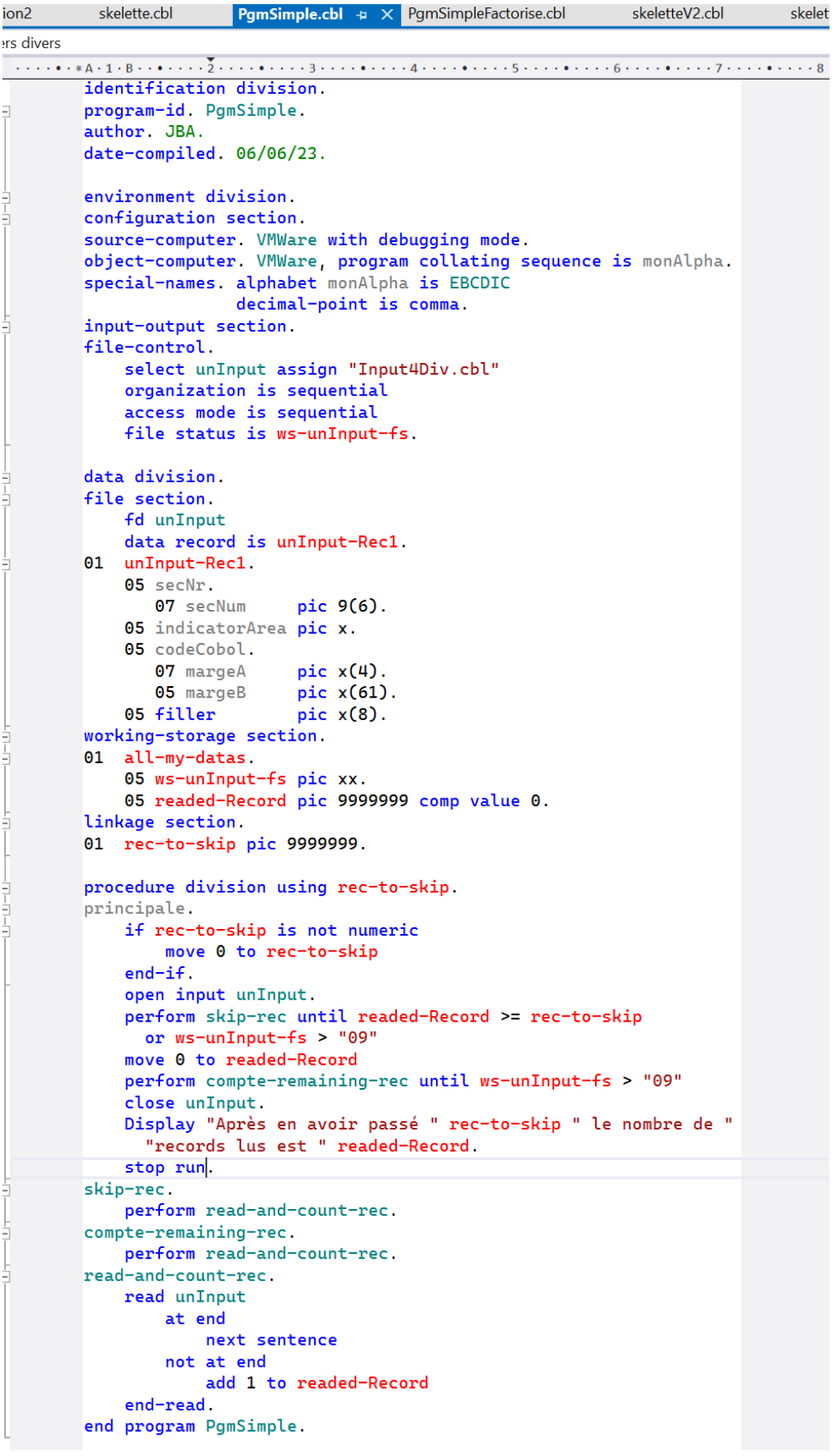


Fig. 2 Exemple d'un très simple programme COBOL

Chapitre 4

La compilation conditionnelle

4.1 Qu'est-elle ?

Sans réellement le réaliser, Monsieur ou Madame Toutlemonde a sans doute déjà été face à un processus effectuant une compilation conditionnelle : l'installation et la configuration d'un système d'exploitation par exemple tiendra compte des caractéristiques de la machine pour en exploiter les capacités et en améliorer l'efficacité. La détection du matériel est maintenant plus automatisée, mais certains se souviendront de questions qui, à l'installation de Windows, étaient posées concernant le matériel. Les utilisateurs de Linux sont sans doute plus conscients de la chose avec la compilation du noyau, …

La compilation n'est pas toujours une traduction univoque de tout un fichier ou ensemble de fichiers constituant un code source ; il faut entendre par là que pour un même texte source à compiler à destination d'une même machine :

- Un même compilateur pourra produire des résultats distincts. Le texte du programme est une chose, mais les finalités attendues au moyen de directives précisées au compilateur en sont une autre. C'est notamment évident lorsque la même source est traduite par le même compilateur tantôt en lui demandant d'optimiser la vitesse, tantôt l'espace mémoire, tantôt d'être capable de fournir un maximum d'informations contextuelles si le programme rencontre un problème forçant son arrêt brutal.
- Des compilateurs d'éditeurs distincts pourront produire des exécutables distincts dont, on peut l'espérer, les exécutions produiront les mêmes résultats, … mais l'un plus rapidement que l'autre qui, lui, consommera peut-être moins d'espace mémoire.

Pilotée de l'extérieur à son lancement, le déroulement de la compilation peut, entre autres, être orienté par les directives de compilation, mais pour certains langages, ou certains éditeurs de compilateurs, également par des instructions évaluées à la volée, de sélection ou d'exclusion de morceaux de code de la globalité du code effectivement traduite par le compilateur. « *La compilation conditionnelle est une facilité qui offre au programmeur l'option de faire compiler ou non des sections de code*

en fonction de conditions spécifiques » [60] en défini le principe, alors que pour son exploitation : « *La compilation conditionnelle permet, en transmettant des valeurs au compilateur, de manipuler le processus d'analyse du compilateur.* » [88] C'est cette partie qui, pour le COBOL, est étudiée dans le présent mémoire.

C'est,
- anticipées par l'auteur de [morceaux de] code à compiler ciblés distincts et
- détectées à la volée par le compilateur dans la globalité de la source traitée,

l'exploitation/l'interprétation d'instructions et conditions appelées <directives>
qui permettront de déterminer et appliquer
un des scenarii prévus pour obtenir un exécutable « customisé » ;
et ça peut être un moyen de « design » de texte source.

On constate qu'il y a donc un travail préparatoire de ce que l'on donnera au compilateur : Vouloir faire usage de la compilation conditionnelle consiste en l'exploitation d'une technique d'écriture du texte source qui a pour but de préparer, dans un même code constitué d'un fichier ou d'un ensemble de fichiers, plusieurs scénarii. La version de ce qui constitue le scénario à compiler=<le texte du programme à traduire en exécutable> parmi les possibilités est déterminée à la volée par le processus de compilation qui interprètera des paramètres et conditions qu'on lui adresse. « *Quand la compilation conditionnelle est utilisée dans un programme, ce qui est compilé est seulement une des nombreuses versions possibles du code source, chacune déterminée par une affectation particulière des variables de compilation conditionnelle* » [59].

« *Le texte du programme est créé sur base du texte source* » [7]. Donc pour obtenir le scénario voulu, des fichiers de code qui constitueront le texte source, « *On peut inclure ou exclure des parties du programme en fonction de différentes conditions* (déterminées au moyen de directives d'affectation ou d'évaluation) *Une directive est une ligne dans le programme qui commence par '#'* » [76]. Le caractère '#' est utilisé en C, mais en COBOL il s'agit de la paire '>>' selon la norme ; certains dialectes ayant prévu la compilation conditionnelle dans une syntaxe propre bien avant la norme utilisent d'autres moyens d'identification, le caractère '$' par exemple.

Que peut-on faire de la technique 'compilation conditionnelle' qui vient d'être expliquée ?

« *La compilation conditionnelle est un mécanisme utilisé pour implémenter de la variabilité lors du processus de compilation … La compilation conditionnelle est un mécanisme courant pour mettre en œuvre une ligne de produits logiciels*. » [82] Il semble donc intéressant de regarder si nous aurons à faire à des lignes de produits logiciels.

« *La ligne de produits logiciels (SPL) est un paradigme de développement qui vise à créer des systèmes logiciels variables* » M.V. Couto et al. [86] qui citent Clements [63] Bien que l'introduction de la compilation conditionnelle faite dans les codes exemples n'aie pas pour objectif de produire et distribuer différents variants des programmes pour qu'ils tournent en parallèle, le texte source permettra théoriquement bien de produire plusieurs textes de programme embrassant ainsi la définition de SPL (Software Product Line). A l'exception de l'activation/désactivation des instructions de débogages produisant éventuellement des variants distincts tournant en parallèle (chaque programmeur - sa version), les autres fonctionnalités soumises à la compilation conditionnelle ne le sont que pour faciliter la programmation, factoriser de manière centralisée et réutiliser facilement des sources. Factoriser les clones est recommandé par Koschke [75]. Et si besoin est pour le lecteur, Fenske et al. expliquent comment passer de produits clonés vers une ligne de produits [98].

4.2 Apparition de la compilation conditionnelle dans quelques langages sur mainframe.

Telles qu'on les trouve dans le document C28-6571-1 IBM Operating System/360 PL/1: Language Specifications, p131, Chapter 9 : Program Modification paru en juillet 1965 [7]; on constate l'existence d'instructions définies pour le préprocesseur (elles commencent par %) afin qu'il puisse, sur base du texte source et du résultat de ces instructions, constituer le texte du programme à compiler. Il s'agit de la définition de « macro » : un nom donné à un ensemble d'instructions qui, automatiquement et par le précompilateur, seront insérées dans le code à chaque fois que le nom de la macro sera invoqué. Le concept de « macro » est utilisé dans bien d'autres langages (Assembleur, C, Fortran,…). Il permet d'obtenir pour un même texte source invoquant une macro, en ne changeant que la définition de la macro, des textes de programme bien différents. Avec le mot-clé DEFINE [41], le langage COBOL a, à partir de la version 1960, aussi disposé de la possibilité de définir des macros, concept qui fut, faute d'être utilisé, retiré du langage dès la version COBOL-68. Notons que si les macros offrent des facilités certaines, elles ne font pas partie de présent travail puisqu'antérieurement à la compilation on passe par une étape de « modification du code » de la macro, le texte source ne contient donc qu'un seul scénario. NPL, pour New Programming Language [5], qui devint PL/1, prévoyait pourtant la possibilité de compilation conditionnelle telle que définie et décrite précédemment, mais le manuel PL/1 d'IBM [7] n'en fait aucune mention.

```
COMMENT TO ADJUST THE PRIORITY, CORE ESTIMATE, AND STACK SIZE            00007200
        OF LIBMAIN/DISK, SEE SEQUENCE NUMBER 39599000;                   00007210
 LABEL GOGOGO,NORMALERROR,P2BUSY,TIMER,EXTERNAL,INQUEST,                 00008000
       PROCSWIT,P2FAKE,KEYBOARDREQUEST,RETURN,COMINIT,MEMORYPARITY  %WF  00009000
       ;                                                                 00010000
 DEFINE MARK = "?"#;                                                     00011000
 DEFINE LEFTARROW="←"#;                                                 %DS00011500
 DEFINE SHEETMAX=9#;                                                     00012000
DEFINE GETUSERDISK(GETUSERDISK1)=PETUSERDISK(GETUSERDISK1,0)#;%          00012001
 $ SET OMIT = NOT(DUMP OR DEBUGGING)                                     00012159
    DEFINE DUMPNOW(DUMPNOW1) =                                           00012160
       DUMPCORE(DUMPNOW1&(SPACE(22)+1)[15:33:15])#;                      00012165
 $ POP OMIT                                                              00012166
    INTEGER RRRMECH=@201;%                                               00013000
DEFINE                                                                   00013850
 $ SET OMIT = AUXMEM                                                     00013860
       SPACESTACKSIZE = 80#;                                             00013870
 $ SET OMIT = NOT AUXMEM                                                 00013880
       SPACESTACKSIZE = 100#;                                            00013890
 $ POP OMIT OMIT                                                         00013900
 SAVE REAL PROCEDURE GETSPACE(SIZE,TYPE,SAVEF);                          00014000
                     VALUE SIZE,TYPE,SAVEF;%                             00015000
                     INTEGER SIZE,TYPE;%                                 00016000
                     BOOLEAN SAVEF;          FORWARD;%                   00017000
SAVE REAL PROCEDURE WAITIO(IOD,MASK,U);%                                 00018000
                    VALUE IOD,MASK,U; REAL IOD,MASK,U; FORWARD;%         00019000
```

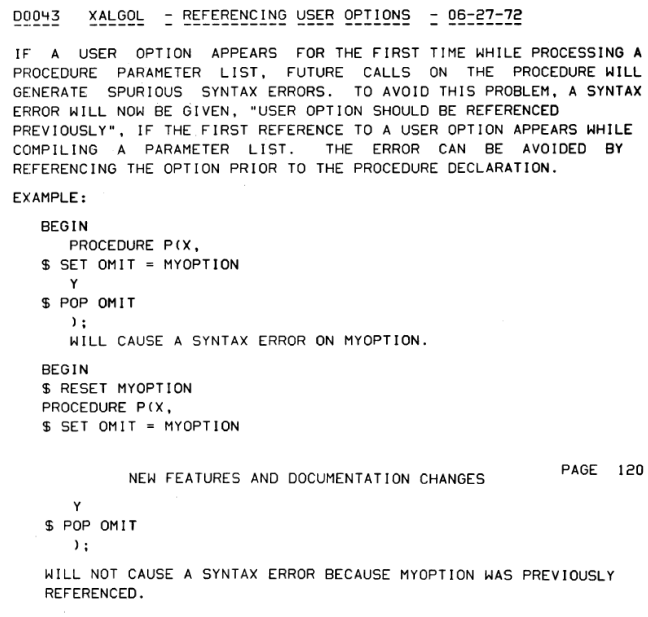
Fig. 3 Usage de compilation conditionnelle dans du code écrit en ALGOL Extrait du Master Control Program de Burroughs [15]*, utilisant la compilation conditionnelle pour déterminer un spacestacksize disponible dépendant de la configuration*

Hormis pour des compilateurs de langages destinés à la programmation de systèmes multi-architectures (tel l'ALGOL ou XALGOL de Burroughs, utilisés notamment pour l'écriture du système d'exploitation MCP évoluant avec les capacités de ses machines) rien de « certain » n'a été trouvé quant à l'apparition ou la motivation de la compilation conditionnelle ou de ce qui produit un résultat équivalent. Pour en évaluer l'évolution du concept, il a donc fallu se plonger dans des manuels accompagnant les compilateurs pour quelques éditeurs. Les versions des compilateurs des langages Fortran (langage dédié au calcul) et Cobol (langage dédié à la gestion de données business) ont été auscultés plus profondément.

- 1957 – Définissant le langage FORTRAN à la suite du projet initié en 1954, la publication « THE FORTRAN AUTOMATIC CODING SYSTEM » [1] n'évoque aucune compilation conditionnelle ou instruction de debugging qui, plus tard, pourrait y amener.
- Avril 1960 – Le « plus vieux » manuel Fortran retrouvé, publié par IBM et intitulé « Systems manual for 704 FORTRAN and 709 FORTRAN » [3] ne laisse apparaitre aucun type de compilation conditionnelle.
- Avril 1960 – Publié par le « Department of Defense » des Etats-Unis, le « Report to CONFERENCE on DATA SYSTEMS LANGUAGES Including INITIAL SPECIFICATIONS for a COMMON

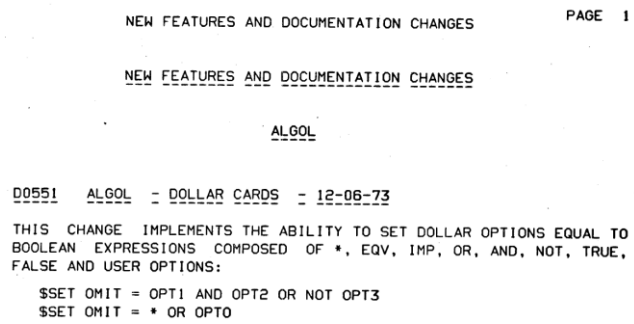
BUSINESS ORIENTED LANGUAGE (COBOL) for Programming Electronic Digital Computers » [2] n'évoque non plus aucune notion se rapprochant de la compilation conditionnelle.

- 1964 – « IBM System/360 FORTRAN IV Language » [9] File No. S360-25 Form C28-6515-6 contient des prémisses de la compilation conditionnelle. L'apparition de « Debug Facility » (p.113) concrétisé sous forme d'instructions regroupées en un ou plusieurs « debug packets » situés juste avant l'instruction END du programme. C'est une intervention manuelle (ajout ou retrait de ces paquets) qui en conditionnera la compilation. Selon ce document (p.120) il s'agit d'une extension IBM au Basic FORTRAN IV. Dans son article sur FORTRAN IV, B.M. Leavenworth [4] ne fait en tout cas aucune mention de cette extension au langage malgré l'initiative introduite par IBM.
- Avril 1965 – La publication « ECMA STANDARD on FORTRAN » [6] ne spécifie rien se rapprochant de la compilation conditionnelle.
- Septembre 1970 – Le « COBOL REFERENCE MANUAL » [11] pour systèmes B2500 et B3500 de Burroughs précise que certains aspects produits par la compilation peuvent être contrôlés par l'introduction de cartes de contrôle, identifiables par la présence du caractère $ en colonne 7, et qui permettent de modifier des directives de compilation, mais pas encore d'inclure ou exclure du code source du texte du programme.
- Avril 1974 – Dans sa publication 5000763 (2.6) qui est une note documentaire [16] concernant les améliorations logicielles pour son système B6700, Burroughs annonce plusieurs facilités additionnelles pour les cartes dollar, dont la directive OMIT (p.36-37) qui est à l'origine du sujet du présent mémoire, **implémentant pleinement de la compilation conditionnelle** en permettant de prévoir plusieurs scénarii de compilation en COBOL.
  Cobol a hérité de ce qui était déjà d'application en 1972 dans le compilateur XALGOL de l'éditeur



```
D0043   XALGOL  - REFERENCING USER OPTIONS  - 06-27-72

IF A USER OPTION APPEARS FOR THE FIRST TIME WHILE PROCESSING A
PROCEDURE PARAMETER LIST, FUTURE CALLS ON THE PROCEDURE WILL
GENERATE SPURIOUS SYNTAX ERRORS. TO AVOID THIS PROBLEM, A SYNTAX
ERROR WILL NOW BE GIVEN, "USER OPTION SHOULD BE REFERENCED
PREVIOUSLY", IF THE FIRST REFERENCE TO A USER OPTION APPEARS WHILE
COMPILING A PARAMETER LIST.  THE ERROR CAN BE AVOIDED BY
REFERENCING THE OPTION PRIOR TO THE PROCEDURE DECLARATION.

EXAMPLE:

   BEGIN
      PROCEDURE P(X,
   $ SET OMIT = MYOPTION
      Y
   $ POP OMIT
      );
      WILL CAUSE A SYNTAX ERROR ON MYOPTION.

   BEGIN
   $ RESET MYOPTION
   PROCEDURE P(X,
   $ SET OMIT = MYOPTION

            NEW FEATURES AND DOCUMENTATION CHANGES          PAGE  120

      Y
   $ POP OMIT
      );

   WILL NOT CAUSE A SYNTAX ERROR BECAUSE MYOPTION WAS PREVIOUSLY
   REFERENCED.
```

Fig. 4 Première trace trouvée de compilation conditionnelle B6700_SoftwareNotes_II.3_Oct72 [14]. *(p.119)*



NEW FEATURES AND DOCUMENTATION CHANGES PAGE 1

NEW FEATURES AND DOCUMENTATION CHANGES

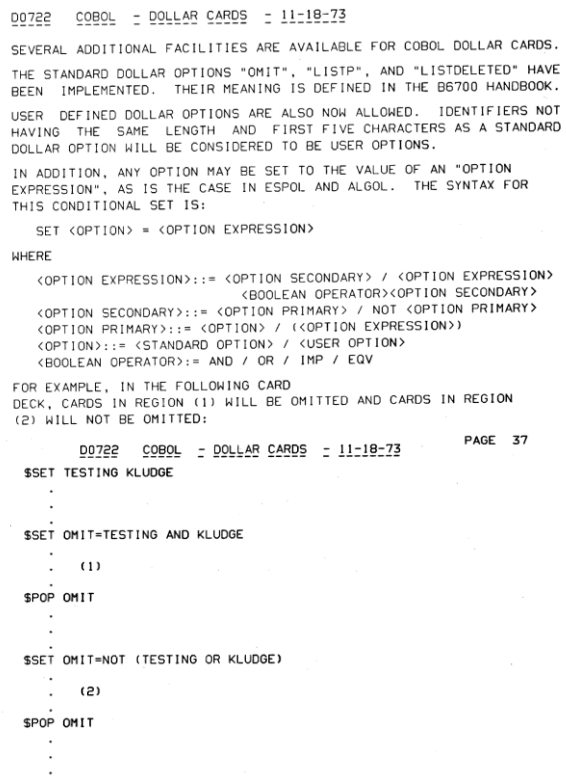
ALGOL

D0551 ALGOL - DOLLAR CARDS - 12-06-73

THIS CHANGE IMPLEMENTS THE ABILITY TO SET DOLLAR OPTIONS EQUAL TO BOOLEAN EXPRESSIONS COMPOSED OF *, EQV, IMP, OR, AND, NOT, TRUE, FALSE AND USER OPTIONS:

```
$SET OMIT = OPT1 AND OPT2 OR NOT OPT3
$SET OMIT = * OR OPT0
```

Fig. 5 Possibilité d'expression conditionnelle dans l'ALGOL de Burroughs 5000763_B6700_Software_Notes_2.6_Apr74 [16]



D0722 COBOL - DOLLAR CARDS - 11-18-73

SEVERAL ADDITIONAL FACILITIES ARE AVAILABLE FOR COBOL DOLLAR CARDS.

THE STANDARD DOLLAR OPTIONS "OMIT", "LISTP", AND "LISTDELETED" HAVE BEEN IMPLEMENTED. THEIR MEANING IS DEFINED IN THE B6700 HANDBOOK.

USER DEFINED DOLLAR OPTIONS ARE ALSO NOW ALLOWED. IDENTIFIERS NOT HAVING THE SAME LENGTH AND FIRST FIVE CHARACTERS AS A STANDARD DOLLAR OPTION WILL BE CONSIDERED TO BE USER OPTIONS.

IN ADDITION, ANY OPTION MAY BE SET TO THE VALUE OF AN "OPTION EXPRESSION", AS IS THE CASE IN ESPOL AND ALGOL. THE SYNTAX FOR THIS CONDITIONAL SET IS:

```
   SET <OPTION> = <OPTION EXPRESSION>
```

WHERE

```
   <OPTION EXPRESSION>::= <OPTION SECONDARY> / <OPTION EXPRESSION>
                          <BOOLEAN OPERATOR><OPTION SECONDARY>
   <OPTION SECONDARY>::= <OPTION PRIMARY> / NOT <OPTION PRIMARY>
   <OPTION PRIMARY>::= <OPTION> / (<OPTION EXPRESSION>)
   <OPTION>::= <STANDARD OPTION> / <USER OPTION>
   <BOOLEAN OPERATOR>:= AND / OR / IMP / EQV
```

FOR EXAMPLE, IN THE FOLLOWING CARD DECK, CARDS IN REGION (1) WILL BE OMITTED AND CARDS IN REGION (2) WILL NOT BE OMITTED:

D0722 COBOL - DOLLAR CARDS - 11-18-73 PAGE 37

```
$SET TESTING KLUDGE
   .
   .
   .
$SET OMIT=TESTING AND KLUDGE
   .
   .   (1)
   .
$POP OMIT
   .
   .
   .
$SET OMIT=NOT (TESTING OR KLUDGE)
   .
   .   (2)
   .
$POP OMIT
   .
   .
   .
```

Fig. 6 Burroughs introduit la compilation conditionnelle à son COBOL '74 5000763_B6700_Software_Notes_2.6_Apr74 [16]

- Mai 1975 - George N. BAIRD expose les possibilités standardisées de debugging définies en COBOL-74 ; son article « Program debugging using COBOL'74 » [18] évoque, sans en citer, l'existence d'initiatives d'éditeurs. Rappelant que jusqu'alors, seule l'instruction DISPLAY avait été définie, il attire l'attention sur les risques d'avoir recourt à l'ajout/suppression de telles phrases qui pourraient résulter en un disfonctionnement du programme par une altération involontaire de sa logique. Bien qu'apparaissent dans la norme les instructions TRACE et EXHIBIT, la grande avancée dans la norme du COBOL'74 (X3.23-1974) [19], est l'apparition de l'option **WITH DEBUGGING MODE**. Activé/désactivé à l'initiative du programmeur, ce véritable « flag » de compilation conditionnelle est applicable :
  - Aux lignes de debugging ;
    ces lignes reconnaissables par la présence d'un caractère '**D**' en colonne 7 sont des instructions COBOL qui seront compilées si l'option **WITH DEBUGGING MODE** est

activée, ou qui seront considérées comme des commentaires et non converties en code exécutable en l'absence de l'option **WITH DEBUGGING MODE** ;

- aux directives spécifiques **USE FOR DEBUGGING** ;
  à l'instar des lignes de debugging, les sections déclaratives marquées **USE FOR DEBUGGING** seront, ou non, compilées en cas d'invocation, ou non, du **WITH DEBUGGING MODE**.

Notons que, comme mentionné supra en parlant des ajouts/suppressions de **DISPLAY**, l'activation/désactivation de **WITH DEBUGGING MODE** risque d'altérer la logique du programme puisqu'il change le texte du programme à la base de l'exécutable produit.

Relevons également que ce Debug Facility apparu avec la norme COBOL'74 (X3.23-1974), avait, comme le mentionnent J. Sammet & J Garfunkel [41] en 1985, été marquée « pour suppression » à partir d'une norme suivante.

- Novembre 1976 – IBM décrit l'incorporation des prescriptions du COBOL'74 dans la logique de son compilateur OS/VS COBOL Rel 2 et en détaille le fonctionnement dans sa publication LY28-6486-2 [22].
- 1977 – Le Basic COBOL de Sperry-UNIVAC n'est, dans sa publication UP-8057 Rev.2 [24], pas encore conforme à la norme du COBOL'74 . Cette version en propose bien les instructions **TRACE** et **EXHIBIT**, mais ne dispose pas du **WITH DEBUGGING MODE**. Par contre on y retrouve la notion de debugging packets comme présente dans le FORTRAN IV pour IBM Système/360, et fonctionnant suivant un principe tout à fait similaire.
- Avril 1978 – La publication de la norme X3.9-1978 [27]définissant le Fortran-77 ne fait mention d'aucune spécificité de compilation conditionnelle, ni de debugging qui aurait pu y amener.
- Janvier 1979 – Le manuel (AA-C985A-TE) pour le COBOL-74 pour Vax-11 [29] présenté par digital n'implémente aucune des facilités de debugging -conditionnel- pourtant apparues 5 ans plus tôt dans la norme du COBOL-74.
- Juillet 1980 - le manuel de référence 92060-90023 de Fortran IV pour le compilateur HP [33] propose l'usage du 'D' en colonne 1 pour identifier des instructions à ne traduire en code exécutable que si la demande du mode debug est précisé lors de la compilation (Appendice J, p J-2 et J-3).
- Aout 1983 – On retrouve dans le manuel GC-26-3857-3 d'IBM [38] décrivant son VS COBOL les possibilités du mode debug (incluant la compilation conditionnelle **WITH DEBUGGING MODE**) introduites par le COBOL'74, mais aussi, p.405, la technique des « debugging packets » telle qu'apparue en 1964 dans son compilateur FORTRAN IV (voir supra).
- 1991 – la publication ISO/IEC 1539 : 1991 (E) [47] décrivant la norme du FORTRAN-90 ne mentionne rien.
- 1991 – les « CCR » (Compiler Control Record) de contrôle du processus de compilation conditionnelle SET, RESET, POP (p17-9) et OMIT (p17-32) sont incorporées au manuel du COBOL '74 d'Unisys [46].
- Juin 1992 – La documentation du compilateur FORTRAN 77 de HP (31501-90010) [51] présente aussi les facilités nécessaires à la compilation conditionnelle **$IF [$ELSE] $ENDIF** p.7-41 et **$SET** p.7-80

  Cette syntaxe de compilation conditionnelle sera reprise par quelques éditeurs de compilateur COBOL (UNISYS, AcuCobol, MicroFocus)
- Février 1999 – La norme ISO/IEC 1539-3:1999 [56] est dédiée à définir la compilation conditionnelle en FORTRAN. Cette norme ne rend pas la compilation conditionnelle obligatoire en Fortran, mais si un éditeur de compilateur FORTRAN implémente la compilation conditionnelle, alors il doit respecter cette norme.

  Les lignes destinées au compilateur pour lui gérer la compilation conditionnelle doivent

commencer par **??** en colonnes 1 et 2.
La norme sera retirée en mai 2011.

- Décembre 2001 – la norme ISO/IEC 1989:2002 [65] pour le langage COBOL précise ce qu'il en est de la compilation conditionnelle ; Grace à des directives identifiables au double chevron **>>**, il est possible de définir des variables et des instructions qui seront interprétées et évaluées par le compilateur afin savoir s'il doit, ou non, compiler les lignes de code entre deux directives.
  La compilation conditionnelle telle que définie dans cette norme ne sera implémentée dans le compilateur IBM qu'à partir de l'Enterprise COBOL for z/OS Release 6.2 [111].
- Janvier 2005 – Selon son manuel AA-Q2G0H-TK, la société HP [71] propose un compilateur COBOL disposant d'une compilation conditionnel basée sur un principe étendu des « debugging line » : le compilateur ne considère plus que la seule lettre '**D**' en colonne 7, mais les 26 lettres en majuscule et en minuscules.

4.3 Pourquoi la compilation conditionnelle est-elle apparue en COBOL ou FORTRAN ?

Faute d'avoir pu trouver des documents traitant du sujet ou d'avoir pu interroger des personnes (opérateurs, programmeurs,…) ayant vécus l'évolution, il s'agit d'hypothèses au regard de l'apparition de la compilation conditionnelle pour ces deux langages sur des plateformes mainframe.

Initialement la compilation conditionnelle - binaire, gérée par un seul flag - fut probablement la simple héritière évoluée des interventions manuelles effectuées par les opérateurs/opératrices sur des câblages ou interrupteurs de configurations, ou sur les paquets de cartes perforées.

On affecte le processus de compilation en spécifiant tel(s) ou tel(s) paramètre(s).

On produit tel exécutable en plaçant tel groupe de blocs de cartes,
ou tel autre exécutable en retirant tel autre groupe de blocs de cartes.

Même si, à des fins de débogage par exemple, des cartes de couleurs différentes (ou don le flanc avait simplement été marqué d'un trait de couleur) pouvaient avoir été utilisées pour distinguer les groupes de blocs de cartes à joindre au code, où à l'en retirer, cela n'excluait pas les risques de fausses manœuvres (interversion de blocs insérés) et finalement un travail à recommencer.
Si l'action manuelle était bien la manière de travailler, elle s'explique aussi par la possibilité de garder, à une époque où la mémoire était limitée et onéreuse :
- un compilateur non-complexifié par la détection des instructions conditionnelles ;
- un exécutable produit aussi limité que possible dès que possible.

Si on retrouve des explications sur le fonctionnement, il fut impossible de retrouver les discussions et argumentations « à priori » qui ont mené à l'instauration de cette fonctionnalité dans la définition du langage. Hypothèse donc : grâce à l'augmentation des capacités des machines et une adaptation des compilateurs, l'apparition d'une première compilation conditionnelle détectant le caractère '**D**' en colonne 1 pour le Fortran ou en colonne 7 pour le COBOL et agissant en conséquence, ou le remplacement, en COBOL seulement, des « debugging packets » par les déclaratives, a permis la fin du processus manuel fastidieux et risqué d'ajout et/ou de retrait de cartes perforées destinées au debugging, ainsi que l'économie du temps nécessaire à ces manipulations.

Que ce soit dans les manuels des compilateurs Fortran ou Cobol, ce sont dans des chapitres (ou paragraphes) dédiés au débogage que sont apparues les premières évocations d'instructions codées dans la source mais dont la compilation sera effective ou non en fonction d'un choix binaire porté à la connaissance du compilateur, soit au moyen d'un paramètre reçu à son lancement, soit au moyen d'un code convenu préalablement rencontré dans le texte source.

Dans son article « Program debugging using COBOL '74 » paru en 1975, à postériori donc, George N. BAIRD [18] présente bien la compilation conditionnelle comme la solution aux ajouts/retraits, qui se répèteront au cours de la vie de l'application, d'instructions de débogage ; il signale au passage que c'est aussi plus aisé et moins lourd que, pour un résultat presqu'équivalent, une programmation effectuée par le développeur qui maintiendrait une valeur qu'il gèrerait et testerait au run time pour l'aider à déboguer (p.ex. `DEBUG-MODE-ACTIF PIC X(3).`).

```
PERFORM SEND-MAIL
IF DEBUG-MODE-ACTIF = "YES"
   DISPLAY "I Send a Mail"
.
```

Depuis les offres techniques se multiplient également sur les systèmes mainframes (divers SGBD, télécommunications, parallélisme ou non,…). Ben que proportionnellement toujours très sommairement décrits dans les manuels, de nouveaux usages de la compilation conditionnelle sont donnés.

Suppose that the CICS® compiler option is in effect and the compiler works with the integrated CICS translator:

```
>>IF IGY-CICS
     EXEC CICS READ FILE-1... *> Read a record in CICS
>>ELSE
     READ FILE-2              *> Read a record in BATCH
>>END-IF
```

Fig. 7 Exemple de compilation conditionnelle donnée par IBM dans la documentation d'Enterprise Cobol for z/OS 6.2 [12]

Ce mémoire en proposera d'autres.

4.4 Quelles techniques/variétés de compilation conditionnelle ressortent de ces manuels COBOL ?

Le but de la présente section est d'expliquer les concepts existants et usages actuels, pas d'en détailler les syntaxes, d'en préciser les mots clés obligatoires ou optionnels pour lesquels la personne intéressée est invitée à se référer au manuel dédié au compilateur COBOL qu'il compte utiliser.

4.4.1.1 Caractère 'D' en colonne 7

Partie de la norme COBOL X.23-1974

Proposant deux scenarii, la plus simple des techniques de compilation conditionnelle à avoir été implémentée s'articule autour de la présence ou non, à l'initiative du programmeur, de la formule convenue **`WITH DEBUGGING MODE`** à l'endroit convenu dans le code programmé : en fin de la première ligne de la première section de la première division, à savoir **`SOURCE-COMPUTER`** dans la **`CONFIGURATION SECTION`** de l'**`ENVIRONMENT DIVISION`**.

Le principe est assez simple, pour tout type de ligne qui suit : si un '**`D`**' est rencontré en colonne 7 (colonne appelée INDICATOR AREA), la ligne (une telle ligne est appelée « debugging line ») sera considérée par le compilateur :
- comme du commentaire s'il n'a pas reçu la formule **`WITH DEBUGGING MODE`** (scénario 1)
- comme du code à compiler sinon (scénario 2).

```
           PERFORM SEND-MAIL
      D    DISPLAY "I Send a Mail"
```

[12] https://www.ibm.com/docs/en/cobol-zos/6.2?topic=compilation-examples-conditional

Si le texte du programme résultant, tant en mode debug qu'en mode non debug, est syntaxiquement correct au sens COBOL, c'est autorisé.

La présence de **WITH DEBUGGING MODE** déterminera comme « code à compiler » TOUTES les lignes du texte source ayant le caractère '**D**' requis en colonne 7 et les intègrera au texte du programme ; ce sera donc « tout ou rien » .

Il est important de constater que si un programme est morcelé en plusieurs fichiers de code source, *soit parce que suite à une factorisation il les partage avec d'autres programmes, soit simplement pour permettre à plusieurs personnes de travailler simultanément, chacune sur une partie du code du programme,* qu'une personne compilant le programme en mode débogage avec l'intention de suivre les fonctionnalités qu'elle-même a modifiées, elle sera impactée par toutes les debugging lines de tous les fichiers fusionnés .

Tel que décrit ci-dessus et conformément à ce qu'il y a dans les manuels, l'usage du **DEBUGGING MODE** se contente d'obtenir la compilation des DEBUGGING LINES, mais ce mode ne désactive aucune ligne qui serait à n'exécuter qu'en mode normal mais jamais en mode DEBUG (par exemple l'envoi d'un signal (mail, ou message dans une queue) vers une personne ou un partenaire extérieur à la société…) . Cela pourrait pourtant s'avérer nécessaire pour éviter que, à l'extérieur justement, on ne donne une suite réelle à un message généré par un test en mode débogage. Voici trois exemples montrant comment malgré tout obtenir cette alternative lors du mode débogage.

Le premier n'exclut pas mais, pour un résultat équivalent en mode debug, surcharge l'affectation

```
           MOVE "Production@gmail.com" TO DESTINATAIRE-MAIL
      D    MOVE "cobolistes@gmail.com" TO DESTINATAIRE-MAIL
           PERFORM SEND-MAIL
```

Le deuxième exemple se contente d'exclure une instruction si le mode debug est activé

```
      D    IF 1 = 2
              PERFORM SEND-MAIL
      D    .
```

Le troisième, en mode debug, exclu l'action d'envoi et la remplace par l'affichage d'un message

```
      D    IF 1 = 1
      D       DISPLAY "I Send a Mail"
      D    ELSE
              PERFORM SEND-MAIL
      D    .
```

4.4.1.2 Les déclaratives de format et usage USE FOR DEBUGGING

Partie de la norme COBOL X.23-1974

En COBOL, on appelle une **DECLARATIVE** un ensemble d'instructions exécutables regroupé dans une section localisée entre les mots **DECLARATIVES.** et **END DECLARATIVES.** Plusieurs déclaratives peuvent être définies entre ces deux mots clés. Ceci est toujours localisé tout au début de la **PROCEDURE DIVISION.**

Une déclarative n'est pas exécutée explicitement : pas de **PERFORM** ni de **GO TO** pour l'initier. Une directive est associée à un critère d'exécution que l'on retrouve dans sa définition, derrière le mot réservé **USE**. Il appartient au compilateur de déterminer tous les endroits où ce critère pourrait être satisfait, d'y intégrer lui-même la vérification du critère si besoin est, et d'y veiller à l'exécution de la directive si le critère est satisfait (« souscrire à un évènement »).

Un seul format de déclarative est soumis à la compilation conditionnelle, il s'agit de celui faisant usage de **USE FOR DEBUGGING ON** … Les déclaratives respectant ce format, les instructions vérifiant leurs critères et les exécutions de ces directives ne seront compilées que si l'option **WITH DEBUGGING MODE** est présente. ATTENTION DONC : puisque l'usage de ce format de déclarative requiert l'option **WITH DEBUGGING MODE**, cela va de facto impliquer la compilation de toutes les debugging lines.

```
PROCEDURE DIVISION.
DECLARATIVES.
DESTINATAIRE-MAIL SECTION.
    USE FOR DEBUGGING ON SEND-MAIL.
DESTINATAIRE-DE-TEST.
    MOVE "cobolistes@gmail.com" TO DESTINATAIRE-MAIL (1).
END DECLARATIVES.
```

L'avantage de la déclarative est que l'on ne doit pas passer en revue l'entièreté du programme à la recherche d'exécution(s) du paragraphe SEND-MAIL pour y ajouter une debugging line remplaçant le destinataire tel que dans les exemples précédents ; c'est donc moins de travail de programmation, et la source du programme se lit quasi dans sa version finale puisque ce qui concerne le debugging est rassemblé dans les déclaratives.

La version suivante comportant de nombreux envois diversifiés

```
PROCEDURE DIVISION.
     ...
     MOVE "Business@gmail.com"    TO DESTINATAIRE-MAIL (1)
D    MOVE "develop@gmail.com"     TO DESTINATAIRE-MAIL (1)
D    MOVE "TeamLead@gmail.com"    TO DESTINATAIRE-CC-MAIL (1)
     PERFORM SEND-MAIL
     ...
     MOVE "Production@gmail.com" TO DESTINATAIRE-MAIL (1)
D    MOVE "cobolistes@gmail.com" TO DESTINATAIRE-MAIL (1)
D    MOVE "TeamLead@gmail.com"   TO DESTINATAIRE-CC-MAIL (1)
     PERFORM SEND-MAIL
     ...
     MOVE "Audit@gmail.com"       TO DESTINATAIRE-MAIL (1)
D    MOVE "QC-Team@gmail.com"     TO DESTINATAIRE-MAIL (1)
D    MOVE "TeamLead@gmail.com"    TO DESTINATAIRE-CC-MAIL (1)
D    MOVE "Prd-Owner@gmail.com"   TO DESTINATAIRE-CC-MAIL (2)
     PERFORM SEND-MAIL
```

Peut être remplacée par

```
PROCEDURE DIVISION.
DECLARATIVES.
DESTINATAIRE-MAIL SECTION.
    USE FOR DEBUGGING ON SEND-MAIL.
DESTINATAIRE-DE-TEST.
    IF DESTINATAIRE-MAIL (1) = "Production@gmail.com"
       MOVE "cobolistes@gmail.com" TO DESTINATAIRE-MAIL (1)
    ELSE
    IF DESTINATAIRE-MAIL (1) = "Audit@gmail.com"
       MOVE "QC-Team@GMAIL.COM"    TO DESTINATAIRE-MAIL (1)
       MOVE "Prd-Owner@gmail.com" TO DESTINATAIRE-CC-MAIL (2)
    ELSE
    IF DESTINATAIRE-MAIL (1) = "Business@gmail.com"
       MOVE "develop@gmail.com"    TO DESTINATAIRE-MAIL (1)
    .
    MOVE "TeamLead@gmail.com"      TO DESTINATAIRE-CC-MAIL (1)
    .
END DECLARATIVES.

*Et dès après les déclaratives, le pgm a son aspect final
 ...
 MOVE "Business@gmail.com"    TO DESTINATAIRE-MAIL (1)
 PERFORM SEND-MAIL
 ...
 MOVE "Production@gmail.com" TO DESTINATAIRE-MAIL (1)
 PERFORM SEND-MAIL
 ...
 MOVE "Audit@gmail.com"       TO DESTINATAIRE-MAIL (1)
 PERFORM SEND-MAIL
```

4.4.1.3 Précision concernant les DEBUG PACKETS

Pour information et afin d'éviter toute confusion basée sur le nom du concept, ou sur le nom de la carte COBOL (**DEBUG** `location`) identifiant un tel paquet d'instructions compilables et exécutables : les **DEBUG PACKETS** ne dépendent absolument pas du **WITH DEBUGGING MODE**. Si des DEBUG packets sont présents après la source du programme, ils seront compilés et feront partie de l'exécutable. Sauf à considérer l'intervention manuelle d'ajout/retrait comme l'indicateur, ils n'ont donc rien à voir avec la compilation conditionnelle.

Au vu des explications les concernant, tout porte à comprendre qu'ils sont les prédécesseurs des déclaratives de format **USE FOR DEBUGGING ON** `location` tel que décrit au point précédent. En l'absence de l'indicateur **WITH DEBUGGING MODE**, inexistant, leur localisation après la fin du programme facilitait l'opération d'ajout ou de retrait des cartes les constituant.

4.4.1.4 La technique « retirée » de la norme

Jusqu'ici a été abordée une compilation conditionnelle binaire et élémentaire basée sur la présence ou non de **WITH DEBUGGING MODE** :

- indicateur activé ➔ compilation de la ligne, sans alternative (sauf recours à une astuce)
- indicateur non activé ➔ la ligne est considérée comme du commentaire et n'est pas compilée ;

le compilateur n'avait rien à interpréter mais juste à connaitre le statut d'indicateur(s) et à détecter une directive de format **USE FOR DEBUGGING**, ou la présence en colonne 7 d'une lettre associée à l'indicateur.

Bien qu'assumées par les éditeurs de compilateurs COBOL pour assurer une compatibilité arrière des codes y ayant recours, ces fonctionnalités ont été retirées de la norme COBOL comme l'avait annoncé J. Sammet & J Garfunkel [41] à la suite de leur étude de l'évolution du COBOL.
Faute de pouvoir retrouver toutes les évolutions de la norme COBOL, impossible de savoir à partir de laquelle, mais dans la norme ISO/IEC 1989 :2002(e) au plus tard, la technique a réintégré la norme.

L'actuelle norme COBOL, et au plus tard depuis la norme ISO/IEC 1989 :2002(e) dispose d'un ensemble plus efficace de possibilités qui vont être présentées ci-dessous *… après avoir présenté les compilations conditionnelles :* de HP, ensuite celle presqu'aussi puissante et déjà implémentée dans le compilateur COBOL fourni par Burroughs en 1974, puis une syntaxe qui se répandit avant la norme. S'en suivra une comparaison d'elles.

4.4.1.5 Caractère alphabétique non accentué 'a' .. 'z' 'A' .. 'Z' en colonne 7

Il s'agit ici d'une compilation conditionnelle fournie par HP [71], c'est une extension de la norme initiale et du caractère '**D**' en colonne 7. Les 26 lettres minuscules et les 26 lettres majuscules peuvent être utilisées en colonne 7 pour marquer une debugging line. L'usage de **WITH DEBUGGING MODE** précisera au compilateur que toutes les debugging lines, quelle que soit en colonne 7 la lettre et sa casse, doivent être compilées. A part la diversité des lettres, ceci donne le même résultat que la description précédente limitée à la lettre '**D**'.

La différence réside lorsqu'on n'utilise pas l'option **WITH DEBUGGING MODE**. Lorsqu'on ne l'utilise pas, les debugging lines ne sont normalement pas compilées, mais le seront malgré tout celles dont la lettre est dans la liste de lettres séparées par une virgule passée au compilateur via le paramètre **CONDITIONALS=(..)**. Ce paramètre demande donc au compilateur de compiler, bien qu'on ne soit pas en debugging mode, les debugging lines précédées d'une de ces lettres.

Exemple.

```
      MOVE "Production@gmail.com" TO DESTINATAIRE-MAIL (1)
M     MOVE "cobolistes@gmail.com" TO DESTINATAIRE-MAIL (1)
c     MOVE "testeurs@gmail.com"   TO DESTINATAIRE-CC-MAIL (1)
F     MOVE "TeamLead@gmail.com"   TO DESTINATAIRE-CC-MAIL (2)
p     MOVE "Prd-Owner@gmail.com"  TO DESTINATAIRE-CC-MAIL (3)
M     IF 1 = 1
M        DISPLAY "I Send a Mail to " DESTINATAIRE-MAIL (1)
c        " and to " DESTINATAIRE-CC-MAIL (1)
F        " and to " DESTINATAIRE-CC-MAIL (2)
p        " and to " DESTINATAIRE-CC-MAIL (3)
M     ELSE
         PERFORM SEND-MAIL
M     .
```

En passant le paramètre **CONDITIONALS=(p,M)** on compilera le texte de programme équivalent à

```
      MOVE "Production@gmail.com" TO DESTINATAIRE-MAIL (1)
      MOVE "cobolistes@gmail.com" TO DESTINATAIRE-MAIL (1)
*     MOVE "testeurs@gmail.com"   TO DESTINATAIRE-CC-MAIL (1)
*     MOVE "TeamLead@gmail.com"   TO DESTINATAIRE-CC-MAIL (2)
      MOVE "Prd-Owner@gmail.com"  TO DESTINATAIRE-CC-MAIL (3)
      IF 1 = 1
         DISPLAY "I Send a Mail to " DESTINATAIRE-MAIL (1)
*        " and to " DESTINATAIRE-CC-MAIL (1)
*        " and to " DESTINATAIRE-CC-MAIL (2)
         " and to " DESTINATAIRE-CC-MAIL (3)
      ELSE
```

```
           PERFORM SEND-MAIL
       .
```

Cette technique plus flexible que l'usage du seul '**D**' implique plus de risques :
- il y a d'autant plus nombreuses combinaisons possibles à analyser qu'il y a de lettres utilisées,
- l'activation de debugging lines ne dépend plus du seul programmeur pouvant éditer le texte source, mais de toute personne ayant la possibilité d'en lancer une compilation.

Dans l'exemple ci-dessus, pour éviter qu'un mail ne puisse partir vers un destinataire autre que celui de production, l'usage d'une de trois autres lettres lorsque le M n'est pas dans la liste **CONDITIONALS=(..)** aura pour conséquence de produire une faute de compilation : l'absence de l'instruction **DISPLAY** laissera orphelines les lignes commençant par " and to " ...

Cette extension proposée par HP est une manière d'obtenir plus de flexibilité que celle explicitée précédemment, et fournit une réponse au « tout ou rien » en attribuant, pour le débogage par exemple, une lettre par COPY, mais pas seulement : elle permet beaucoup de scénarii et entre autres de déjà effectuer de la compilation conditionnelle à d'autre fin que du simple débogage.

Exemple limité à ce que pourrait être la partie lecture d'un compte par un programme envisagé pour être utilisé dans un environnement disposant d'un système de base de données (lettre '**S**'), ou alors ayant recours à un fichier indexé (lettre '**I**').

```
S      MOVE WANTED-BANKACCOUNT-NR  TO  KEY—ACCOUNT-TBL
S      PERFORM GET-ACCOUNT--DBMS
I      MOVE WANTED-BANKACCOUNT-NR  TO  KEY-ACCOUNT-FILE
I      PERFORM GET-ACCOUNT--FILE-INDEXED
       IF NOT ACCOUNT-NOT-FOUND
I         MOVE CORRESPONDING ACCOUNT-REC TO STD-LAYOUT
S         MOVE CORRESPONDING ACCOUNT-ROW TO STD-LAYOUT
          PERFORM MANAGE-ACCOUNT-FROM-STD-LAYOUT
       .
```

Pour éviter du code mort lors d'une programmation telle que ci-dessus, toutes les instructions des paragraphes conditionnellement exécutés doivent être soumises à la même condition de compilation, ainsi celles de GET-ACCOUNT--DBMS doivent contenir un '**S**' dans leur colonne 7.

Trucs et astuces : Il est important de remarquer aussi que les options '**S**' et '**I**' ne sont ici pas mutuellement exclusives et qu'on pourrait très bien compiler le texte source en utilisant **CONDITIONALS=(I,S)** menant éventuellement à un résultat pouvant être très aléatoire lors de l'exécution ; par exemple si le compte est introuvable en DB mais est bien présent dans le fichier indexé, STD-LAYOUT verra ses champs affectés sur base d'un ACCOUNT-ROW au contenu indéterminé. Lorsque c'est requis, on peut rendre des flags mutuellement exclusifs au moyen d'une déclaration qui mènerait à une erreur de compilation:

```
I 01 CHOISISSEZ-I-XOR-S PIC X VALUE "I".
S 01 CHOISISSEZ-I-XOR-S PIC X VALUE "S".
```

Et on peut s'assurer qu'au moins une des deux options aura été choisie par une instruction dont la compilation ne dépendra, elle, d'aucun des deux caractères '**I**' ou '**S**'

```
       DISPLAY "l'option choisie est " CHOISISSEZ-I-XOR-S.
```

En dehors du XOR présenté ci-dessus, malgré l'absence d'opérateur logique on peut s'assurer de certains assemblages de conditions en utilisant cette définition de compilation conditionnelle.

Ainsi si on souhaite AU MOINS UNE condition parmi plusieurs :

```
       01 OBTENEZ-R-OR-S-OR-T.
```

```
R     02 FILLER PIC X VALUE "R".
S     02 FILLER PIC X VALUE "S".
T     02 FILLER PIC X VALUE "T".
```

OBTENEZ-R-OR-S-OR-T aura une longueur nulle en l'absence d'activation d'au moins une des trois conditions, et l'instruction ci-dessous provoquera alors une faute de compilation

```
DISPLAY "l'option choisie est " OBTENEZ-R-OR-S-OR-T.
```

Et on peut s'assurer que si une option est choisie, alors d'autres le sont aussi, en acceptant éventuellement qu'aucune ne le soit :
assurons nous de l'existence de l'une (X dans cet exemple) si une des autres existe

```
X 01 X-AND-Y-AND-Z PIC X VALUE "&".
Y 01 NOW-Y REDEFINES X-AND-Y-AND-Z PIC X.
Z 01 NOW-Z REDEFINES X-AND-Y-AND-Z PIC X.
```

Et comme précédemment c'est lors de la compilation qu'on fait valider la condition en vérifiant que si X existe, les deux autres aussi sans quoi une faute de compilation survient :

```
X     DISPLAY "On a bien un AND " X-AND-Y-AND-Z NOW-Y NOW-Z.
```

alors que pour vérifier l'activation des trois de manière inconditionnelle on codera

```
DISPLAY "On a toujours " X-AND-Y-AND-Z NOW-Y NOW-Z.
```

Grâce aux facilités de COBOL pour définir une structure hiérarchique, les trucs et astuces présentés ci-dessus sont combinables pour certains.

4.4.1.6 Unisys, une vraie compilation conditionnelle en COBOL depuis 1974 (Burroughs)

Burroughs, qui a fusionné avec Sperry pour former Unisys, a implémenté dans son compilateur COBOL [46] une technique mise au point pour ses compilateurs XALGOL et ALGOL.

La technique est basée sur ce que la société appelait les « dollar cards » à l'époque, appelées « compiler control records » (CCRs) actuellement. Dans le texte source, un CCR est une ligne commençant par le caractère '**$**' (en colonne 7 pour le COBOL). Le texte source est l'input du compilateur ; éparpillées dans ce texte source, on peut trouver zéro, une ou des lignes commençant par le caractère '**$**' ; ce ne sont pas des lignes que le compilateur doit compiler, ce sont des lignes d'instructions qui lui sont données au moyen de commandes (au nombre de 3) et dont il doit tenir compte pour effectuer son travail.

Les options permettant le contrôle du processus de compilation ont des valeurs par défaut. Si l'utilisateur le souhaite, pour des options qui l'intéressent, il peut au lancement du job de compilation spécifier, via un fichier spécifique ou comme paramètres en ligne de commande, des valeurs autres qui seront prédominantes sur celles par défaut. Enfin, à l'intérieur même du code source, les CCRs servent à fixer ou faire évoluer des options de compilation lorsqu'il a été décidé d'en appliquer des spécifiques, locales à la source compilée, par rapport à celles définies par défaut ou au lancement de processus de compilation ou ailleurs dans le même texte source. Même si elles peuvent varier en type (booléen, énuméré, valeur, …) ou en nombre au cours des versions du compilateur, la liste de ces directives prédéfinies est exhaustive et connue du compilateur. Seule une d'entre elle **OMIT** fut prévue pour déterminer ce qui devait être compilé, ou pas ; trois autres **LIST**, **LISTDOLLAR** et **LISTOMITTED** permettent de décider ce qui en apparaitra sur le listing de compilation.

La directive **OMIT**, lorsqu'elle est à TRUE, informe le compilateur d'ignorer de son processus de compilation toutes les instructions du langage compilé qui suivent jusqu'à un changement de valeur de OMIT. A contrario, lorsque la directive **OMIT** est à FALSE, il doit compiler les instructions du langage compilé qui suivent jusqu'à un changement de valeur de **OMIT**. Quelle que soit la valeur de la directive **OMIT**, les CCRs rencontrées seront toujours interprétées par le compilateur. Cette version de la

compilation conditionnelle consiste donc à manipuler la valeur effective de la directive **OMIT** pour choisir le scénario local à compiler.

La directive **LISTDOLLAR** n'a qu'un effet visuel : lorsqu'elle est à TRUE, les CCRs (précédemment appelées DOLLAR CARDs d'où le nom de la directive) apparaitront aussi sur le listing de compilation, et elles n'apparaitront, sauf exception ci-après, pas si l'option est à FALSE. Quelle que soit la valeur de **LISTDOLLAR**, un CCR qui commence par '**$$**' en colonnes 7 et 8 apparaitra sur le listing de compilation pour autant qu'il soit généré. La directive **LISTDOLLAR** peut aussi s'écrire **LIST$.**

La directive **LISTOMITTED** n'a aussi qu'un effet visuel : lorsqu'elle est à TRUE, les instructions ignorées du compilateur de par la valeur de **OMIT** apparaitront bien sur le listing de compilation mais suffixées du mot **OMIT** en fin de ligne pour que le lecteur sache qu'elles sont omises. Lorsque l'option est à FALSE, les lignes non compilées n'apparaitront pas sur le listing. La directive **LISTOMITTED** peut aussi s'abréger en **LISTO**.

Si la directive **LIST** (production d'un listing de compilation) est à FALSE, les valeurs respectives de **LISTDOLLAR** et de **LISTOMITTED** sont ignorées puisqu'aucun listing n'est produit.

Lorsqu'elles sont à TRUE, les directives **LIST** et **LISTDOLLAR** permettent de suivre le résultat de l'évaluation de chaque CCR : chacune d'elle sera aussi suffixée de **OMIT** si son évaluation est TRUE … et donc il est possible de procéder au débogage des CCRs interprétées par le compilateur dans le cadre de la compilation conditionnelle.

Lorsque **LISTDOLLAR** et **LISTOMITTED** sont à FALSE, le listing de compilation généré quand **LIST** est à TRUE sera débarrassé de ce qui n'est pas compilé (CCRs et instruction omitted) pour faciliter le suivi, la compréhension et la certitude puisque seul le scénario compilé apparait sur le listing.

En plus de ces quatre directives décrites ci-avant, l'utilisateur a la possibilité de déclarer et manipuler ses propres options, de type booléen seulement. Les options définies par l'utilisateur doivent chacune avoir un nom unique de maximum 31 caractères. Lorsqu'un nom d'option inconnu est passé comme paramètre au lancement du compilateur, ce nom sera assumé être une option utilisateur, sa déclaration avec la valeur TRUE par défaut sera implicite ; son nom de développeur ou l'identifiant de sa DB de test peuvent être de tels paramètres qui définiront des variables de compilations. Une option utilisateur ne doit pas obligatoirement être passée en paramètre pour être déclarée et initialisée, les commandes ci-dessous qui les manipulent procèderont aux déclarations explicites.

Pour chacune des quatre options prédéfinies explicitées, ainsi que les noms d'option définis par l'utilisateur, trois commandes permettent de déterminer la valeur à prendre en compte :

- **$**[**$**][ ]*__SET__[ ]+((option)+|(option = expression-booléenne))
- **$**[**$**] [ ]*__RESET__[ ]+(option)+
- **$**[**$**] [ ]*__POP__[ ]+(option)+

Le premier '$' doit être en colonne 7, le deuxième '$' est facultatif mais doit être en colonne 8 s'il est présent ; avant la commande la présence d'espace est facultative, mais il en faut au moins un après la commande .

Dans sa première forme la commande **SET** (resp. **RESET**) ***ne modifie pas*** la valeur des options, elle ***empile une nouvelle valeur*** TRUE (resp. FALSE) sur la pile de chacune des options présentes sur le CCR, cette dernière valeur empilée occulte les précédentes et est considérée comme la valeur de l'option. Pour chaque option, la pile peut contenir jusqu'aux 47 dernières valeurs empilées ; la commande POP dépile la valeur présente au sommet de la pile, s'il n'y a pas de valeur précédente à rendre active,

la commande **POP** place un FALSE comme valeur de l'option.
Dans sa deuxième forme la commande **SET** évalue l'expression booléenne et empile le résultat TRUE ou FALSE sur la pile de l'option. L'expression booléenne doit tenir sur la ligne du CCR, elle peut contenir des **AND**, **OR**, **NOT**, parenthèses, sous-expressions, comparaisons ; si dans l'expression booléenne le nom d'un opérande est non connu, l'opérande sera considérée comme une opérande définie par l'utilisateur et sa valeur par défaut est FALSE ; si l'opérande est connue, la valeur du haut de la pile sera utilisée. C'est cette deuxième forme du **SET**, ainsi que **POP** qui seront utilisées pour faire évoluer l'option **OMIT**.

En résumé : l'option de compilation de type booléen **OMIT** utilisée en accordance avec les options de compilation définies par l'utilisateur permettent d'effectuer de la compilation conditionnelle beaucoup plus étendue que le simple **WITH DEBUGGING MODE** ou les 2x26 caractères de l'alphabet. En effet, sur base de critères éventuellement différents pour chacun d'entre eux, les CCRs **OMIT** effectueront chacun un choix binaire concernant les instructions sur lesquelles s'étend leur portée respective.

L'exemple suivant montre l'usage des 3 commandes dont **$SET** dans les deux formes, la définition d'options par l'utilisateur (BANKACCFILE[4(I|O])), l'usage d'une expression booléenne, ainsi qu'en dernière colonne le coté visible d'un OMIT qui est à TRUE.
Ce sont trois extraits d'un programme envisagé pour manipuler un fichier de comptes bancaires (BANKACCFILE) en output (BANKACCFILE4O) ou en input (BANKACCFILE4I). Le choix du programmeur (1er extrait) ne s'est pas porté vers la production d'un fichier output puisque **$RESET** BANKACCFILE4O le positionne à FALSE, l'instruction évalue donc **OMIT** à TRUE et cela est visible au suffixe de la CCR, ainsi qu'à l'instruction COBOL qui en dépend. Le programmeur avait par contre décidé que le fichier devait servir d'input grâce au **$SET** positionnant BANKACCFILE4I à TRUE d'où **OMIT** est FALSE, la CCR et les instructions d'ouverture et lecture initiale du fichier ne sont pas de suffixées de **OMIT**

```
JBA_LIST_P8B_SAMPLE_UNISYS_COBOL85_CONDITIONAL_COMPILE_DB_et_OMIT.txt - Bloc-notes
ıier  Edition  Format  Affichage  Aide
                 025000  $RESET BANKACCFILE4O                                                    XX221224    004:0000:1
                 026000  $SET BANKACCFILE4I                                                      XX221224    004:0000:1
JBA_LIST_P8B_SAMPLE_UNISYS_COBOL85_CONDITIONAL_COMPILE_DB_et_OMIT.txt - Bloc-notes
ıier  Edition  Format  Affichage  Aide
               2:011100  $SET BANKACCFILE      =       BANKACCFILE4I OR       BANKACCFILE4O XX221218    004:0000:1
JBA_LIST_P8B_SAMPLE_UNISYS_COBOL85_CONDITIONAL_COMPILE_DB_et_OMIT.txt - Bloc-notes
ıier  Edition  Format  Affichage  Aide
                 108502                                                                          XX221224    004:01D3:2
                 108550  $SET OMIT = NOT BANKACCFILE4O                                           XX221224       OMIT
                 108600         PERFORM CREE-MON-FICHIER-TEST                                    XX221224       OMIT
                 108650  $POP OMIT                                                               XX221224    004:01D3:2
                 108660                                                                          XX221224    004:01D3:2
                 108680  $SET OMIT = NOT BANKACCFILE4I                                           XX221224    004:01D3:2
                 108700         PERFORM OPEN-I--ACCOUNT-FILE                                     XX221219    004:01D3:2
                 108800         PERFORM READ--ACCOUNT-FILE                                       XX221219    004:01D5:1
                 108810  $POP OMIT                                                               XX221224    004:01D7:0
                 108820                                                                          XX221224    004:01D7:0
```

Fig. 8 Extrait de listing de compilation montrant le résultat de l'exploitation de directive de compilation conditionnelle

4.4.1.7 En attendant une norme, un style intermédiaire de compilation conditionnelle

Quelques éditeurs (Unisys, GnuCOBOL, ACUCOBOL, COBOL-IT, MicroFocus qui gère plusieurs « dialectes ») de compilateur COBOL ont utilisé des syntaxes assez semblables entre elles au moyen des commandes **$SET**, **$IF**, la possibilité d'une alternative **$ELSE** ou même **$ELSE IF** ou **$ELIF** et terminée par **$END**. Pour toutes ces versions de COBOL, le compilateur compilera les instructions du bloc dont la condition est TRUE, ou du bloc **$ELSE** si aucune condition n'est TRUE et pour autant que ce bloc **$ELSE** existe.

A côté de sa version propre décrite au paragraphe précédent, Unisys propose également les CCR dans cette syntaxe depuis la version de son COBOL 85 de 2003 au plus tard ; cet éditeur est le seul à permettre une expression booléenne éventuellement composée d'opérateur(s) logique(s) ; l'usage d'une pile par option pour gérer l'évolution de sa valeur est aussi spécifique à Unisys et d'application dans cette syntaxe aussi. Rappel, le compilateur UNISYS considèrera toujours tous les CCRs rencontrés.

Lorsqu'elle existe, la commande **$SET** des autres éditeurs (ACUCOBOL, COBOL-IT, GnuCOBOL) ne permet que de définir des constantes (numérique ou alphabétique)
$SET CONSTANT nom-cst-de-comp ((valeur numérique)|littéral)
en l'absence de **$SET** le Visual Cobol 6.0 de MICROFOCUS utilise les déclarations de constantes au niveau 78. Quel que soit le mode de déclaration, les constantes définies peuvent être de type chaine de caractères ou numérique.

Les conditions évaluables sont plus limitées :

- La comparaison d'une constante avec une valeur,
  ou pour certains : la comparaison de deux constantes entre elles
- Toutes les relations sont possibles grâce à [**NOT**](**<**|**=**|**>**),
  sauf COBOL-IT qui ne permet pas **<** et **>**
- L'existence de la définition d'un nom de constante **IS** [**NOT**] (**DEFINED**|**SET**),
  sauf COBOL-IT
- Des éditeurs tels que AcuCOBOL, COBOL-IT prévoient aussi l'usage de 10 variables booléennes X0 -X9 dont les valeurs ON/OFF peuvent varier grâce à l'usage de directives **$SET** et qui sont évaluables par les directives **$IF**
- Il n'y a pas d'opérateur logique binaire utilisable.
  Heureusement, contrairement à Unisys, les CCRs ne semblent pas prises en considération si elles sont dans un bloc non compilé ; le compilateur y veille malgré tout à l'appariement des **$IF** avec les **$END** ce qui permet de recourir aux **$IF** imbriqués ou successifs pour simuler ces opérateurs logiques.

Pour distinguer les lignes non compilées sur le listing de compilation, Unisys utilise les mêmes directives et la même présentation que pour la directive **$OMIT** : la présence de **OMIT** en fin de ligne pour tout **$IF** évalué à FALSE ainsi que pour toute ligne COBOL non compilée.
Les autres éditeurs ne l'évoquent pas… PRPLIST ?

4.4.1.8 Introduction de la compilation conditionnelle dans la norme COBOL ISO/IEC 1989:2002

La norme ISO/IEC 1989:2002 [65] se veut, entre autres, introduire une syntaxe de directives destinées à la compilation conditionnelle. Tous les éditeurs qui en implémenteront cette partie disposeront de moyens aisément compréhensibles, relativement complets et efficaces pour permettre aux programmeurs de préparer plusieurs scénarii grâce à une compilation conditionnelle digne de ce nom.

Cette norme est implémentée, selon les informations de ces éditeurs, dans les compilateurs MicroFocus Visual COBOL depuis la version 5.0, dans le compilateur IBM COBOL pour Z/OS depuis la version 6.2, ainsi que partiellement dans GnuCOBOL depuis la version 2.1 même si c'est pour un résultat identique.
Une directive commence par **>>** et peut être précédée d'espace(s)

Les directives sont au nombre de trois :

- **>>DEFINE** sert à la manipulation de variable(s) de compilation.

- **>>EVALUATE** est utilisée pour choisir un bloc d'instructions COBOL à compiler pour autant qu'une condition soit remplie, ou qu'il existe un bloc par défaut si aucune condition n'est remplie
- **>>IF** est utilisée pour compiler un bloc d'instructions COBOL pour autant que la condition soit remplie, ou un bloc par défaut s'il existe et que la condition n'est pas remplie

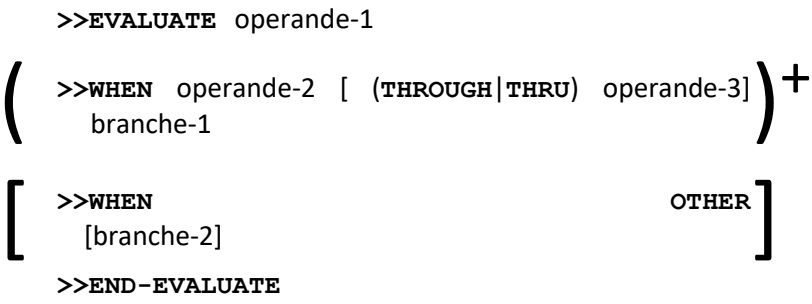
>>DEFINE : cette directive doit s'écrire sur une seule ligne et sert à :

- Définir le nom (unique) d'une variable de compilation ; sa valeur pouvant être exprimée sous forme d'expression arithmétique, sous forme de littéral, ou est obtenue comme paramètre de l'environnement selon une méthode à définir par l'éditeur du compilateur. Le nom d'une variable de compilation doit être distinct de tout nom de directive ;
  >>DEFINE nom-var-comp [**AS**] (arithm-express|littéral|**PARAMETER**)
- Annuler la définition d'une variable de compilation. Le nom de la variable de compilation sera alors considéré comme inexistant. Il sera donc possible de le définir à nouveau ;
  >>DEFINE nom-var-comp [**AS**] **OFF**
- Modifier la valeur de la variable de compilation
  >>DEFINE nom-var-comp [**AS**] (arithm-express|littéral|**PARAMETER**) **OVERRIDE**

Tout ceci est interprété par le compilateur durant la phase de compilation, il y a donc des limitations plus strictes que dans le code COBOL, par exemple l'expression arithmétique ne contient pas de COMPUTE, ni d'exposant, doit proscrire les divisions par zéro, travaille avec des nombres en point fixe, …

L'existence d'une variable de compilation peut être testée au moyen de **IS** [**NOT**] **DEFINED** pour servir de condition dans une directive **>>EVALUATE** ou **>>IF.**

>>EVALUATE : cette directive se compose de plusieurs mots clés et s'étend sur plusieurs lignes pour proposer plusieurs branches de code compilable, dont maximum une sera compilée. La directive est composée de 4 modèles de phrases repris ci-dessous.



- Chacune des phrases de la directive commence sur une nouvelle ligne et est entièrement formulée sur cette ligne ;
- Operande-1, operande-2 et operande-3 si elles existent, doivent être du même type ;
- **THROUGH** ou **THRU**, qui sont équivalents, ne sont autorisés que si les opérande-i résultent en valeurs numériques ;
- Une phrase **>>WHEN** operande-2 sera évaluée à TRUE si l'évaluation de l'operande-1 est égal à l'évaluation de l'operande-2 ;
- Une phrase **>>WHEN** operande-2 (**THROUGH**|**THRU**) operande-3 sera évaluée à TRUE si l'évaluation de l'operande-1 appartient à l'intervalle numérique fermé borné par l'évaluation de l'operande-2 et l'évaluation de l'opérande-3.

- Les phrases **>>WHEN** sont évaluées dans l'ordre ; dès que l'une est TRUE, les instructions COBOL de sa branche font partie du texte du programme et sont compilées,
  et toutes les autres phrases **>>WHEN** avec leur branche respective ainsi que, si elle existe, la phrase **>>WHEN OTHER** avec sa branche, sont ignorées.
  Si aucune phrase **>>WHEN** operande-… n'a été évaluée à TRUE et qu'il existe une phrase **>>WHEN OTHER**, ce sera alors la branche-2 associée à cette phrase qui sera incluse au texte du programme et compilée.
- Les directives **>>EVALUATE** peuvent être imbriquées

>>IF : cette directive se compose de plusieurs mots clés et s'étend sur plusieurs lignes pour proposer 1 ou 2 branches de code compilable, dont maximum une sera compilée. La directive est composée de 3 modèles de phrases repris ci-dessous.

```
>>IF                                   operande-1
   [branche-1]
[  >>ELSE                                        ]
   [branche-2]
>>END-IF
```

maintient une continuité avec l'usage intermédiaire développé en attendant la présente norme. La formulation est également plus simple que son équivalent

```
>>EVALUATE operande-1
>>WHEN                                  TRUE
   [branche-1]
[  >>WHEN                               OTHER    ]
   [branche-2]
>>END-EVALUATE
```

Operande-1 doit être une expression conditionnelle au résultat booléen, puisque comparée à TRUE.

4.5 Gros point d'attention !

Quelle que soit l'implémentation de la compilation conditionnelle utilisée, il faut rester extrêmement vigilant en procédant à des activations/désactivations de code qui peuvent amener des erreurs. Dans le meilleur des cas, elles seront des fautes de compilation et facilement perçues, alors que dans la pire des situations, elles changeront imperceptiblement la logique d'un programme (disparition d'un else, ou d'un point terminant un IF). Les risques (et la complexité consommatrice de ressources) sont rappelés dans la littérature évoquant la variabilité dans les programmes [87][102] (indépendamment de COBOL).
Exemple de situation « horrible » : le code est bon en mode « débogage », mais la désactivation de celui-ci fait exister l'erreur dans la fonctionnalité métier.

Il est à noter que pour un même texte source, s'il contient n variables distinctes évaluées par une directive #IFDEF, 2^n est le nombre envisageable de textes de programme [87][92]. Or selon la norme COBOL, outre l'existence ou non des variables de compilation **>>IF *maVariableDeCoCo* DEFINED**, il est prévu que chacune d'elles puisse contenir une valeur (un nombre entier, une chaine de caractère, un booléen) susceptible de changer durant le processus de compilation, faisant croître encore plus le nombre de textes de programme possibles. Imaginons que seules 5 variables de compilation, chacune

de type booléen, sont utilisées dans un texte source, le nombre de textes de programme possibles devient, si la valeur d'une variable reste inchangée au cours d'un même processus de compilation, minimum 3^5 puisque pour chaque variable de compilation on a potentiellement 3 états : <non définie>, <définie valant 'vrai'> et <définie valant 'faux'> ; en se rappelant que l'état de chacune des variables peut changer, s'il y a 77 évaluations >>IF … indépendantes, le nombre de textes de programme potentiels devient 3^{77}.

Chapitre 5

La recherche

5.1 Question de recherche

Pour le monde financier (bancaire et assurance) ou des administrations qui, confiant dans la stabilité et la compatibilité future qu'annonçaient les mainframes, ont, dès le milieu des années 1960 début des années 1970, commencé pleinement leur informatisation, le nombre de langages de programmation à disposition était limité. Le langage FORTRAN (1957) était très apprécié pour ses capacités de calcul ; il y avait la puissance de l'ALGOL (1958), … mais de tous, l'usage du langage COBOL (1959), dédié à la gestion, devint prédominant puisque toutes ces institutions financières devaient surtout faire du traitement de données, de la gestion « business ».
Contrairement à l'assembleur, au Fortran, PL/1 ou Algol qui nécessitaient un apprentissage certain de leurs syntaxes et de concepts, la programmation en COBOL s'appréhendait plus aisément. Usitant d'une syntaxe verbeuse clairement définie, COBOL exprimait en anglais et presqu'en langage clair et compréhensible par « tout le monde » ce qu'il y avait à faire. Grace à des programmes « squelettes » un expert business motivé et ayant suivi quelques formations pouvait devenir programmeur COBOL, de base peut-être, mais fonctionnel quand même. Suivant son attrait, sa motivation, l'encadrement d'informaticien(s) formé(s) lui permettant d'acquérir des techniques, sa connaissance fonctionnelle pouvait faire de lui « un informaticien » jusqu'à sa fin de carrière, maintenant et propageant dans le temps le langage de programmation qu'il connaissait.
Quelles qu'aient été les compositions d'équipes dans cette multitude de sociétés et d'administrations … elles auront produit énormément d'applications composées, d'autant plus de programmes, totalisant un nombre incroyable de lignes de code.

Selon l'enquête de MicroFocus commandée à Vanson Bourne dont les résultats ont été publiés en 2022 [13] il y aurait 800 milliards de lignes de code en COBOL. Malgré plusieurs demandes, les réponses de MicroFocus n'ont jamais expliqué le protocole d'enquête, ni chiffré en nombres absolus ce qui a servi à calculer les pourcentages publiés ou étayé ce nombre de lignes de code. Monsieur Ed Airey, de

[13] https://www.microfocus.com/en-us/press-room/press-releases/2022/cobol-market-shown-to-be-three-times-larger-than-previously-estimated-in-new-independent-survey

MicroFocus, coprésentateur avec Monsieur Jimmy Mortimer de Vanson Bourne, du webinar « how much cobol is really out there ? » a répondu le 23/3/2023 « *At this time, we're not providing details beyond that of what I shared previously in our summary reports and results webinar*. » Aucune réponse n'a été fournie par Vanson Bourne.

Depuis, le monde de l'informatique a évolué de manière incroyable, surtout en termes d'accessibilité : fini le temps où les ordinateurs n'étaient accessibles qu'aux institutions (entreprises, administrations, structures d'enseignement, …) ; l'apprentissage d'un langage de programmation ne se limite plus aux centres de recherche, au domaine militaire, au milieu universitaire, ni aux hautes écoles formant les programmeurs devant rejoindre les institutions financières, administratives, industrielles, … toutes soucieuses de ne pas trop vite perdre les lourds investissement réalisés et confiantes dans la stabilité d'une combinaison machine+langages proposée par un fournisseur. L'informatique s'est démocratisée :

- Les ordinateurs personnels se sont développés,
- À côté des programmes propriétaires payants on trouve autre chose que des copies pirates : il a des programmes gratuits, partagés ; … parfois soutenus par des communautés et de très haute qualité.
- À côté des environnements de développement intégré – les IDE – (heureusement de moins en moins sobres) il y a parfois des générateurs de code,
- À côtés des programmes de gestion, de bureautique, de CAO, … on trouve des programmes ludiques,
- À côtés des langages de programmations historiques dédicacés au monde professionnel, … on trouve des langages de programmation facilitant le développement informatique pour de nombreux domaines ;
- La documentation ne se recherche plus dans des livres, ne s'échange plus par disquettes, … mais se glane sur internet,
- Les questions-réponses s'échangent sur les forums,
- Pour beaucoup, la programmation ne se fait plus de bout en bout mais elle assemble des morceaux disponibles sur internet.
- Tout un chacun disposant d'un ordinateur et y passant du temps peut développer et rendre disponible une application ; son utilité, son unicité, son prix, sa stabilité, sa fiabilité, sa publicité : tous des facteurs qui interviendront pour déterminer sa popularité

Certes cela est plein d'avantages, mais cela entraîne des conséquences.

Quand on est habitué à « tant de facilités », quand on programme pour une visibilité immédiate, quand on dispose des ressources de la machine (surpuissante) pour soi tout seul, quand on … , on peut avoir une appréhension face à ce que tout le monde considère comme « préhistorique », « ringard », qui « ne traite que des caractères sur des écrans de 25 lignes et 80 colonnes », avec, dans le meilleur des cas, juste un peu de couleurs. Cela peut paraitre rébarbatif à nombre de jeunes programmeurs. Les offres d'emploi pour le fortran, le PL/1, le RPG, l'Algol sont occasionnelles, celles pour le COBOL sont plus nombreuses et il y a chaque fois plusieurs candidats (facilement une dizaine, parfois passé 100) mais au vu des programmeurs croisés chez un client, recroisé chez un autre, … ce sont souvent les freelances qui tournent, d'un poste à l'autre, avec de meilleures conditions à chaque fois et exceptionnels sont ceux de moins de 50 ans. Dans l'équipe de 14 personnes développant en COBOL à l'ONEm, en janvier 2023, l'âge moyen était de 58,35 ans grâce à l'engagement de deux « jeunes » cobolistes ayant respectivement 54 et 57 ans… enfin, un tout jeune de 33 ans a commencé cette année. Et dans la foulée : on vient aussi de solliciter, et il a accepté, un ancien collègue pensionné à ses 65 ans … en 2018. Effectivement, avant que les vieilles applications n'aient été automatiquement converties,

réécrites, remplacées ou abandonnées, bref tant qu'elles seront utilisées, elles doivent être maintenues pour suivre, à minima lorsqu'elles sont concernées, les évolutions de la législation.

… il va donc falloir attirer du monde vers la programmation COBOL.

Simplifier la tâche des programmeurs, ou les rapprocher de facilités qu'ils connaissent et attendent comme « minimums » peut donc s'avérer nécessaire pour que le COBOL leur semble moins rébarbatif.

L'usage de la compilation conditionnelle en COBOL permet-elle, à cette fin,
de proposer des codes au « design » plus contemporains ?

5.2 Méthodologie

La méthodologie a débuté en déterminant un objectif initial : que peut-on faire de plus en COBOL avec la compilation conditionnelle pour capter la relève des jeunes programmeurs ?

Plusieurs requêtes basées sur, par exemples
cobol AND « conditional compilation » AND (« oo emulation » OR « object oriented emulation »)
cobol AND « conditional compilation » AND (« structured copy » OR « structured copies »)
…
et d'autres ne donnaient aucun résultat.
C'est donc sur la contrainte minimum (ne reprenant que les noms de la technique et du langage) que fut basé l'état de l'art `"conditional compilation" AND cobol`. Cette simple combinaison ne fournit pas non plus beaucoup de résultats comme c'est indiqué dans l'état de l'art, et un seul (N. Volanschi (*op.cit.*)) associant les deux concepts pour dire que la compilation conditionnelle n'existe pas en COBOL. La norme définissant la compilation conditionnelle en COBOL existait bien mais n'était pas encore implémentée dans les compilateurs, et les compilateurs implémentant une compilation conditionnelle la fournissaient à titre « d'extension propre » à la norme.

Avant de rencontrer, même excessivement peu utilisée, la compilation conditionnelle sur le matériel Unisys il y a une huitaine d'année, son existence m'était inconnue et l'absence d'article aurait semblé couler de source,… mais on était maintenant après cette rencontre. Il a alors semblé important de prendre connaissance de l'histoire de la compilation conditionnelle sur mainframe, de sa présence dans certains compilateurs et pas d'autres. Chacun pouvant contenir l'une ou l'autre richesse et/ou facilité, il fut décidé de joindre une description et une comparaison de celles retenues.

L'association des deux concepts n'existant pas, cela laissait beaucoup de champ libre, encore fallait-il savoir ce que recouvrait la notion de « conditional compilation ». Quelques articles en donnaient une définition ou en citaient une. De proche en proche les recherches ont alors été étendues et ont été retenus les articles où une application proposée semblait applicable à COBOL. Le concept des Software Product Line (SPL), basées sur la compilation conditionnelle lorsqu'elles ne sont pas gérées au runtime, est apparu dans les articles et avait aussi été évoqué par le professeur Cleve (promoteur du présent mémoire), il fut donc aussi une orientation prise pour les lectures. Cela permet de mettre des noms de concepts bien définis sur des usages de la compilation conditionnelle.

Les développements de codes, que rien n'empêchait de commencer, furent entamés en parallèle. Certains furent initiés par les lectures, certains furent rattachés à des lectures, et l'émulation de l'OO semble innovant.

Au cours des 8 dernières années nous avons incorporé de la compilation conditionnelle dans quelques codes sur le lieu de travail. Comme expliqué dans les résultats, ces ajouts sont à l'origine des seules réactions qui ont pu être collectées.

5.3 Les compilations conditionnelles de dialectes COBOL, et celle de la norme

5.3.1 Comparaisons et critiques

Les diverses compilations conditionnelles relevées dans une section de l'état de l'art ont leurs avantages et inconvénients qui vont être ici comparés. Les différences relevées se regroupent sur ces trois caractéristiques

- Le moyen : les codes **`D|a..zA..Z`** vs les constantes vs « l'empilement » vs les variables
- La condition : son activation, sa formulation et ses opérandes ;
- La facilité d'usage : être certain de sa programmation conditionnelle, la visibilité sur le listing de compilation vs la coloration syntaxique dans l'éditeur

5.3.1.1 La version de base

L'usage du '**`D`**' est relativement lisible, dispose d'une facilité d'usage sans égale, simple et efficace pour du debugging (sa finalité), mais hélas très limité en possibilité. Il présente les gros inconvénients de ne produire que deux alternatives : tout ou rien. C'est le programmeur qui manipule le code contenant la phrase **`SOURCE-COMPUTER`** qui à la main pour activer ou désactiver la compilation conditionnelle, et lui seul. À moins que, comme c'est maintenant fort répandu, chacun puisse pour son environnement local de travail prendre une copie (clone) tant du programme que d'éventuel(s) COPY(s) qu'il ne souhaite pas voir modifié(s) pour la stabilité de ses propres tests. S'il s'agit là d'une facilité proposée par les gestionnaires de sources, cela peut s'avérer problématique lorsque plusieurs personnes doivent travailler sur des COPY's servant à constituer le texte source et chacun effectuer leurs tests.

A noter comme risque supplémentaire que si dans la source l'option **`WITH DEBUGGING MODE`** est présente au moment d'envoyer (add – commit – push) le programme en production, ce sera la totalité des instructions de débogage qui sera compilée et potentiellement exécutée. Des éditeurs ont incorporé un Run Time Switch permettant de passer outre les instructions de débogage même lorsque celles-ci sont présentes dans l'exécutable. Tout ceci n'est pas sans conséquences lorsqu'un programme traite des dizaines de milliers, voire des millions de données.

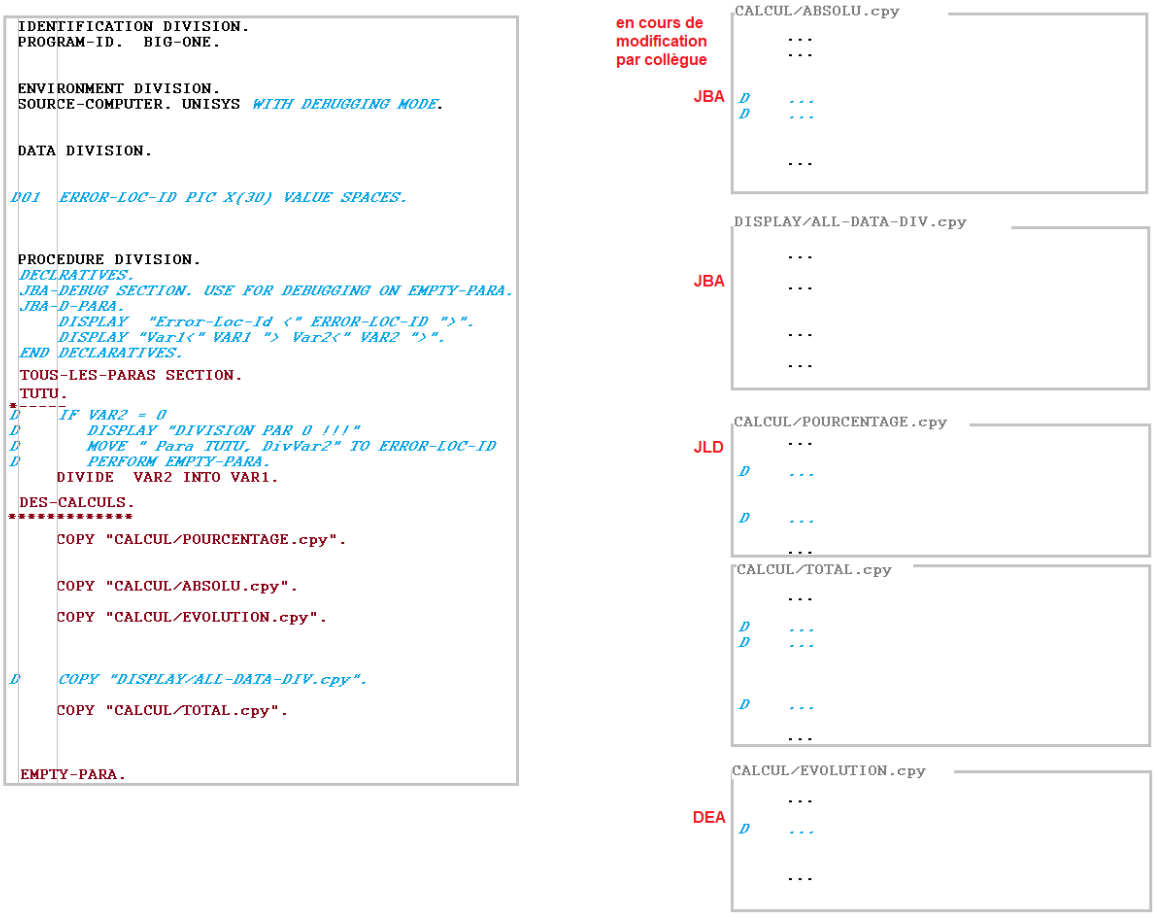


Fig. 9 L'usage de l'option « WITH DEBUGGING MODE »

L'influence de l'option **`WITH DEBUGGING MODE`** porte sur tout ce qui est mis en évidence en bleu ; remarquons que les lignes commençant par le caractère '**D**' peuvent être présentes au niveau des déclarations, d'instructions et même pour de la prise en compte d'un COPY.
(Le caractère '**D**' est bien en colonne 7, mais comme on ne travaille plus avec des cartes perforées, les 6 premières colonnes qui étaient destinées à contenir les numéros de cartes ne sont plus montrées : généralement les éditeurs actuels affichent à partir de la colonne 7 (appelée Indicator Area))

Tous les autres styles de compilation conditionnelle peuvent produire un résultat identique.

5.3.1.2 Les constantes

Basée sur la définition de « constantes » qui peuvent être testées (**`$IF`** … **`[$ELSE…]`** … **`$END`**), une version implémentée par quelques éditeurs (le cas Burroughs/Unisys est discuté plus loin) permet de remédier au problème du « tout ou rien » de la précédente technique et offre une compilation conditionnelle assez complète qui s'affranchi pleinement de l'objectif initial du seul débogage. Ainsi on dispose de :

- La possibilité de définir :
  - Des paramètre(s) **`CONSTANT nom-constate valeur`** sur la ligne de commande de compilation (typiquement pour l'identifiant du soumissionnaire de la compilation, ou de l'environnement…) ;
  - De nombreuses constantes (d'aspect technique ou de débogage) de compilation dans le code COBOL **`$SET CONSTANT nom-constate valeur`**
  - ou déclarées de niveau 78

que le compilateur peut tester/évaluer ;

- La possibilité de programmer, à destination du processus de compilation, des choix exclusifs, inclusifs, conjonctifs avec ou sans « choix par défaut » grâce aux **$ELSE** et à la possibilité d'imbriquer les **$IF .. [$ELSE] .. $END** ;
- Comme pour l'usage de l'unique indicateur **with debugging mode**, le programmeur est seul maitre de l'usage de ses constantes de compilation, et non pas celui qui soumet la compilation ;
- Avec l'usage de l'unique indicateur **with debugging mode**, avant d'obtenir l'accès à cette option, un contributeur au texte source pouvait être contraint d'attendre que des modifications plus lourdes au fichier source contenant l'**environment division** aient été effectuées par un autre programmeur.
  On dispose maintenant de plus de facilités. Sans que ce soit obligatoire mais dans le cadre d'une bonne pratique pour une meilleure vue d'ensemble : lorsque c'est possible, en déplaçant les déclarations/définitions des constantes dans un COPY dédié qui ne contiendrait rien d'autre, on disposera d'un COPY très rapidement acquis (checkout) modifié (edit, save) et remis (checkin) à la disposition des contributeurs. Pour activer ou désactiver sa ou ses constantes de compilation, fini les longues attentes de disponibilité d'un code comme ça pouvait être le cas. Si malgré tout ce COPY devait être inaccessible, ou si ce regroupement ne devait pas avoir été mis en place, il sera toujours possible au programmeur de définir où bon lui semble la constante de compilation dont il aurait besoin.

Il reste malgré tout quelques inconvénients :

- Dans un même texte source, une constante de compilation définie pour un utilisateur l'est pour tous, avec la même valeur ; L'usage temporaire d'une version locale d'un COPY pourra y palier ;
- Le fait que la valeur d'une constante définie ne puisse évoluer en cours de compilation ;
- Il n'est pas garanti qu'on puisse tester dans un **$IF** la valeur d'une constante sans avoir préalablement vérifié que la constante est définie ; comme il n'y a pas de **AND**, de **OR** ni de **()**, il faudra systématiquement avoir recours à des **$IF** imbriqués pour d'abord tester l'existence d'une constante, et ensuite si l'existence est confirmée, une succession de **$IF** … **$END** par valeur éventuellement envisagée ; ce qui s'avèrera vite être lourd à programmer, et perturbateur pour la lecture du code.
- Comme il n'y a pas de **AND**, de **OR** ni de **()** il faudra systématiquement avoir recours à des **$IF** imbriqués pour simuler des expressions conditionnelles composées de plusieurs constantes; ce qui s'avèrera vite être très très lourd à programmer … *mais ce besoin a-t-il été estimé « nécessaire » dans l'usage envisagé de cette technique ?*
- A moins d'aller dans le(s) COPY's où sont déclarées les constantes de compilation pour le débogage et y remplacer les **$** par des ***** avant d'aller en production, ou alors d'avoir une valeur spécifique qui sera testée systématiquement dans un **$IF** imbriqué au **$IF** testant l'existence, il est difficile de garantir ce qui se passera en production.

Dans les extraits de code ci-dessous :

- Le programme **BIG-ONE** et quelques COPY's qu'il utilise. On garde l'option **with debugging mode** parce qu'elle est compatible, toujours possible et utile pour les directives.
- Comme évoqué plus haut on regroupe, tant que faire se peut, la gestion des constantes de compilation conditionnelle dans un COPY (ici **BIG-ONE/DISPLAY-TO-SET.cpy**) qui est copié dans le texte du programme pour piloter la partie conditionnelle de la compilation.
  On trouve deux versions de ce COPY:

- Celle de gauche, plus lourde mais très flexible, se structure en deux parties :
  + Les choix indépendants du débogage et de la personne qui compile,
  + ensuite, en deux sous-parties :
    - les choix de débogage de chaque contributeur, en fonction de ce qu'il modifie ou l'intéresse d'obtenir pour contrôler son travail ;
    - le calcul des constantes de débogages à positionner pour tous les COPY's, effectué sur base des choix individuels, du soumissionnaire de la compilation, et protégé contre l'activation accidentelle en PROD*uction*.
      Diverses successions (**OR**) de **$IF** imbriqués (**AND**) permettent de calculer s'il faut activer, ou non, des constantes telles **TRACE-ABSOLU** ou **TRACE-TOTAL**. Chacun qui compilera obtiendra donc un exécutable avec ses seuls choix de débogage.

  Pour l'exemple, **TRACE-DATA-ACCESS** est encore actuellement considérée comme vitale pour tous, et même éventuellement en PROD ; elle n'est donc pas encore soumise au choix individuel ni au calcul, mais elle le sera à terme.
- Celui de droite est plus synthétique, probablement plus compréhensible, mais l'exécutable généré sera le même pour tout le monde, même pour la PROD. Il faut changer **$** ⇔ ***** pour activer ou désactiver une constante.

Des versions intermédiaires (p.ex. pas d'individualisation mais protection contre le débogage en PROD) sont évidemment envisageables.

- L'exemple de choix technique entre les **MODULES-ACCOUNT/….cpy** est ici conditionnel. Dans la pratique, on garde rarement ce genre de possibilité, sauf pendant une phase de développement de nouvelle technique pour pouvoir tester des runs parallèles, ou pouvoir réaliser un hot-fix pour la PROD qui utilise encore l'ancienne technique (les fichiers indexés p.ex.) alors qu'en DEV on travaille à l'implémentation de la nouvelle technique (accès DB). Dans l'éventualité d'un tel cas de figure (hot-fix), il eut été plus prudent de « calculer » l'interdiction d'activation en PROD de **USE-DBMS** jusqu'au moment du basculement effectif.
  Remarquons sur la déclaration du fichier indexé et l'incorporation des paragraphes d'accès que la même constante est testée pour intégrer les COPY's concernés.
- Pour l'usage qui en est fait, c'est quasi exclusivement l'existence ou non d'une constante qui est testée ; la valeur des constantes est rarement prise en compte : aller éparpiller des valeurs dans de nombreux COPY's potentiellement utilisés par de nombreux programmes pourrait s'évérer risqué, et constituer une dette technique en cas de changement de valeurs.

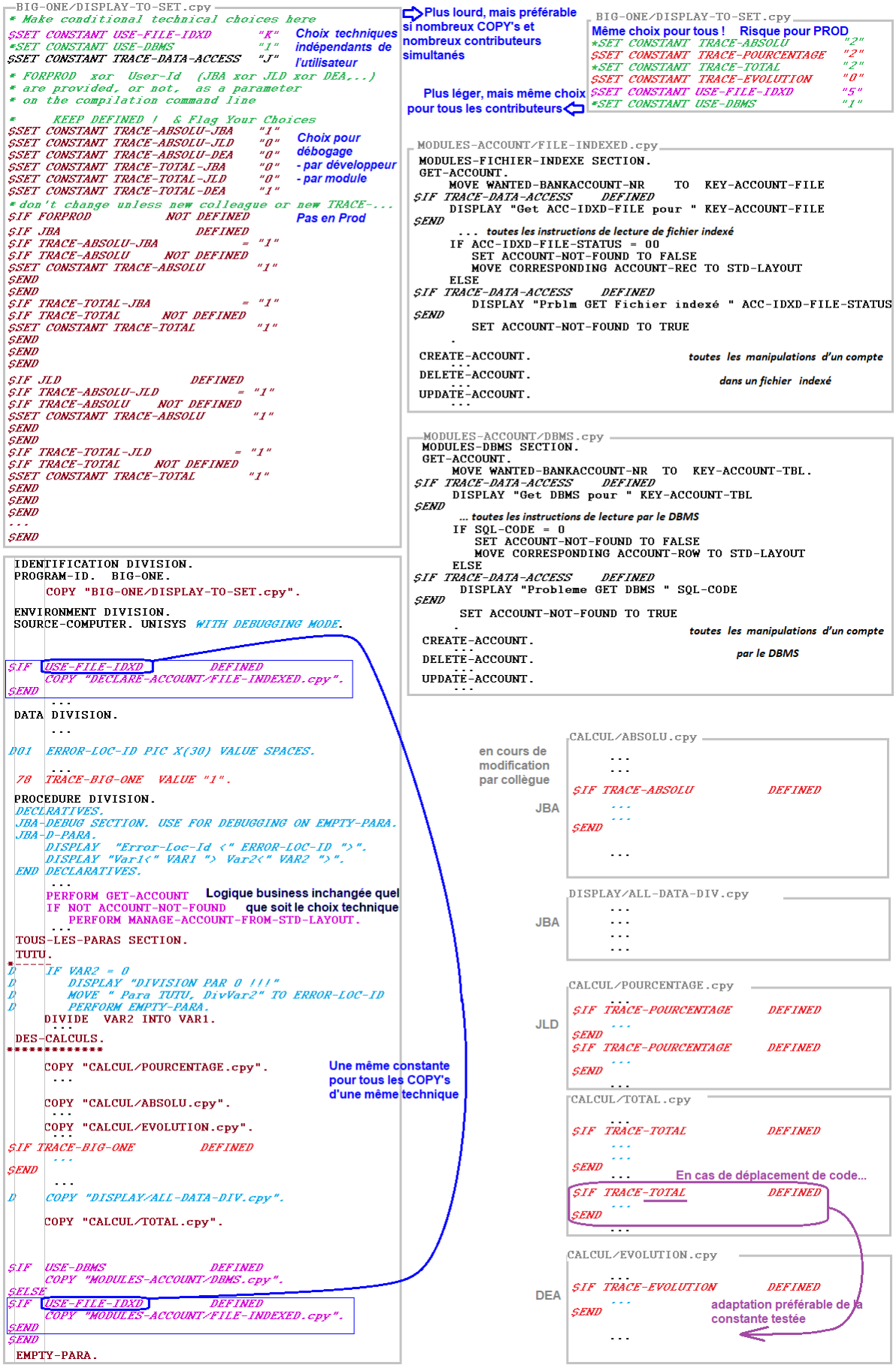


Fig. 10 L'usage des constantes avec $SET et $IF

Non explicitement documenté, mais conséquence probable de l'assimilation de <une constante définie au compilateur par l'instruction **$SET CONSTANTE**> à <une déclaration COBOL de niveau 78>, une telle constante est connue et utilisable tant par le compilateur que par le code COBOL. Comme écrit précédemment, pour effectuer des choix de compilation, le compilateur peut tester l'existence de la définition d'une constante, son contenu dès que son existence est confirmée,… mais le programmeur peut aussi utiliser le nom d'une constante dans une instruction COBOL. Ceci ne sera compilable sans erreur que dans la partie de code suivant la déclaration de la constante par l'instruction **$SET**.

5.3.1.3 Les 52 lettres de HP

Comme la première qui n'utilisait que le '**D**', la finalité de cette technique d'HP utilisant le '**D**' et 25 majuscules de plus, et les 26 minuscules est normalement de faciliter le débogage. En conséquence, l'usage du **with debugging mode** activera en une fois toutes les lignes quelques soient les lettres utilisées puisqu'elles sont censées exister pour du débogage. Il semble préférable d'éviter l'usage de cette option pour éviter que, comme avec la première technique, elle ne se retrouve compilée en production ; au besoin préférons lui les 52 lettres en paramètre de compilation.

A des fins de débogage plus ciblé, l'activation se faisant en choisissant la ou les lettres au moment de lancer la compilation, la technique permet de solutionner le <tout ou rien>.

Remarquons que si des lettres sont fournies en paramètres au lancement de la compilation, elles sont assimilables à des constantes puisqu'elles sont fixées pour toute la compilation, et aucune autre ne pourra s'ajouter durant le processus.

La technique n'a pas été conçue pour mais son détournement à d'autres fins est possible : dès que l'on veut faire plus que du débogage, tel que de la compilation conditionnelle pour des choix fonctionnels (choisir la langue de l'utilisateur ou la langue de sa région de résidence) ou techniques (choisir entre des méthodes d'accès exclusives) cette technique montre des risques. Certains d'entre eux peuvent heureusement être gardés sous contrôle en ayant recours à des trucs et astuces comme cela a été évoqué lors de la présentation de la méthode. Ces trucs et astuces utilisant les définitions de variables COBOL, éventuellement en combinaison avec des **DISPLAY**, permettent de surveiller des règles de présences simultanées ou non d'options activées ; elles ne permettent pas, et c'est une limitation majeure de la technique, de formuler une expression conditionnelle composée pour déterminer le souhait de compiler ou non des instructions.
La technique montre aussi un avantage, il n'y a pas de déclaration ni d'instruction de compilation conditionnelle insérées dans les lignes de code qui en rendraient la lecture moins fluide : tout est déterminé à partir de la colonne 7. La logique fonctionnelle du programme apparait très lisiblement ici sur 3 lignes, et l'aspect conditionnel sur 2 lignes de COPY. Pas de **$SET** ni de **$IF** … **$END** parasite.

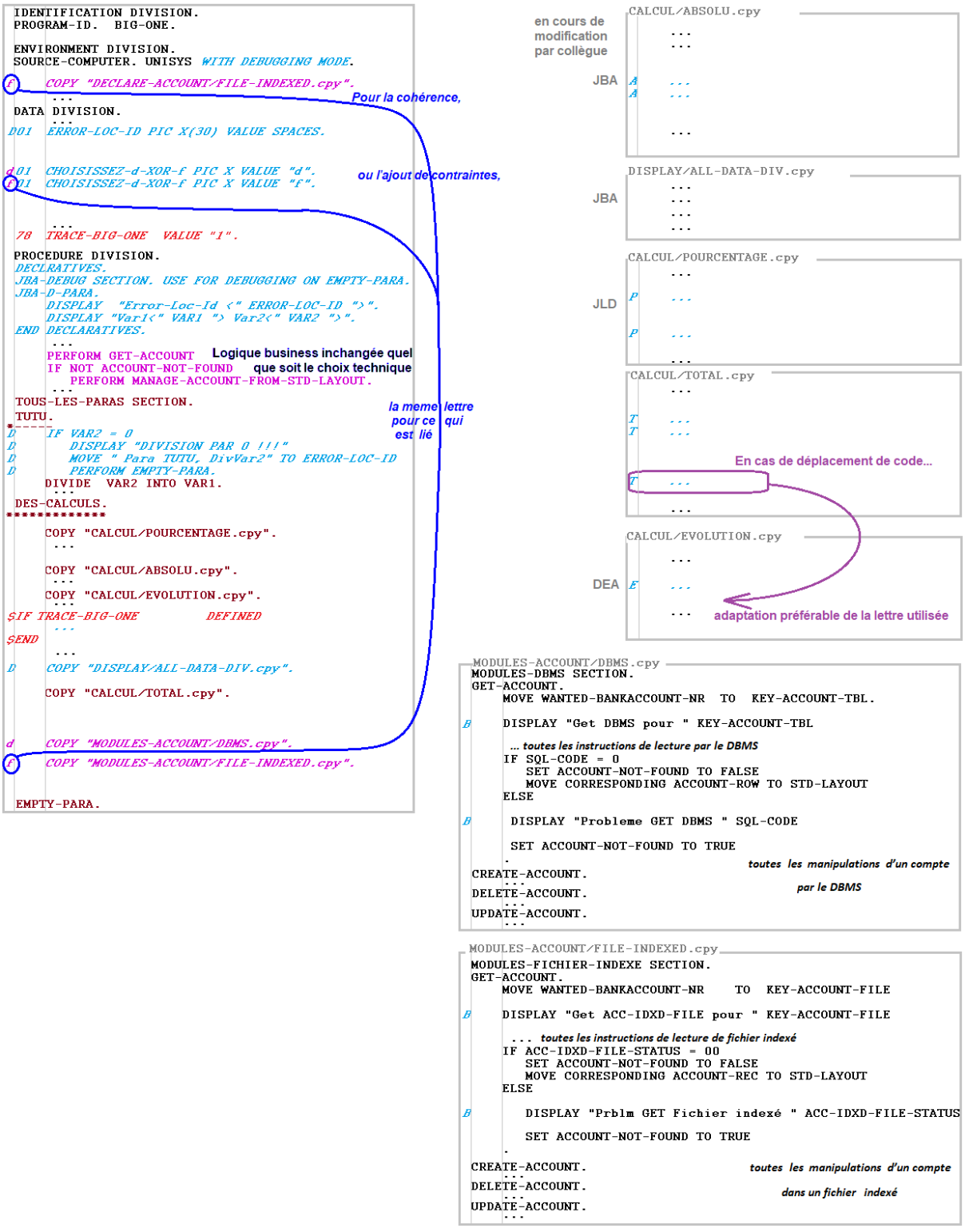


Fig. 11 Les 52 lettres de HP

Et si demain les méthodes d'accès homonymes ne sont plus exclusives mais doivent cohabiter dans un même programme, on pourra simplement avoir recours à la qualification du nom de paragraphe à exécuter en précisant le nom de la section qui le contient.

Des simples lettres comme critère pour du débogage ou des choix, c'est déjà possible bien que limité, très simple à coder, mais nécessite beaucoup de rigueur et de coordination des intervenants (les programmeurs entre eux, et finalement le responsable de la compilation en production). Il appartient à

chaque société utilisant cette technique de mettre en place des règles précises pour éviter des enchevêtrement d'usage de lettres ; par exemple : les consonnes majuscules pour le débogage, les voyelles minuscules pour les choix fonctionnels, les consonnes minuscules pour les choix techniques, les voyelles majuscules pour des situations temporaires, ... par exemple aussi : la structuration des noms de variables pour valider les XOR, AND dans les trucs et astuces de la méthode, …

Les aspects <casse-tête> apparaitront d'autant plus vite que le nombre de programmes se partageant l'usage de COPY's augmentera… or au plus on factorise de code pour éviter les redondances, au plus on a de COPY's, et au plus de programmes en utilisent.

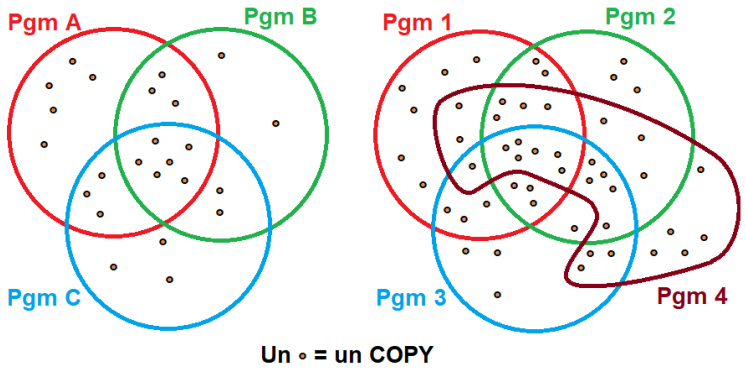


Fig. 12 Casse-tête pour l'usage des 52 lettres si multiplication des copy's

(Dans le texte source, la même lettre peut être utilisée pour déterminer l'inclusion ou non des COPY's liés : celui/ceux de déclarations nécessaires pour celui d'instructions à incorporer au texte du programme, comme par exemple la lettre f détermine l'inclusion de la déclaration du fichier indexé et l'inclusion du COPY avec les paragraphes d'instructions manipulant le fichier).

5.3.1.4 Les piles de booléens de Burroughs (Unisys)

Développée pour ses compilateurs XALGOL et ALGOL, Burroughs a implémenté les mêmes fonctionnalités (directive **`OMIT`** avec les commandes **`$SET`**|**`$RESET`**|**`$POP`**) de compilation conditionnelle dans son compilateur COBOL dès 1974.
Bien qu'elle leur soit antérieure, elle contenait déjà des facilités qui manquent aux autres techniques évoquées jusqu'à présent :

- On ne travaille pas avec des constantes mais des « variables » (empilement/dépilement de valeurs booléennes)
- On a la possibilité d'utiliser des opérateurs booléens (**`AND`**, **`OR`**, **`()`**, **`NOT`**) pour constituer des expressions booléennes
- Une expression booléenne peut utiliser comme opérande une variable de compilation non définie, si elle n'existe pas, elle sera considérée comme ayant la valeur FALSE ;
- Dans un COPY dédié aux variables de compilation voulues, là comme ailleurs, chacun peut utiliser son identifiant de développeur (JBA, JLD…) pour activer « à la carte » ce qu'il souhaite, sans impacter les autres.
- Il y a pour la production une protection possible contre de résiduelles instructions de débogage :
    - Soit parce que la variable n'est pas définie, donc vaut FALSE, sa négation fera alors que **`OMIT`** sera TRUE, et les instructions ne seront donc pas compilées.
    - Soit parce qu'on le programme explicitement
- La bonne gestion des directives **`LIST`**, **`LISTDOLLAR`** et **`LISTOMITTED`** permet à la fois un débogage aisé de la compilation conditionnelle lorsqu'elles sont toutes trois à TRUE, et une lecture fluide du seul texte du programme lorsque seule **`LIST`** est à TRUE.

- Maintenant l'usage de sa syntaxe propre des **OMIT** pour la rétrocompatibilité, Unisys a intégré à la version 85 de son compilateur COBOL la syntaxe d'autres éditeurs faisant usage des **$IF** … **[$ELSE** …**]** **$END** ; Il accorde aux **$IF** les capacités d'expressions conditionnelles (**AND**, **OR**, **()**, **NOT**) dont il disposait déjà avec **OMIT**, en revanche il maintient le seul type booléen pour la définition des variables de compilations.

Parmi les inconvénients :

- Les opérandes n'ont qu'un seul type autorisé : booléen
- Le concept **OMIT condition** à TRUE fait ignorer (omettre) ce qui suit, ce qui est l'opposé du concept usuel **IF condition** à TRUE qui fait prendre en compte ce qui suit.
- L'absence d'une instruction semblable à un **ELSE** pour le **OMIT** est regrettable …
  … mais se solutionne très facilement :

```
$SET OMIT = longue-condition-bien-compliquée
     DISPLAY "Condition non remplie, on compile ceci"
$SET OMIT = NOT OMIT
     DISPLAY "Condition remplie, on compile ce code ci"
$POP OMIT OMIT
```

  Il suffit de bien penser à dépiler DEUX valeurs empilées de OMIT
- C'est un avantage puisqu'il permet de simuler un ELSE, mais aussi un inconvénient, en tout cas un risque qui peut ne pas faciliter la compréhension : ***toute CCR rencontrée, même dans un bloc OMITTED, sera interprétée.*** Ci-dessous Operande-A et Operande-B sont gérées de la même manière par le processus de compilation malgré la permutation des lignes **RESET** et **POP** et ***dans tous les cas,*** les deux se verront empiler une nouvelle valeur FALSE quelles ***qu'étaient leurs existence ou valeur respectives initiales.***

```
$SET OMIT = operande-A OR operande-B
$SET OMIT = NOT operande-A
* Operande-A est à TRUE, => on la force à False, puis POP OMIT
$RESET operande-A
$POP OMIT
$SET OMIT = NOT operande-B
* Operande-B est à TRUE, => POP, puis on le force à False
$POP OMIT
$RESET operande-B
$POP OMIT
```

  Ci-dessous on effectuera bien un **OMIT** si la **Condition-A** est vérifiée, mais l'émulation du **ELSE** (empilement d'une valeur **OMIT** négation de la précédente) sera annihilé par l'empilement d'une troisième valeur de **OMIT** basée exclusivement sur **Condition-B**

```
$SET OMIT = Condition-A
     DISPLAY "Condition-A non remplie, on compile ceci"
$SET   OMIT = NOT OMIT
$SET     OMIT = Condition-B
           DISPLAY "Condition-B non remplie, on compile ceci"
$POP     OMIT
$POP   OMIT
$POP OMIT
```

  Le deuxième **OMIT** empilé comme équivalent du **ELSE** ne servant à rien suite à l'empilement d'un troisième, le code ci-dessous donne le même résultat

```
$SET OMIT = Condition-A
      DISPLAY "Condition-A non remplie, on compile ceci"
```

```
$POP OMIT
$SET OMIT = Condition-B
      DISPLAY "Condition-B non remplie, on compile ceci"
$POP OMIT
```

Pour garantir l'exhaustivité des deux codes, on doit éviter les empilements erronés de valeur en procédant comme suit (une sorte de **ELSE-IF**) :

```
$SET OMIT = Condition-A
     DISPLAY "Condition-A non remplie, on compile ceci"
$SET OMIT = NOT OMIT  OR  Condition-B
     DISPLAY "Condition-B non remplie, on compile ceci"
$POP OMIT OMIT
```

Heureusement Unisys dispose de la seconde syntaxe évoquée et cohabitante avec la première, mais sans empilement de valeur :

```
$SET OMIT = une-condition              $IF  une-condition-contraire
…                                      …
$SET OMIT = NOT OMIT                   $ELSE
…                                      …
$POP OMIT OMIT                         $END
```

Pour garder une certaine standardisation, le déplacement de code de débogage d'un COPY (p.ex. **TOTAL**) à un autre (p.ex. **EVOLUTION**) impliquera une charge de travail (ici : procéder au renommage de **TRACE-TOTAL** en **TRACE-EVOLUTION**) des conditions déplacées).

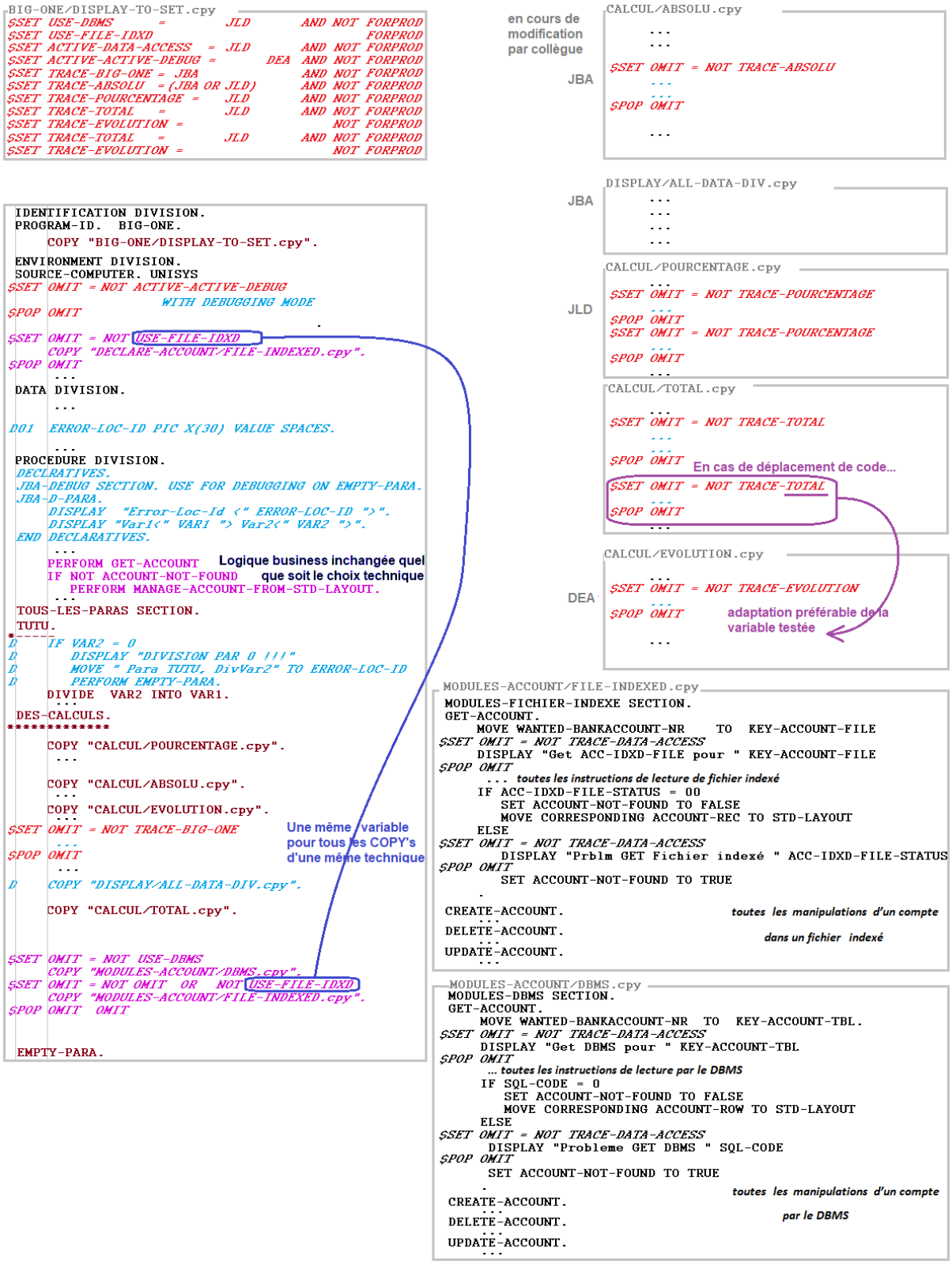


Fig. 13 La compilation conditionnelle selon Unisys : $SET OMIT = condition Exemple V1

Remarquons dans le COPY …**`/DISPLAY-TO-SET.cpy`** (*rassemblant l'activation des variables de compilation dédiées au débogage*) que chaque programmeur y précise ses choix ciblés grâce aux trois lettres de son identifiant de telle sorte que, lorsqu'il effectuera 'sa' compilation, il n'activera que les activités de débogage demandées. Remarquons aussi le **`NOT FORPROD`** qui forcera le FALSE de ces variables de compilations lorsqu'on compilera pour la production. Le compilateur n'a pas la notion

que les variables JBA, JLD, DEA … sont des identifiants, il n'a pas la notion que **FORPROD** est utilisé lorsqu'on compile pour la production, ce pourrait être **AZERTY** ou simplement **P**, pour lui ce sont des variables comme les autres, considérant à TRUE celles qu'il reçoit à la soumission de la compilation, à FALSE les autres. Il appartient au programmeur de passer son identifiant lorsqu'il demande une compilation, et à l'opérateur de production de passer **FORPROD** lorsqu'il compile pour la production.

L'usage standardisé d'une variable de compilation, dénommée **NODISPLAY** dans l'exemple ci-après, permet de solutionner le problème de code qui passerait d'un COPY à un autre. De même si un COPY devait être divisé en plusieurs morceaux, il suffirait de définit une nouvelle variable de compilation par nouveau COPY, puis ensuite, basé sur la négation de cette nouvelle variable, empiler une nouvelle valeur de **NODISPLAY** au début de chaque COPY, et de la dépiler en fin de COPY ; c'est ce que, dans le schéma suivant, on a appliqué pour les deux COPY's **MODULES-ACCOUNT/….cpy**.
(Si ça semble très « lourd », c'est parce que c'est, sur ce schéma, très condensé.)

BIG-ONE/DISPLAY-TO-SET.cpy

```
$SET USE-DBMS      =          JLD        AND NOT FORPROD
$SET USE-FILE-IDXD                       AND NOT FORPROD
$SET ACTIVE-DATA-ACCESS   = JLD           AND NOT FORPROD
$SET ACTIVE-ACTIVE-DEBUG =          DEA   AND NOT FORPROD
$SET TRACE-BIG-ONE = JBA                  AND NOT FORPROD
$SET TRACE-ABSOLU  = (JBA OR JLD)         AND NOT FORPROD
$SET TRACE-POURCENTAGE =    JLD           AND NOT FORPROD
$SET TRACE-TOTAL   =        JLD           AND NOT FORPROD
$SET TRACE-EVOLUTION =                    NOT FORPROD
$SET TRACE-TOTAL   =        JLD           AND NOT FORPROD
$SET TRACE-EVOLUTION =                    NOT FORPROD
```

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID.  BIG-ONE.
      COPY "BIG-ONE/DISPLAY-TO-SET.cpy".
 ENVIRONMENT DIVISION.
 SOURCE-COMPUTER. UNISYS
$SET OMIT = NOT ACTIVE-ACTIVE-DEBUG
                    WITH DEBUGGING MODE
$POP OMIT
                                .
$SET OMIT = NOT USE-FILE-IDXD
      COPY "DECLARE-ACCOUNT/FILE-INDEXED.cpy".
$POP OMIT
       ...
 DATA DIVISION.
       ...
D01   ERROR-LOC-ID PIC X(30) VALUE SPACES.
       ...
 PROCEDURE DIVISION.
 DECLARATIVES.
 JBA-DEBUG SECTION. USE FOR DEBUGGING ON EMPTY-PARA.
 JBA-D-PARA.
      DISPLAY  "Error-Loc-Id <" ERROR-LOC-ID ">".
      DISPLAY "Var1<" VAR1 "> Var2<" VAR2 ">".
 END DECLARATIVES.
       ...
      PERFORM GET-ACCOUNT    Logique business inchangée quel
      IF NOT ACCOUNT-NOT-FOUND    que soit le choix technique
         PERFORM MANAGE-ACCOUNT-FROM-STD-LAYOUT.
       ...
 TOUS-LES-PARAS SECTION.
 TUTU.
*-----
D     IF VAR2 = 0
D        DISPLAY "DIVISION PAR 0 !!!"
D        MOVE " Para TUTU, DivVar2" TO ERROR-LOC-ID
D        PERFORM EMPTY-PARA.
      DIVIDE  VAR2 INTO VAR1.
       ...
 DES-CALCULS.
**************
      COPY "CALCUL/POURCENTAGE.cpy".
       ...
      COPY "CALCUL/ABSOLU.cpy".
       ...
      COPY "CALCUL/EVOLUTION.cpy".
       ...
$SET OMIT = NOT TRACE-BIG-ONE
       ...
$POP OMIT
       ...
D     COPY "DISPLAY/ALL-DATA-DIV.cpy".

      COPY "CALCUL/TOTAL.cpy".



$SET OMIT = NOT USE-DBMS
      COPY "MODULES-ACCOUNT/DBMS.cpy".
$SET OMIT = NOT (OMIT  AND  USE-FILE-IDXD)
      COPY "MODULES-ACCOUNT/FILE-INDEXED.cpy".
$POP OMIT
$POP OMIT

 EMPTY-PARA.
```

en cours de modification par collègue

JBA

CALCUL/ABSOLU.cpy

```
$SET NODISPLAY = NOT TRACE-ABSOLU
      ...

$SET OMIT = NODISPLAY
      ...
      ...
$POP OMIT

      ...
$POP NODISPLAY
```

JBA

DISPLAY/ALL-DATA-DIV.cpy

```
      ...
      ...
      ...
      ...
```

JLD

CALCUL/POURCENTAGE.cpy

```
$SET NODISPLAY = NOT TRACE-POURCENTAGE

      ...

$SET OMIT = NODISPLAY
      ...
$POP OMIT
$SET OMIT = NODISPLAY
      ...
$POP OMIT

      ...
$POP NODISPLAY
```

CALCUL/TOTAL.cpy

```
$SET NODISPLAY = NOT TRACE-TOTAL
      ...
$SET OMIT = NODISPLAY
      ...
      ...
$POP OMIT
```

En cas de déplacement de code...

```
$SET OMIT = NODISPLAY
      ...
$POP OMIT
      ...
$POP NODISPLAY
```

DEA

CALCUL/EVOLUTION.cpy

```
$SET NODISPLAY = NOT TRACE-EVOLUTION
      ...

$SET OMIT = NODISPLAY
      ...
$POP OMIT

      ...
$POP NODISPLAY
```

rien ne change d'un point de vue compilation conditionnelle

MODULES-ACCOUNT/FILE-INDEXED.cpy

```
$SET NODISPLAY = NOT TRACE-DATA-ACCESS
 MODULES-FICHIER-INDEXE SECTION.
 GET-ACCOUNT.
      MOVE WANTED-BANKACCOUNT-NR     TO  KEY-ACCOUNT-FILE
$SET OMIT = NODISPLAY
      DISPLAY "Get ACC-IDXD-FILE pour " KEY-ACCOUNT-FILE
$POP OMIT
       ... toutes les instructions de lecture de fichier indexé
      IF ACC-IDXD-FILE-STATUS = 00
         SET ACCOUNT-NOT-FOUND TO FALSE
         MOVE CORRESPONDING ACCOUNT-REC TO STD-LAYOUT
      ELSE
$SET OMIT = NODISPLAY
         DISPLAY "Prblm GET Fichier indexé " ACC-IDXD-FILE-STATUS
$POP OMIT
         SET ACCOUNT-NOT-FOUND TO TRUE
      .
 CREATE-ACCOUNT.                          toutes les manipulations d'un compte
       ...
 DELETE-ACCOUNT.                                dans un fichier indexé
       ...
 UPDATE-ACCOUNT.
       ...
$POP NODISPLAY
```

MODULES-ACCOUNT/DBMS.cpy

```
$SET NODISPLAY = NOT TRACE-DATA-ACCESS
 MODULES-DBMS SECTION.
 GET-ACCOUNT.
      MOVE WANTED-BANKACCOUNT-NR   TO  KEY-ACCOUNT-TBL.
$SET OMIT = NODISPLAY
      DISPLAY "Get DBMS pour " KEY-ACCOUNT-TBL
$POP OMIT
       ... toutes les instructions de lecture par le DBMS
      IF SQL-CODE = 0
         SET ACCOUNT-NOT-FOUND TO FALSE
         MOVE CORRESPONDING ACCOUNT-ROW TO STD-LAYOUT
      ELSE
$SET OMIT = NODISPLAY
         DISPLAY "Probleme GET DBMS " SQL-CODE
$POP OMIT
         SET ACCOUNT-NOT-FOUND TO TRUE
      .                                   toutes les manipulations d'un compte
 CREATE-ACCOUNT.                                  par le DBMS
       ...
 DELETE-ACCOUNT.
       ...
 UPDATE-ACCOUNT.
       ...
$POP NODISPLAY
```

Fig. 14 La compilation conditionnelle selon Unisys : $SET OMIT = condition Exemple V2

Il est précisé dans le manuel que par défaut l'ensemble des piles peut totaliser 6000 mots, qu'on peut l'étendre jusqu'à 64000 mots ; même si cela semble beaucoup, il est important de bien gérer ses variables de compilation. Ainsi par exemple : sachant qu'elle peut être rencontrée de nombreuses fois à TRUE ou FALSE, si on souhaite détecter avoir rencontré au moins une fois une variable de compilation spécifique appelée **VariableSpecifique** lorsqu'elle valait TRUE, l'usage systématique de
$SET AuMoinsUneFois = AuMoinsUneFois OR VariableSpecifique
empilant chaque fois une nouvelle valeur, risque de faire atteindre la limite de sa pile. Une solution nécessite un peu plus de code pour juste mémoriser une valeur, sans accroitre la pile :

```
$SET Tempo = AuMoinsUneFois OR VariableSpecifique
$POP AuMoinsUneFois
$SET AuMoinsUneFois = Tempo
$POP Tempo
```

Il n'y aura qu'une et une seule occurrence de **AuMoinsUneFois**, et celle-ci prendra la valeur TRUE si et seulement si le processus de compilation aura rencontré au moins une occurrence de **VariableSpecifique** valant TRUE
(Ceci sera utilisé dans l'émulation au mieux de l'OO pour cet éditeur.)

Pour le listing de compilation produit, cet éditeur de compilateur COBOL offre également la possibilité de bien paramétrer ce que l'on souhaite y voir : on peut forcer l'affichage de toutes les lignes de contrôles du compilateur (**$SET, $RESET, $POP**), ou exiger de ne pas les voir ; on peut exiger de voir lister tous les scénarii -*ceux n'étant pas compilés étant suffixés de OMIT*- ou de ne voir que celui ayant contribué à produire l'exécutable.

5.3.1.5 La norme ISO/IEC 1989 :2002

La dernière technique de compilation conditionnelle implémentée en COBOL est celle définie dans la norme ISO/IEC 1989 :2002. Grâce à ses nombreuses possibilités <définition de variables numériques, booléennes ou chaînes de caractères, ou dont le contenu est reçu en paramètre, adaptation de leur contenu et possibilité d'évaluation pour les booléens ou de calcul pour les numériques, suppression de la définition de variables, test sur l'existence de la définition d'une variable, condition composée (**AND**, **OR**, **()**, **NOT**, relationnelle, appartenance à un intervalle numérique>), cette technique de compilation conditionnelle peut se substituer à toutes les autres. En outre, elle bénéficie de la coloration syntaxique dans l'IDE (parfois avec retard, suivant la complexité du code, l'imbrication des COPY, l'évaluation des conditions,…) ce qui permet de rapidement visualiser ce qu'il en sera de la compilation étant donné les variables de compilation ; l'analyse du listing de compilation permet également d'être certain du code ayant servi à produire l'exécutable.

Cette norme prévoyant trois types possibles (numérique, booléen, chaine de caractères) de variables, il est interdit d'en utiliser une sans qu'elle ait été définie -quel serait son type ?- Il est heureusement possible de tester si une variable a été définie ou non, mais malheureusement impossible de déterminer le type utilisé pour la définir, ce qui, à défaut d'une bonne coordination entre le programmeur et le responsable de la compilation, est problématique pour celles passées comme paramètres de compilation. L'usage d'une variable de compilation non définie provoquera une erreur de compilation, il en est de même si une variable définie est utilisée en lien (opération, comparaison) avec un autre type. Ceci ne devrait pas perturber les programmeurs COBOL habitués à déclarer leurs variables COBOL avant tout usage.

Avant toute chose, relevons qu'il n'est pas précisé si l'évaluation d'une condition de pré compilation peut être arrêtée dès que le résultat est certain, ou si l'évaluation sera toujours complètement exécutée au risque de provoquer une erreur. C'est un test avec la version 8 du compilateur COBOL de Micro

Focus qui a permis de déterminer que la condition est toujours complètement évaluée,… mais cela dépend peut-être de l'éditeur du compilateur (?)
Ainsi, une instruction **>>IF (BOOL-X NOT DEFINED) OR BOOL-X** provoquera une erreur de compilation si **BOOL-X** n'est pas défini alors que le résultat final 'true' aurait pu être exploité. On n'utilisera donc pas ce genre de condition.

On peut donner un sens booléen à l'existence, ou la non-existence, de la définition d'une variable : cela fonctionnera,… mais le code sera risqué, lourd et peu lisible, surtout lorsqu'on voudra écrire une condition complexe pour déterminer la valeur d'une nouvelle variable de compilation.
En reprenant de l'exemple exposée pour Burroughs le souhait d'obtenir les display's dans le COPY **CALCUL/ABSOLU.cpy**, où on y déterminait

```
$SET TRACE-ABSOLU= (JBA OR JLD) AND NOT FORPROD
```

Obtenir le même résultat pourrait se coder :

```
>>IF (JBA DEFINED OR JLD DEFINED) AND FORPROD NOT DEFINED
>>DEFINE TRACE-ABSOLU as (true)
>>END-IF
```

pour être ensuite utilisé

```
>>IF TRACE-ABSOLUT DEFINED
…
>>END-IF
```

La tentation sera grande d'écrire plus simplement (en tenant compte de la limitation des colonnes)

```
>>DEFINE OPERAND1      AS (JBA DEFINED OR JLD DEFINED) OVERRIDE
>>DEFINE TRACE-ABSOLU AS (OPERAND1 AND FORPROD NOT DEFINED) OVERRIDE
>>DEFINE OPERAND1      AS OFF
```

mais dans ce cas, sous risque de résultat autre qu'attendu, on ne pourra plus tester l'existence ou non de **TRACE-ABSOLU** puisque la variable sera toujours définie et affectée d'un contenu qui sera 'true' ou 'false'. On va donc se retrouver face à des situations ambiguës où parfois il faudra tester l'existence, parfois la valeur. Afin d'éviter cette situation, et dans la bonne habitude COBOL, il semble préférable de toujours définir les variables de compilation dans un, deux ou 𝒳 COPY's spécifiques dédiés.

Le programme BIG-ONE dispose de fonctionnalités activables « à la carte », sa variabilité permet de générer de nombreux variants constitués de combinaisons de fonctionnalités différentes.

Garder dans la source mais non-compilées les instructions de débogage pour un prochain cycle de modifications et de test est préconisé par Epstein [60].

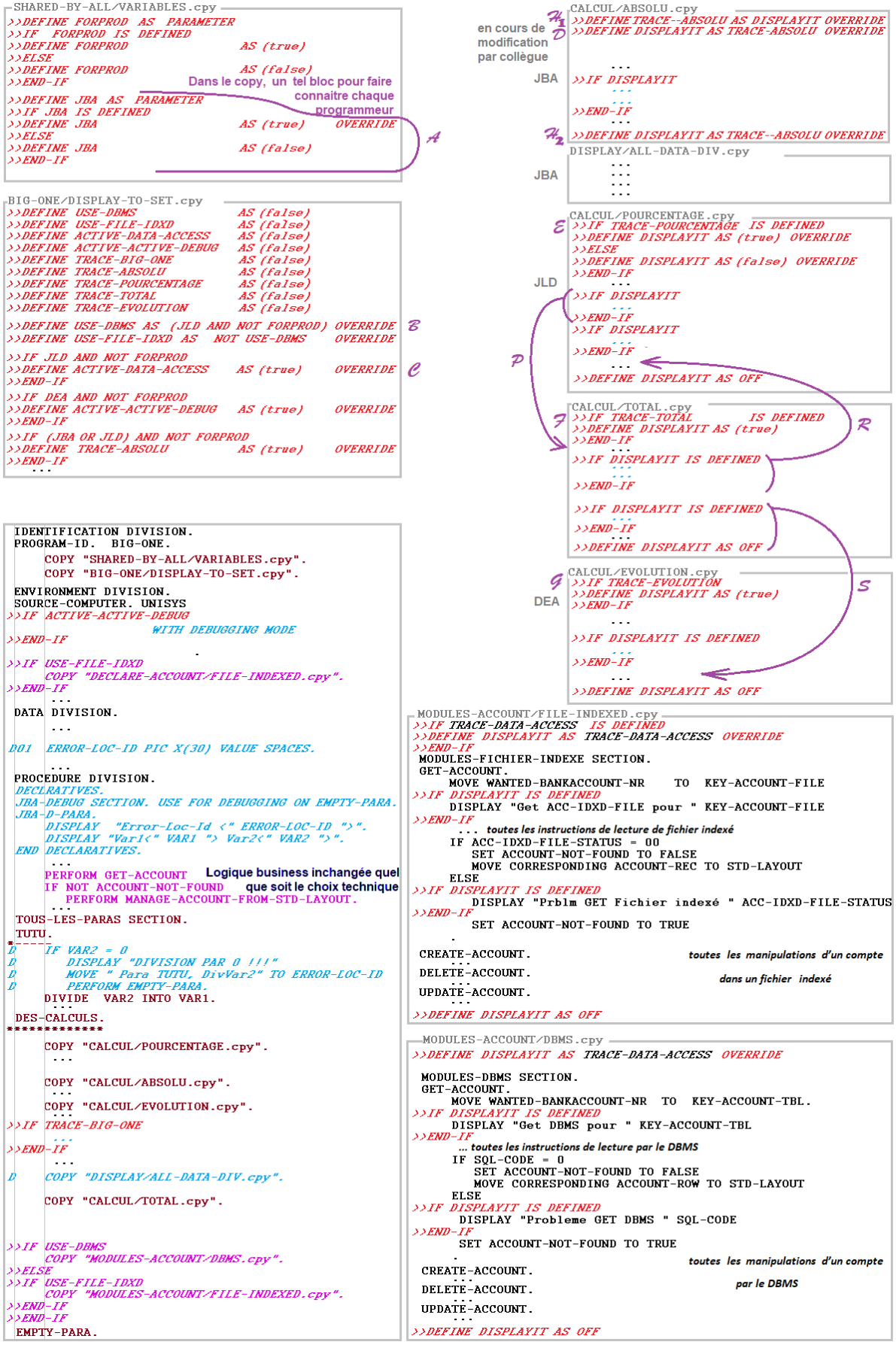


Fig. 15 La compilation conditionnelle selon la norme COBOL 2002 (exemple Unisys V2 transposé)

Exemple :

- Prévoyons un même COPY (p.ex. **`SHARED-BY-ALL/VARIABLES.cpy`**) qui sera à intégrer au début *de tous les programmes*, il contiendrait la définition des variables permettant de déterminer, sur base des paramètres de compilation, le contexte de compilation (pour Production (validation ? Test ?,…) ou non) ainsi que l'utilisateur lançant la compilation ; le résultat sera une série de variables booléennes, plus simple à utiliser dans les évaluations

```
>>DEFINE FORPROD AS PARAMETER
>>IF FORPROD IS DEFINED          On reçoit le paramètre … dont on ignore le type effectif.  MAIS…
>>DEFINE FORPROD AS (true)      ~~override~~ → valide que, si défini, c'est bien ainsi, tel que prévu
>>ELSE
>>DEFINE FORPROD AS (false)                  puisqu'il n'était pas défini, définissons le
>>END-IF
```

et, pour chaque (!) *programmeur* existant (par exemple ici *JBA*) un bloc (*A*)similaire à celui-ci

```
>>DEFINE JBA AS PARAMETER
>>IF JBA IS DEFINED                          On reçoit un paramètre … dont on ignore le type
>>DEFINE RECEIVED-JBA AS JBA OVERRIDE        Éventuellement sauvegardons ce qu'on a reçu
>>DEFINE JBA AS (true) OVERRIDE                       forçons le type booléen pour la suite
>>ELSE
>>DEFINE JBA AS (false)                      puisqu'il n'était pas défini, définissons le
>>END-IF
```

- Un deuxième COPY qui ne serait plus partagé par tous les programmes, mais propre à chacun d'entre eux (p.ex. `nom-du-programme/DISPLAY-TO-SET.cpy`), définirait et fixerait une valeur par défaut à toutes les variables non potentiellement reçues en paramètres et utilisées dans le texte source.
  Dans une seconde partie de ce COPY (ou dans un troisième COPY également propre à chaque programme), on « calcule » la valeur de chacune en fonction des paramètres reçus, ou des valeurs par défaut.
  Deux manières :

```
>>DEFINE USE-DBMS AS (JLD AND NOT FORPROD) OVERRIDE                                    B
```

USE-DBMS a été déclaré dans la première partie de ce COPY, avec la valeur 'false' ;
si ce n'est pas JLD qui compile, USE-DBMS recevra la valeur 'false' et l'exécutable continuera à utiliser la méthode éprouvée d'accès aux fichiers indexés tel que c'est le cas en production ;
si c'est JLD qui compile en développement la source qu'il adapte pour accéder à une DB plutôt qu'à des fichiers indexés, USE-DBMS recevra la valeur 'true' … sa compilation respectera ce scénario et l'exécutable qu'il produira ne perturbera que lui en cas de mauvais fonctionnement. L'inconvénient de ce format est que l'IDE ne permet pas, à la lecture de cette définition, de déduire la valeur attribuée à USE-DBMS.

```
>>IF JLD AND NOT FORPROD                                                  semblable à C
>>DEFINE USE-DBMS AS (true) OVERRIDE
>>END-IF
```

Le résultat sera le même que dans le format précédent, la différence se situe au niveau visuel dans l'IDE puisque si l'évaluation de la condition est 'false', l'IDE attribuera une couleur telle qu'on visualisera que le **`>>DEFINE`**… est ignoré ; ou dans le listing de compilation on verra que le **`>>DEFINE`**… est considéré comme une ligne de commentaire.

Lorsque toutes les implémentations de code demandées à JLD auront été finalisées, testées par lui et seront parfaitement fonctionnelles, on pourra les faire tester par la globalité des programmeurs en utilisant

```
>>DEFINE USE-DBMS AS NOT FORPROD OVERRIDE
```

ou encore

```
>>IF NOT FORPROD
>>DEFINE USE-DBMS AS (true) OVERRIDE
>>END-IF
```

Et pour le passage en production il suffira d'adapter dans **BIG-ONE/DISPLAY-TO-SET.cpy**

```
>>DEFINE USE-DBMS AS (true) OVERRIDE
```

jusqu'à ce qu'on nettoie le code des accès aux fichiers indexés pour ne garder que celui des accès DB.

Cette seconde partie du deuxième COPY (ou ce troisième COPY) spécifique à chaque programme peut contenir, pour tout le reste du texte source, des évaluations exprimant des choix techniques tels que donné en exemple ci-dessus, il peut contenir des choix pour le debugging (**TRACE-**…, **debugging mode**, …)

- Pour simplifier l'implémentation, l'identification, la mobilité *du code de débogage*, dans tous les COPY's, on préconise l'usage d'une et une seule même variable de compilation pour tous les COPY's (dans les exemples : **DISPLAYIT**); l'existence ou la valeur de cette variable de compilation étant déterminée au début du COPY, et sa portée jusqu'à la fin du COPY. Ici encore, il est préférable (et hautement préconisé) de ***toujours*** définir la variable (tel que dans 𝒟 ou ℰ), et d'en tester la valeur plutôt que d'en tester l'existence comme on le fait après ℱ. En effet, si on effectue le déplacement ou une copie 𝒫 vers **CALCUL/TOTAL.cpy** il est possible qu'on teste **DISPLAYIT** alors que celle-ci ne soit pas définie, ce qui provoquera une erreur de compilation.

  Analysons plus finement les différentes déclarations de variables.

  𝒟 Affectera bien à **DISPLAYIT** la valeur calculée pour **TRACE-ABSOLU** ; Si **TRACE-ABSOLU** n'est pas défini, il n'y aura pas de fonctionnement erroné mais une erreur de compilation ; Si **TRACE-ABSOLU** est d'un autre type que booléen, **DISPLAYIT** sera aussi d'un autre type que booléen et les tests **>>IF DISPLAYIT** dans ce COPY génèreront aussi des erreurs de compilation, mais pas de fonctionnement erroné.

  ℰ On est certain que **DISPLAYIT** sera bien défini et de type booléen ; cette variable contiendra par contre une valeur peut-être autre qu'attendue si **TRACE-POURCENTAGE** est défini avec une valeur autre que 'true' ; imaginons que **TRACE-POURCENTAGE** contienne 'false' ou "NO", **DISPLAYIT** contiendra 'true' malgré tout. **DISPLAYIT** étant bien booléen, on n'aura donc pas de faute de compilation, mais peut-être des résultats non souhaités, ce qui peut être nettement pire (surtout dans un cadre autre que le débogage de cet exemple).

  ℱ est tout aussi problématique que ℰ en ce qui concerne le contenu de **TRACE-TOTAL**, avec en plus un postulat peut-être erroné que la définition de **DISPLAYIT** aie bien été supprimée juste avant l'inclusion du COPY **CALCUL/TOTAL.cpy**.

  𝒢 garanti exclusivement que **DISPLAYIT** sera à 'true' si et seulement si **TRACE-EVOLUTION** est 'true' alors que **DISPLAYIT** pourrait être défini et valoir 'true' ou 'false' depuis la fin de **CALCUL/ABSOLU.cpy**.

  $\mathcal{H}_1$ et $\mathcal{H}_2$ permettent de mémoriser puis de restituer la valeur de DISPLAYIT du COPY parent en cas de COPY's imbriqués. Pour être possible au risque, sinon, de provoquer une erreur de compilation, **DISPLAYIT** doit toujours être défini et il est alors recommandé d'utiliser deux instructions semblables à celles-ci dans tous les COPY's. Pour fournir le résultat escompté, chacun des COPY's doit utiliser un nom de variable unique. Cobol n'autorisant pas de

récursivité direct ou indirect dans l'usage des COPY's, une valeur utile mémorisée ne sera pas écrasée ; des inclusions parallèles d'un même COPY ne poseront en revanche aucun problème. Parmi les codes de déclarations envisagés dans les différents COPY's, celui de **`CALCUL/ABSOLU.cpy`** est clairement recommandé.

Analysons la « mobilité » de morceau de code.
Le déplacement *P* posera un problème de compilation si **`DISPLAYIT`** n'est pas défini au moment de l'inclusion de **`CALCUL/TOTAL.cpy`**, et ne l'est pas non plus par *J*.
Le déplacement *R* ne donnera pas obligatoirement le résultat escompté puisque dans **`CALCUL/POURCENTAGE.cpy`** la variable **`DISPLAYIT`** est toujours définie par *E* et les instructions de débogage déplacées seront donc systématiquement exécutées, ce qui n'est pas nécessairement souhaité.
Le déplacement *S* est le seul exemple ici susceptible de fournir le comportement attendu.

Pour ces deux (débogage, passage d'une technologie à une autre) objectifs donnés par les éditeurs en exemples pour l'usage de la compilation conditionnelle, les manières de prévoir les scénarii voulus sont déjà nombreuses ; la définition (et son usage respecté par tous) d'un standard d'entreprise utilisant la compilation conditionnelle s'avèrera une bonne pratique, pour ne pas dire « une pratique vitale ».

5.3.2 Peut-on passer, évoluer d'une compilation conditionnelle à l'autre ?

La problématique de la conversion d'une implémentation de compilation conditionnelle à une autre se pose si la destination n'implémente pas la norme ISO/IEC 1989 :2002. En effet cette dernière, grâce à son large éventail de possibilités, peut produire les résultats obtenus par toutes les autres. Le compilateur Micro Focus Visual COBOL 8.0 (pour Visual Studio) testé gère parfaitement la coexistence des constantes à côté de la norme. Le compilateur IBM Cobol for z/OS (depuis sa version 6.2) incorpore aussi la norme ISO/IEC 1989 :2002 mais pas les instructions $SET, $IF ni les constantes de compilation conditionnelle. En revanche, ce compilateur incorpore la notion de constante au sens COBOL, et les telles constantes peuvent être initialisées par la valeur d'une variable de compilation conditionnelle.

Postulant qu'il soit peu probable qu'on convertisse des applications COBOL utilisant les facilités d'un compilateur vers un COBOL présentant moins de facilités, ce chapitre ne présente qu'un intérêt relatif (par exemple la fusion avec uniformisation du matériel et standardisation des logiciels de plusieurs institutions ayant, au départ, des mainframe et compilateurs d'éditeurs différents). Quelques points d'attention et techniques non exhaustives sont listés pour information.

Le moyen des jusqu'à 52 lettres du compilateur édité par HP peut aisément être remplacée par la technique « des constantes », ou par celle des « variables » empilables de Burroughs (Unisys). L'interprétation par le compilateur HP de chacune des lettres est binaire : '*elle a été reçue comme argument de compilation*' ou '*elle n'a pas été reçue comme argument de compilation*'. Chaque lettre λ utilisée dans la compilation conditionnelle de HP peut être remplacée par un $IF sur une constante ou une variable, appelons là **`CoCo-`**λ , passée au compilateur ; une ligne ou une séquence de lignes successives dépendante de la présence de cette lettre λ en colonne 7 aura simplement à être soumise à la condition de compilation **`$IF CoCo-`**λ **`defined ... $END`** pour les constantes, ou alors à **`$IF CoCo-`**λ **`... $END`** pour les variables booléennes empilées (TRUE si reçue en paramètre, FALSE sinon) d'Unisys. Le texte source perdra un peu en lisibilité, mais le résultat compilé serait identique. (Le compilateur Micro Focus Cobol pour Visual studio n'assume pas le dialecte de HP pour interpréter les lettres en colonne 7 et mettre le scénario en évidence.)

From_HP.cbl

Fichiers divers

```
       identification division.
       program-id. From-HP.

       environment division.
       configuration section.

       data division.
       working-storage section.
       01 DIVERSES-VARIABLE VALUE SPACES.
          05 MAIL--DESTINATAIRE OCCURS 5 PIC X(50).
          05 MAIL--EN-COPIE OCCURS 5 PIC X(50).
          05 MAIL--EN-COPIE-CACHEE OCCURS 5 PIC X(50).
       procedure division.
      k    MOVE "HD60@UNAMUR.BE" TO MAIL--EN-COPIE(1)
      r    MOVE "JOEL@UNAMUR.BE" TO MAIL--DESTINATAIRE(1)
      r    MOVE "XAVIER@UNAMUR.BE" TO MAIL--DESTINATAIRE(2)
      r    MOVE "YVES@UNAMUR.BE" TO MAIL--DESTINATAIRE(3)
      k    DISPLAY "Pas de destinataire spécifique"

           goback.

       end program From-HP.
```

Fig. 16 De la version 52 lettres de HP …

To_Cst.cbl

Fichiers divers | TO-CST | Procedure Divi

```
       identification division.
       program-id. To-Cst.

       environment division.
       configuration section.

       data division.
       working-storage section.
       01 DIVERSES-VARIABLE.
          05 MAIL--DESTINATAIRE OCCURS 5 PIC X(50).
          05 MAIL--EN-COPIE OCCURS 5 PIC X(50).
          05 MAIL--EN-COPIE-cachee OCCURS 5 PIC X(50).
       procedure division.
      $IF CoCo-k defined
           MOVE "HD60@UNAMUR.BE" TO MAIL--EN-COPIE(1)
      $END
      $IF CoCo-r defined
           MOVE "JOEL@UNAMUR.BE" TO MAIL--DESTINATAIRE(1)
           MOVE "XAVIER@UNAMUR.BE" TO MAIL--DESTINATAIRE(2)
           MOVE "YVES@UNAMUR.BE" TO MAIL--DESTINATAIRE(3)
      $END
      $IF CoCo-k defined
           DISPLAY "Pas de destinataire spécifique"
      $END

           goback.

       end program To-Cst.
```

Fig. 17 …à la version avec constantes

La directive CONSTANT CoCo-k "1" est passée, l'IDE met le scénario en évidence.
(on aurait également pu passer la CONSTANTE CoCo-r "1" si nécessaire)

Dans l'autre sens, lorsque ce sera possible, cela s'annoncera rapidement plus ardu, par exemple il faudra :

+ qu'il y ait moins de 52 constantes distinctes … ou même moins encore
  - s'il y a des arrangements de **`$IF constante`** imbriqués,
  - ou s'il y plusieurs valeurs distinctes testées par constante,
  - ou un mélange des deux complications ci-dessus…
+ qu'il y ait moins de 52 variable distinctes, ou même moins encore
  - s'il y a des conditions avec opérateur(s) booléen(s), …

La directive CONSTANT CoCo-User "JLD" est passée, l'IDE met le scénario en évidence

From_Cst.cbl

Fichiers divers

```
       identification division.
       program-id. From-Cst.

       environment division.
       configuration section.

       data division.
       working-storage section.
       01 DIVERSES-VARIABLE.
          05 MAIL--DESTINATAIRE PIC X(50).
          05 MAIL--EN-COPIE PIC X(50).
          05 MAIL--EN-COPIE-cachee PIC X(50).
       procedure division.
      $IF CoCo-User defined
           MOVE "HD60@UNAMUR.BE" TO MAIL--EN-COPIE
      $IF CoCo-User = "JBA"
           MOVE "JOEL@UNAMUR.BE" TO MAIL--DESTINATAIRE
      $END
      $IF CoCo-User = "JLD"
           MOVE "JEAN-LOUIS@UNAMUR.BE" TO MAIL--DESTINATAIRE
      $END
      $IF CoCo-User = "CDU"
           MOVE "CATHY@UNAMUR.BE" TO MAIL--DESTINATAIRE
      $END
      $END
           goback.

       end program From-Cst.
```

Fig. 18 De la version avec constantes…

To_HP.cbl

Fichiers divers | TO-HP | Working

```
       identification division.
       program-id. To-HP.

       environment division.
       configuration section.

       data division.
       working-storage section.
       01  DIVERSES-VARIABLE.
           05 MAIL--EN-COPIE PIC X(50).
           05 MAIL--DESTINATAIRE PIC X(50).
       procedure division.
      p    MOVE "HD60@UNAMUR.BE"       TO MAIL--EN-COPIE
      q    MOVE "JOEL@UNAMUR.BE"       TO MAIL--DESTINATAIRE
      r    MOVE "JEAN-LOUIS@UNAMUR.BE" TO MAIL--DESTINATAIRE
      s    MOVE "CATHY@UNAMUR.BE"      TO MAIL--DESTINATAIRE
           goback.

       end program To-HP.
```

Fig. 19 …à la version 52 lettres de HP

La seule constante de compilation CoCo-User utilisée ci-dessus va, pour ce simple exemple, être à l'origine de l'usage de 4 lettres dans la version du compilateur HP référencé.
Si la constante CoCo-User est déclarée quelle que soit sa valeur, la lettre p doit être passée au compilateur HP.
Si la constante CoCo-User est déclarée valant « JBA » (resp. « JLD ») (resp. « CDU »), il faudra passer aussi la lettre q (resp. r) (resp. s) au compilateur HP.

(Le compilateur Micro Focus Cobol pour Visual studio n'assume pas le dialecte de HP pour interpréter les lettres en colonne 7 et mettre le scénario en évidence.)

La conversion de la technique des constantes vers celles des variables aux valeurs empilées/dépilées est théoriquement pleinement réalisable : chaque paire (nom-de-constante, valeur) peut être substitué par une variable appelée **nom-de-constante-valeur** alors que, soumis à l'évaluation positive de **nom-de-constante-valeur**, on définira dans la **DATA DIVISION** COBOL une zone **01 nom-de-constante CONSTANT valeur.** C'est effectivement pleinement réalisable, mais potentiellement créateur d'une énorme dette technique si les valeurs de variables évoluent, et particulièrement plus si, au lieu de «univoquement changeante » (p.ex. il ne peut y avoir, par définition, qu'une seule DERNIERE-MISE-EN-PROD-DT), il y a augmentation du nombre de valeurs alternatives possibles qu'une constante peut se voir attribuer au lancement de la compilation (p.ex. COCO-USER ← « CDU »/« JLD»/« JBA »[/nouvel utilisateur[..]]…).

Le point critique d'une telle migration se situera plutôt au niveau de la localisation des directives puisque, rappelons-le : les directives localisées entre $IF condition …. $END-IF ne sont pas exécutées lorsque la condition n'est pas vérifiée, alors que les directives du compilateur Unisys, même entre **$SET OMIT** = not condition … **$POP OMIT** sont toujours exécutées. Comme explicité dans le point listant les inconvénients de la méthode (p.42),.il y a moyen de solutionner le problème en ne basculant pas vers les **$SET OMIT** spécifique à Unisys mais en ayant recours à l'usage des instructions $IF condition-complexe-sur-variable(s)-empilée(s) que l'éditeur a introduit à son compilateur Cobol 85.

From_Cst2.cbl

```
identification division.
program-id. From-Cst.

environment division.
configuration section.

data division.
working-storage section.
01 DIVERSES-VARIABLE.
  05 MAIL--DESTINATAIRE PIC X(50).
  05 MAIL--EN-COPIE PIC X(50).
  05 MAIL--EN-COPIE-cachee PIC X(50).
procedure division.
$IF CoCo-User defined
     MOVE "HD60@UNAMUR.BE" TO MAIL--EN-COPIE
     STRING CoCo-User "@UNAMUR.BE" DELIMITED BY SIZE
       INTO MAIL--DESTINATAIRE .
$END
     goback.

end program From-Cst.
```

Aucun problème détecté

To_Omit.cbl

```
identification division.
$SET CoCo--User = CoCo--User-JBA or CoCo--User-JLD
$SET Tempo      = CoCo--User     or CoCo--User-CDU
$POP CoCo--User
$SET CoCo--User = Tempo
$POP Tempo
 program-id. From-Cst2.
 environment division.
 configuration section.
 data division.
 working-storage section.
 01  DIVERSES-VARIABLE.
     05 MAIL--DESTINATAIRE PIC X(50).
     05 MAIL--EN-COPIE PIC X(50).
     05 MAIL--EN-COPIE-cachee PIC X(50).
$SET OMIT = NOT CoCo--User
 01  COCO-USER CONSTANT
$SET OMIT = NOT CoCo--User-JBA
             "JBA".
$POP OMIT
$SET OMIT = NOT CoCo--User-JLD
             "JLD".
$POP OMIT
$SET OMIT = NOT CoCo--User-CDU
             "CDU".
$POP OMIT
*l'usage de plus d'un CoCo--User-... connu donnera compile error !
*l'usage de CoCo--User-... inconnus, les ignorera totalement
$POP OMIT
 procedure division.
$SET OMIT = NOT CoCo--User
     MOVE "HD60@UNAMUR.BE" TO MAIL--EN-COPIE
     STRING COCO-USER "@UNAMUR.BE" DELIMITED BY SIZE
       INTO MAIL--DESTINATAIRE
$POP OMIT
     goback.
```

Aucun problème détecté

Fig. 20 De la version avec constantes, à la version Unisys $SET OMIT = condition

La conversion réciproque : du compilateur Unisys faisant usage des variables aux valeurs booléennes empilées/dépilées vers un compilateur n'utilisant que les constantes dans ses instructions de compilation conditionnelle pourra rapidement s'avérer plus complexe, plus fastidieux, voire probablement impossible face à des cas de figures.

Les variables empilées de la technique de départ (Unisys) sont des variables de seul type booléen ; et l'usage dans une expression d'une variable non déclarée équivaut à considérer qu'elle vaut false ; c'est donc assez simple. A l'arrivée, la technique utilisant les constantes dispose aussi de la possibilité

d'utiliser des constantes dont on limitera les valeurs à '0' et '1' par exemple, mais cette technique distingue l'inexistence de la définition d'une constante de la définition d'une constante avec la valeur false, ou true. Bref, pour une constante, cela fait trois situations possibles. Pour revenir à la situation de seules deux valeurs possibles, la tentation est grande de, dès le début du texte source, évaluer l'existence, ou non, d'une constante de compilation et de la créer avec valeur false en cas de non-existence détectée. Cette solution s'avère satisfaisante lorsqu'il s'agit de l'usage d'une définition passée, ou non, exclusivement en ligne de commande au lancement de la compilation telle que le serait la précision « comme on compile pour l'environnement de développement (resp. production), on doit définir **FORDEV** (resp. **FORPROD**) » … Cette approche de simplification ne peut hélas pas s'appliquer dès qu'il s'agit d'une constante don la valeur sera éventuellement calculée au gré de l'incorporation, ou non, au texte du programme, d'un COPY en déterminant la valeur.

Des solutions de contournement peuvent exister, mais ce ne sera pas toujours possible sans une refonte profonde de la manière d'écrire les différents scénarii dans un seul texte source.

Ci-dessous, basé sur le besoin spécifique aux divers programmeurs d'obtenir la trace, ou non, et certainement jamais en production, de ce qui se fait dans quelques COPY's l'exemple d'un code dans sa version initiale Unisys (appliquant les aspects théoriquement décrits p.46), et deux exemples de conversions.

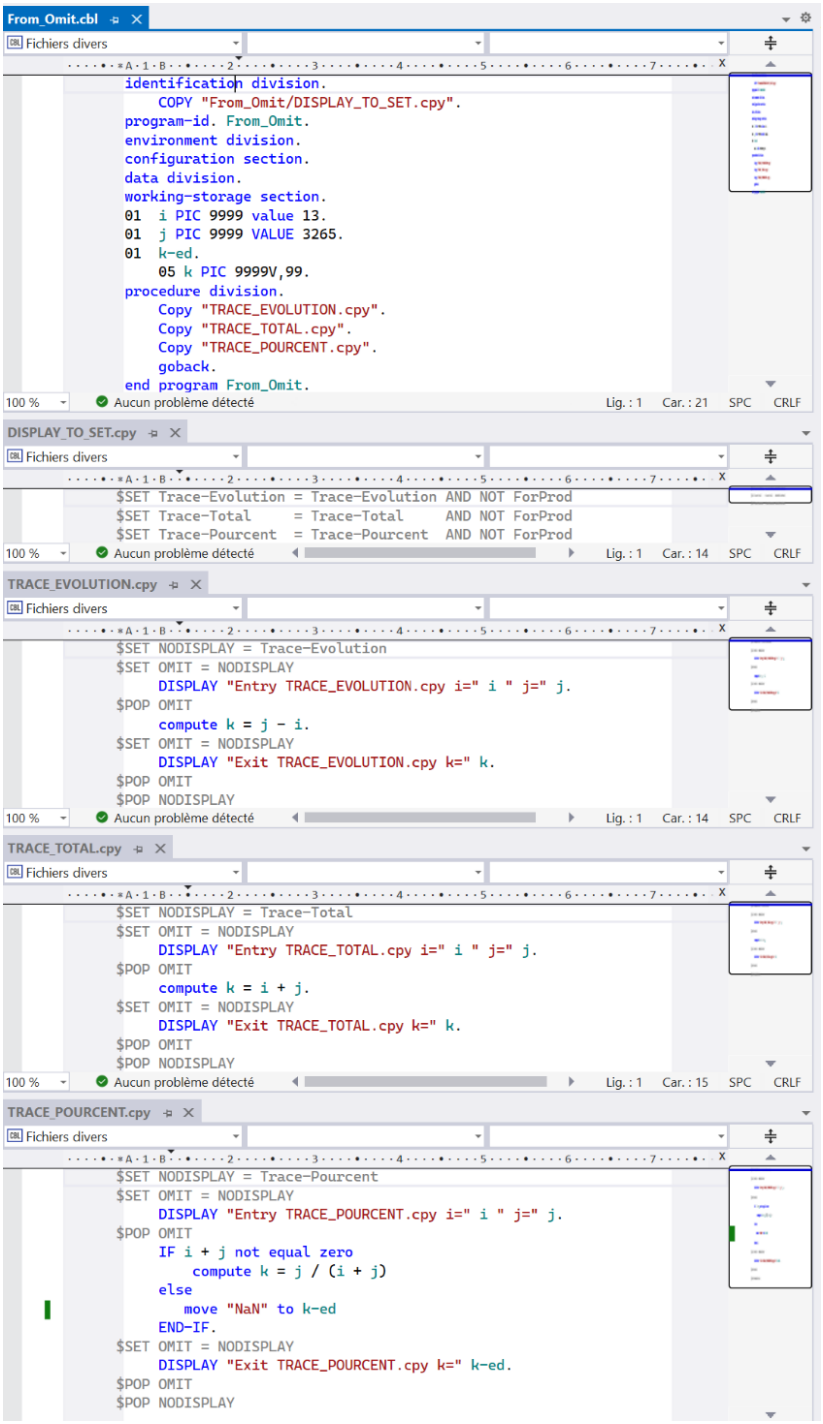


Fig. 21 De la version Unisys $SET OMIT = condition …

Première version de destination : l'usage des constantes ne permet pas de « recalculer » au gré des besoins les demandes de **`Trace-...`** et de les bloquer en production, ni de recalculer **`NODISPLAY`**, or on souhaite maintenir la mobilité du code (déplacer du code d'un COPY à l'autre, diviser un COPY). La solution peut être obtenue en ressortant des COPY's les calculs d'affectation des constantes et d'utiliser ces calculs pour déterminer où, parmi plusieurs choix possibles, inclure un COPY. La définition de la constante **`NoDisplay`** est placée dans la partie « main », ensuite on établit les conditions pour inclure les COPY's AVANT cette définition si les displays sont autorisés (on n'est pas en production) et souhaités pour le COPY, ou après la définition de la constante si les **`Trace-...`** ne sont pas autorisés (production) ou pas souhaités pour le COPY. On constate vite le limitation de la méthode puisque dès

la présence de deux constantes (semblables à **`NoDisplay`**) dont les 4 combinaisons 0-0|0-1|1-0|1-1 existent, il n'est pas possible de procéder à ce type de ventilation étant donné l'impossibilité d'annuler une définition pour passer de 0-1 à 1-0. On constate que la compréhension des scénarii est de prime abord moins évident, malgré leurs simplicités

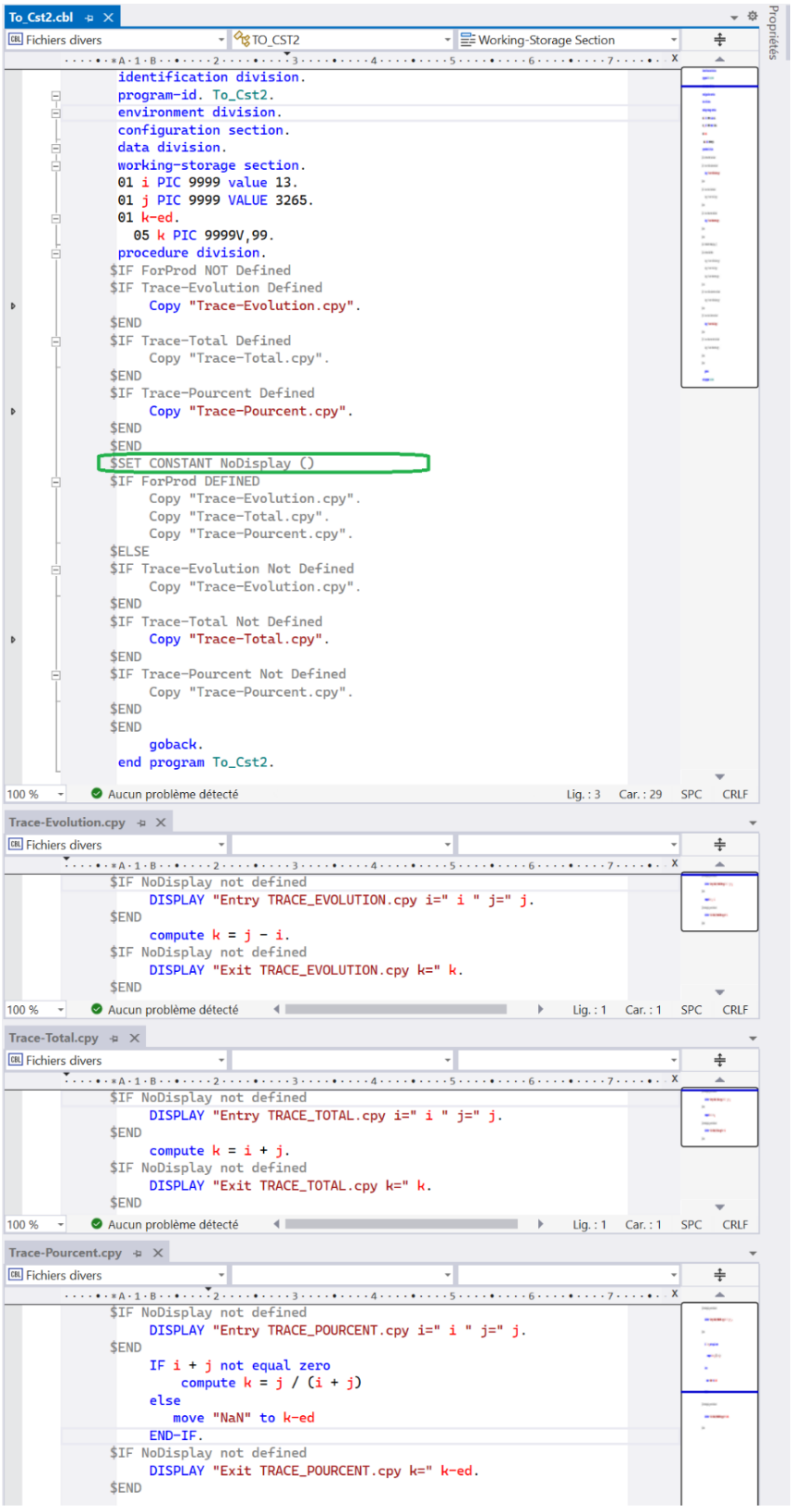


Fig. 22 … à la version avec constantes V1

La deuxième version de destination préservera plus facilement la lecture et compréhension des scénarii au détriment de la mobilité du code : on supprime le bénéfice qu'on avait obtenu grâce à la variable **`NoDisplay`** pour faire usage d'une constante par COPY. Il y a plus de travail en cas de mobilité du code, de scission ou de fusion de COPY's, mais c'est naturellement plus compréhensible.

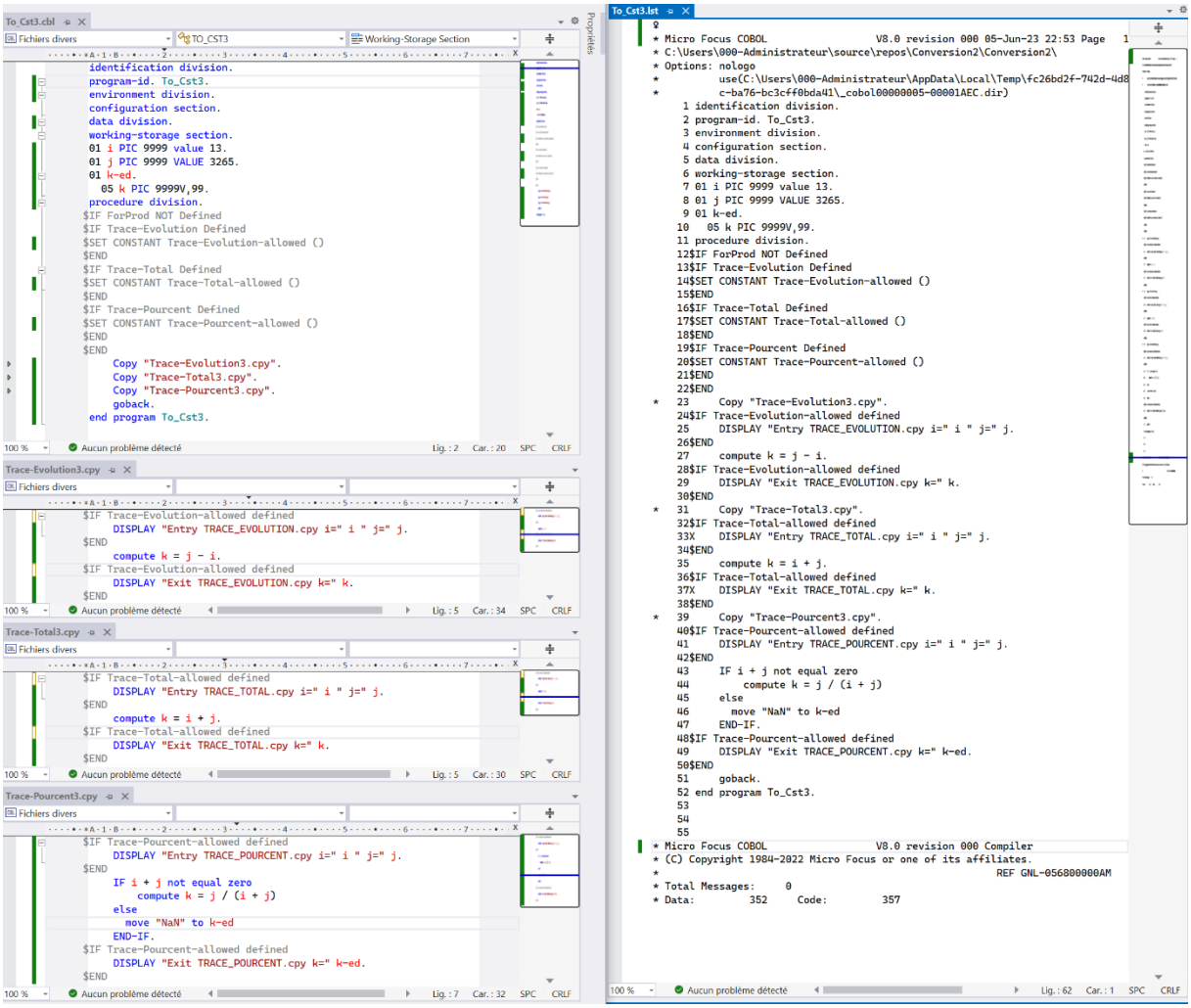


Fig. 23 … à la version avec constantes V2

En tout état de cause, suivant l'usage de la compilation conditionnelle présente dans le texte source de départ, si elle est possible la conversion suivant de simples recettes pourra produire un texte source qui deviendra vite « illisible », « incompréhensible » et probablement très lourd à maintenir.

5.4 Avec les possibilités de compilations conditionnelles de la norme 2002, que peut-on atteindre ?

L'usage proposé après étude de la compilation conditionnelle en COBOL n'a pas pour but d'envisager des programmes destinés à alimenter diverses architectures, mais propose des moyens pour se ***construire*** et enfin ***obtenir*** des facilités dans la programmation du business, quelle que soit la plateforme sur laquelle une équipe travaille en collaboration (standardisation, factorisation de code, disponibilités « à la carte », émulation OO).

5.4.1 L'usage courant du COBOL

Jusqu'à présent, pour éviter ces répétitions multiples et la dette technique, on créait des petits fichiers sources (3 ci-dessous, surlignés de jaune), chacun avec son groupe d'informations « stables »

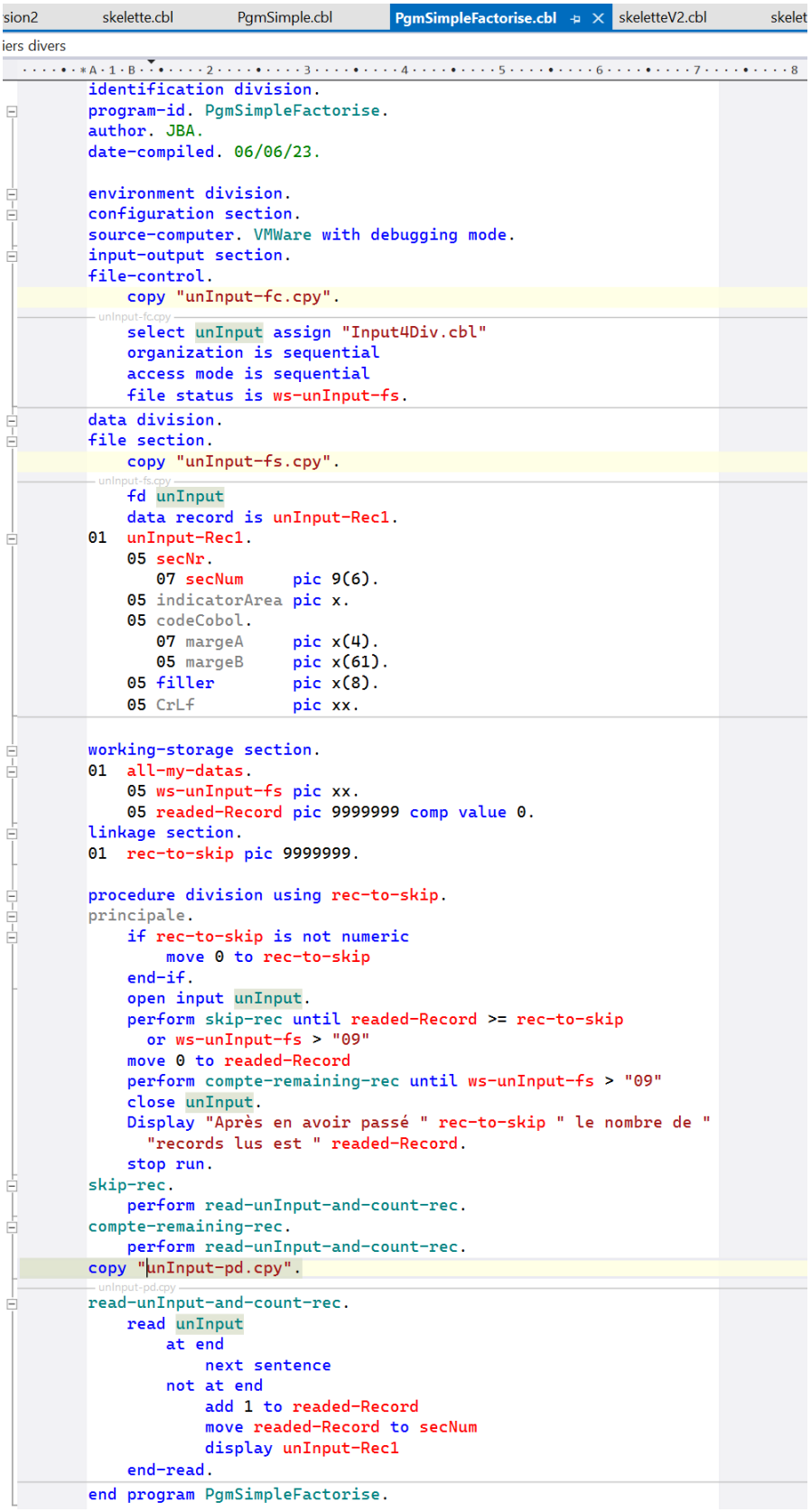


Fig. 24 L'actuel usage courant du COBOL et des multiples COPY's

incorporés dans les différents programmes grâce à la directive COPY. C'est simplificateur et efficace, mais pas sans risque. Suite à certaines évolutions, on pourrait arriver à des discordances entre les contenus de ces différents COPY's qui pourtant concernent, à la base, le même fichier … mais plus la même version du même fichier. On pourrait aussi, par une malheureuse permutation, se tromper dans la localisation de l'endroit où on sollicite un COPY. Parce que lui sait exactement où il en est dans le texte source et qu'il analyse la syntaxe qu'il admet à cette endroit, le compilateur COBOL est très rigoureux … mais sans donner de manière native la possibilité de s'enquérir de cet endroit où l'on est.

5.4.2 Un squelette pour la compilation conditionnelle

Les variables de compilations conditionnelles vont donc être utiles pour donner la possibilité au programmeur de prévoir du code si et seulement si il est à un endroit précis, et ainsi mieux gérer ses COPY's afin d'obtenir durant la phase de compilation des contrôles de pertinence.

Le squelette tel que montré dans la Fig. 25 ne permet d'identifier que la division où l'on est, celui de la Fig. 26 va plus loin et identifie les sections (Ce n'est pas exhaustif, une équipe de cobolistes a à constituer son squelette sur base de son usage de COBOL)

Toutes le directives initialisant les variables de compilation utilisées par le squelette sont isolées dans le COPY **`DEFINE4skelette.cpy`** et ainsi facilement reprises dans tous les programmes.

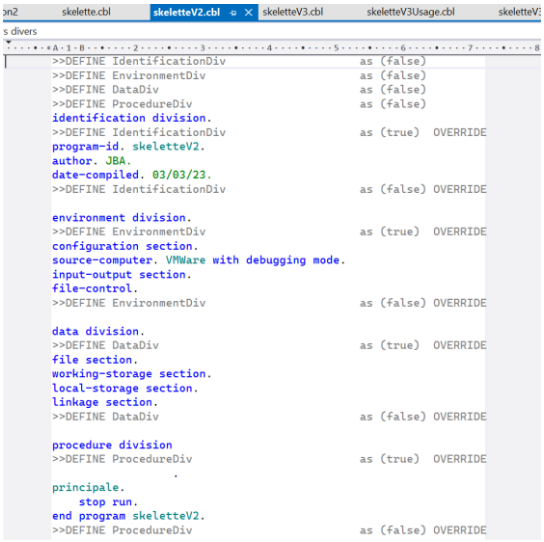


Fig. 25 Un squelette préparé pour l'exploitation de la compilation conditionnelle V1

Fig. 26 Un squelette préparé pour l'exploitation de la compilation conditionnelle V2

5.4.3 D'un usage simple jusqu'au choix de fonctionnalités « à la carte »

Ce squelette étant réalisé et donc utilisable, il n'est maintenant plus nécessaire d'avoir plusieurs COPY pour les différentes composantes que l'on souhaite intégrer : un seul (Fig. 27), plus consistant, dont chaque partie du contenu incorporera le texte du programme en fonction de la localisation. Ce COPY peut reprendre plus de fonctionnalités (ouverture en lecture ou en écriture, instructions de lecture ou d'écriture) que nécessaire, du moment que sont au moins présentes celles requises. Des fonctionnalités spécifiques au programme pour chaque enregistrement lu sont facilement intégrables dans un paragraphe spécifiquement nommé et prévu à cet effet.

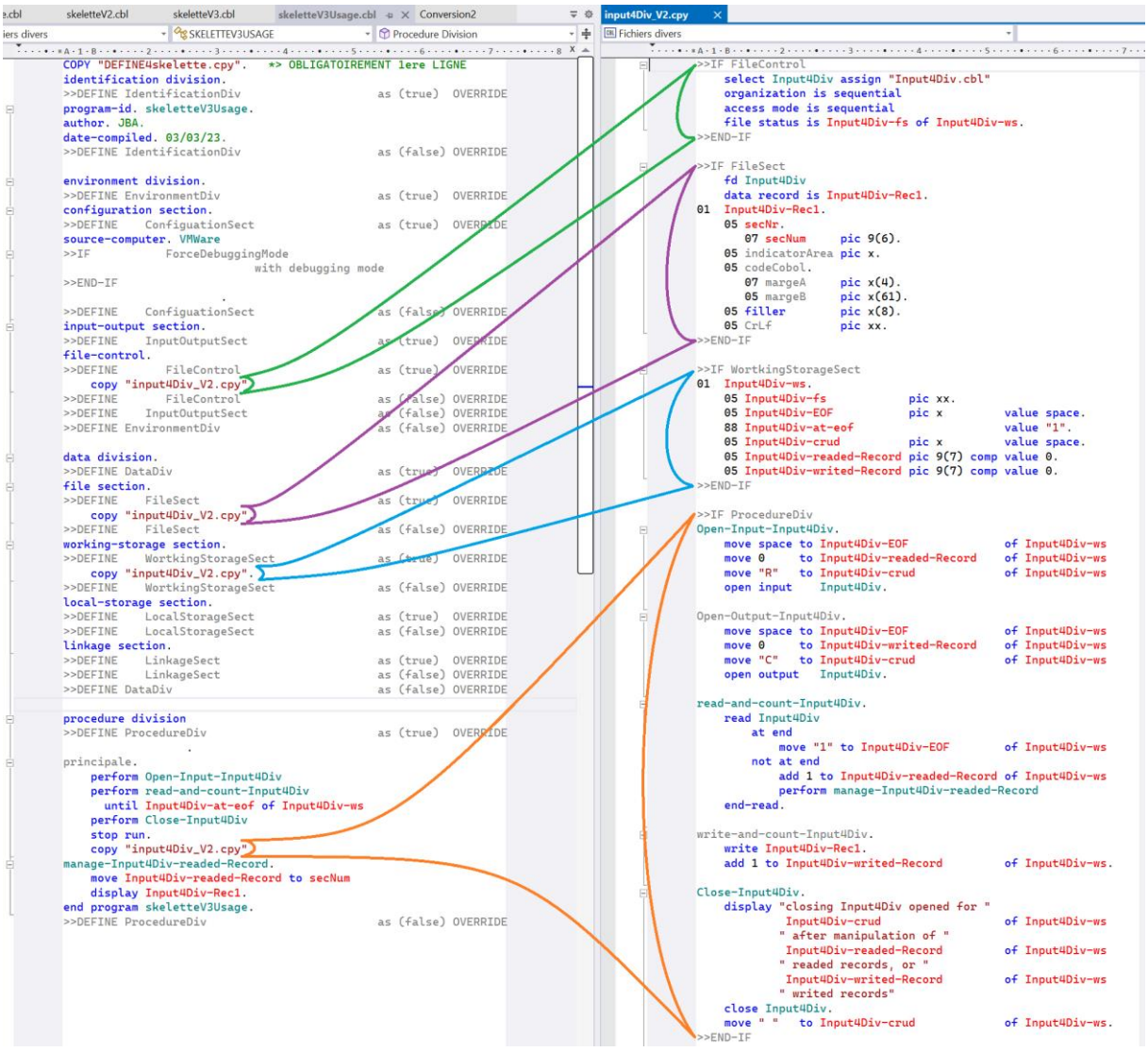


Fig. 27 Tout ce qu'il faut pour exploiter un fichier, regroupé dans un copy, préparé à être correctement réparti

Cela étant, s'il devait exister différentes versions du COPY, il est toujours possible d'obtenir le genre d'anomalie de la *Fig. 28* où on se tromperait malgré tout : on incorpore tantôt une version du COPY, tantôt une autre (…**_V2**, …**_V3**) qui sont peut-être syntaxiquement compatibles, mais avec éventuellement des processus bien différents, ou une variation de longueur ou un changement de structure du data record (ici de **Input4Div-Rec1**) … Ce genre de confusion, non nécessairement perceptible de manière flagrante, peut mener à la génération de résultats incorrects.

C'est évidemment à inclure au code par le programmeur, mais la compilation conditionnelle permet, ***grâce à des contrôles en cascade, de veiller à la cohérence,… sans impacter l'exécutable produit.***

La colonne de droite de la Fig. 29 montre un tel processus de contrôle en actant, lors d'une insertion

effectuée, l'action dans une variable de compilation unique ; l'existence de cette dernière sera impérative pour procéder à l'insertion de la partie suivante. Grâce à cette technique, on constate que pour la *Fig. 28* l'incorporation de **`input4Div_V3.cpy`** dans la **`working-storage`** ait été signalée

```
>>DEFINE    InputOutputSect                 as (true)  OVERRIDE
file-control.
>>DEFINE       FileControl                  as (true)  OVERRIDE
    copy "input4Div_V2.cpy".
>>DEFINE       FileControl                  as (false) OVERRIDE
>>DEFINE    InputOutputSect                 as (false) OVERRIDE
>>DEFINE EnvironmentDiv                     as (false) OVERRIDE

data division.
>>DEFINE DataDiv                            as (true)  OVERRIDE
file section.
>>DEFINE    FileSect                        as (true)  OVERRIDE
    copy "input4Div_V2.cpy".
>>DEFINE    FileSect                        as (false) OVERRIDE
working-storage section.
>>DEFINE    WortkingStorageSect             as (true)  OVERRIDE
    copy "input4Div_V3.cpy".
>>DEFINE    WortkingStorageSect             as (false) OVERRIDE
local-storage section.
>>DEFINE    LocalStorageSect                as (true)  OVERRIDE
```

Fig. 28 Risque d'erreur (p.ex. confusion) dans l'invocation des COPY's

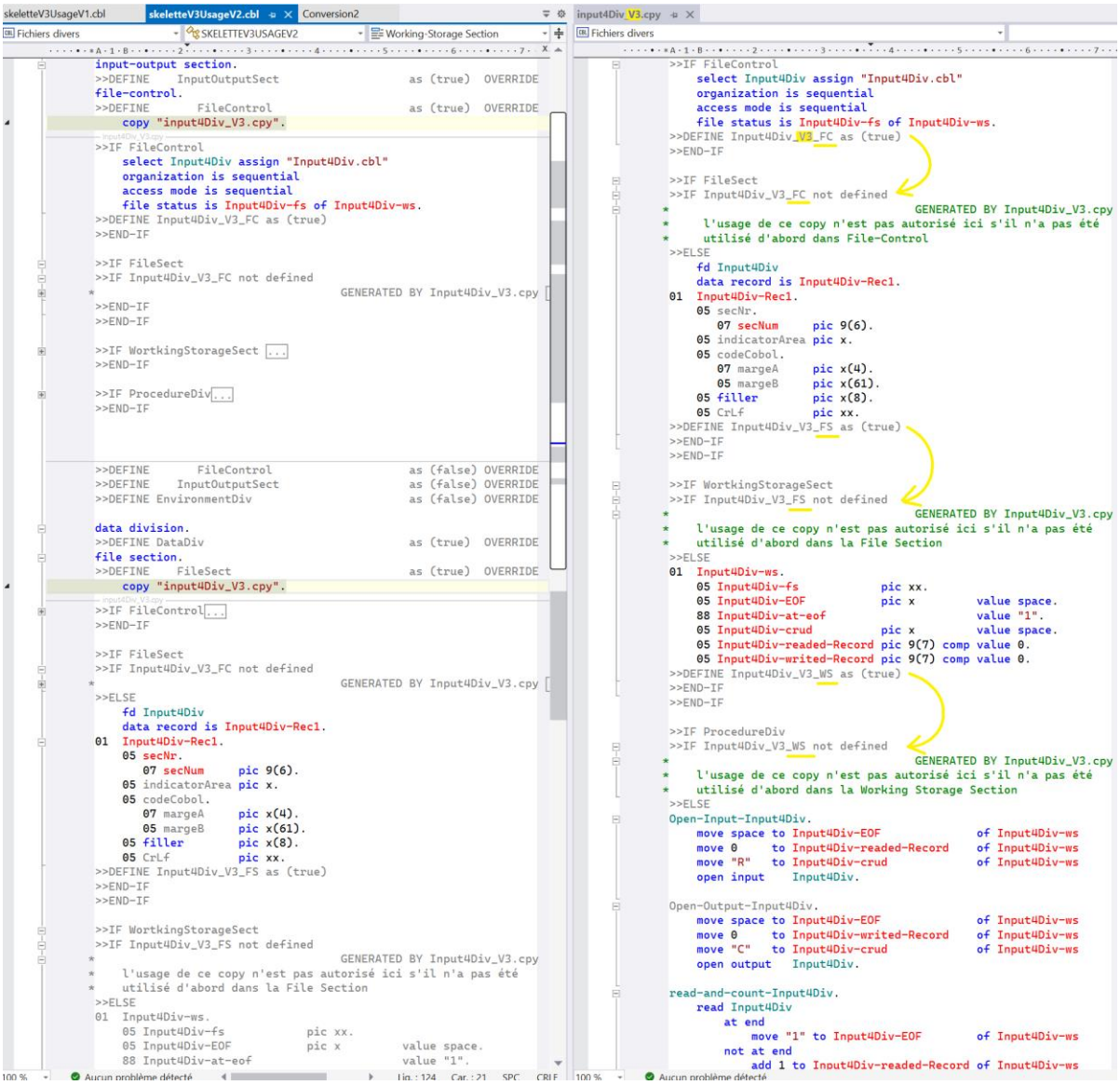


Fig. 29 Intégration dans les COPY's d'un mécanisme pour prévenir le risque de confusion lors de leurs invocations

comme impropre puisque ce COPY est absent dans la **`File Section.`** Cela est détecté et révélé par l'inexistence de **`input4Div_V3_FS`**.

Allons plus loin encore : pourquoi ne se débarrasserait-on pas de la contrainte d'aller dans les programmes changer des noms de COPY à plusieurs places ? Pourquoi ne pas créer par équipe de développement ou par domaine, un COPY englobant *<les COPY's de tous les fichiers>*, ***voire même*** *<les COPY's de toutes les versions de tous les fichiers>* (il est en effet possible que tous les programmes n'évoluent pas à la même vitesse) ? Ensuite il ne resterait plus qu'à choisir « à la carte » la version de chacun des fichiers que le programme doit utiliser. La Fig. 30 ci-dessous affiche l'expansion du COPY englobant (arbitrairement appelé **`AllAvailableFiles.cpy`**) et à l'intérieur même, l'expansion de deux COPY's imbriqués ayant été sélectionnés en définissant les deux variables **`use_...`** requises. L'incorporation des autres COPY's est ici ignorée.

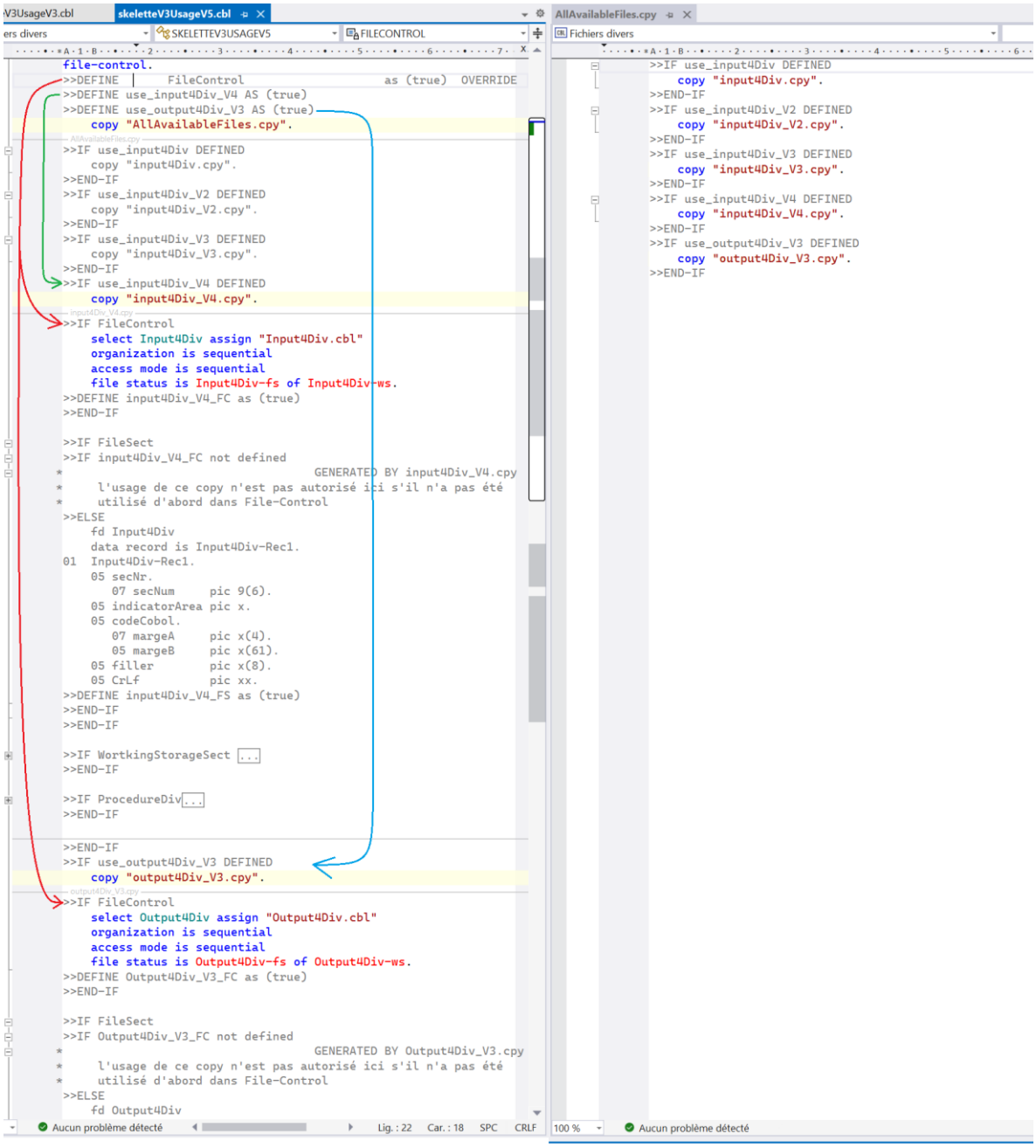


Fig. 30 En façade un COPY contenant tous les COPY's, ceux voulus "activés" par variables : cohérence assurée

Lorsqu'il est à l'aise avec la compilation conditionnelle, le développeur COBOL pourra aussi prévoir de choisir le fichier, sa version, … *et de spécifier si c'est pour de l'input, de l'output, de l'input-output, du séquentiel, de l'indexé,…* évitant ainsi d'inclure des paragraphes qui seront du code mort. Dans les codes ci-dessous où on se limite à deux choix, on opte pour suffixer par **`-4IN`** si on prévoit de lire le fichier ou **`-4OUT`** si on souhaite générer le fichier. Si au moins un des deux choix est exprimé (**`DEFINED`**) pour un fichier, **`AllAvailableFiles.cpy`** doit prendre en compte le COPY de ce fichier pour qu'il apparaisse dans la **`File-Control Section`**, dans la **`File Section`**, dans la **`Working-Storage Section`** et dans la **`Procedure Division`**. Le COPY du fichier lui-même ne devra être adapté que pour limiter les paragraphes d'instructions exécutables à faire compiler, c'est-à-dire dans la **`Procedure Division`**. Une telle possibilité est montrée dans **`theFile4Div.cpy`** de la Fig. 31 ci-dessous, colonne de gauche.
Tout comme cela avait été montré précédemment pour le choix des « **`trace-`**… » (Fig. 15) le développeur pourra aussi décider d'inclure à chaque COPY le code nécessaire pour obtenir ***à la demande*** la possibilité de tracer, à l'écran ou dans un fichier, les actions réalisées dans le COPY, qu'il estimerait important de « suivre ». Dans le COPY **`theFile4Div.cpy`** donné en exemple dans la colonne gauche de la Fig. 31, on trouve que la variable de compilation conditionnelle à activer pour ce fichier est **`trace_theFile4Div.`** Tel que conseillé en Fig. 15, et expliqué (*E*, *H*$_1$ et *H*$_2$ page 51) on préconise de toujours « passer la main » à la même variable, p.ex. **`DISPLAYIT`** définie localement, pour, dans tous les COPY's, contrôler l'inclusion, ou non, des messages (de base) voulus.
La définition d'un COPY de fichier supplémentaire, **`File4Trace.cpy`** dans le cas présent (colonne de droite de la Fig. 31), va contenir ce qui est nécessaires pour offrir un maximum de facilités à tous les COPY's où le programmeur souhaite obtenir la génération d'un logging : des zones de conversion pour des éditions et un log plus lisible, zone d'assemblage du message, pointeur disponible pour la clause **`pointer`** d'une instruction **`string`**, et avec <*en option pour cette facilité*> l'incorporation des codes nécessaires si on souhaite le log dans un fichier en plus d'un affichage à l'écran (on pourrait rendre exclusif le choix de l'écran ou du fichier). Ce qui est présenté dans l'exemple n'est évidemment pas exhaustif : on pourrait souhaiter l'édition d'un timestamp, on pourrait souhaiter le préfixage de chaque message par le numéro du thread/du pid/du mix/du job (selon le système) surtout si des runs parallèles ont lieu, le choix dynamique du nom du fichier de log,… mais tout cela relève du COBOL et n'est pas spécifique à la compilation conditionnelle.

La Fig. 32 montre le peut qui a changé dans le programme principal :
- le **`use_aFile4Div`** est remplacé par **`use_aFile4Div-4OUT`** puisqu'on veut générer ce fichier ;
- le **`use_theFile4Div`** est remplacé par **`use_theFile4Div-4IN`** puisqu'on veut lire ce fichier ;
- un ajoute **`trace_theFile4Div`** pour que soient, *c'est le défaut*, affichés à la console les messages voulus ;
- on a mis en commentaire la variable qui aurait permis que ces messages soient aussi dans un fichier. Le COPY **`File4Trace.cpy`** n'est, dans ce cas, pas explicitement demandé par le programmeur, mais (comme dans **`theFile4Div.cpy)`** par le premier bloc **`>>IF`** de n'importe quel COPY dont on active la demande de **`trace_...`**

Ce qui est montré ici avec des fichiers et leurs accès (et l'ajout de **`trace-...`**) peut tout autant l'être avec les accès aux bases de données : au lieu de **`AllAvailableFiles.cpy`** on aurait **`AllTables-ForDBxyz.cpy`** et on choisirait les tables nécessaires au programme, en précisant pour « lectures seules » ou « pour update » tout comme on choisit « pour lecture » ou « écriture » d'un fichier, un exemple (non commenté) est donné pour le système DB d'Unisys (DMSII). Il en est de même aussi pour les envois ou réceptions de messages sur des queues…

En exploitant la compilation conditionnelle, on bénéfice d'une sélection et activation extérieures aux COPY's proposant des blocs cohérents et complets, ainsi que la possibilité de contrôles effectuables par le compilateur durant son travail.

Fig. 31 Toujours plus de paramétrages dans les COPY's // un copy sollicité par les autres

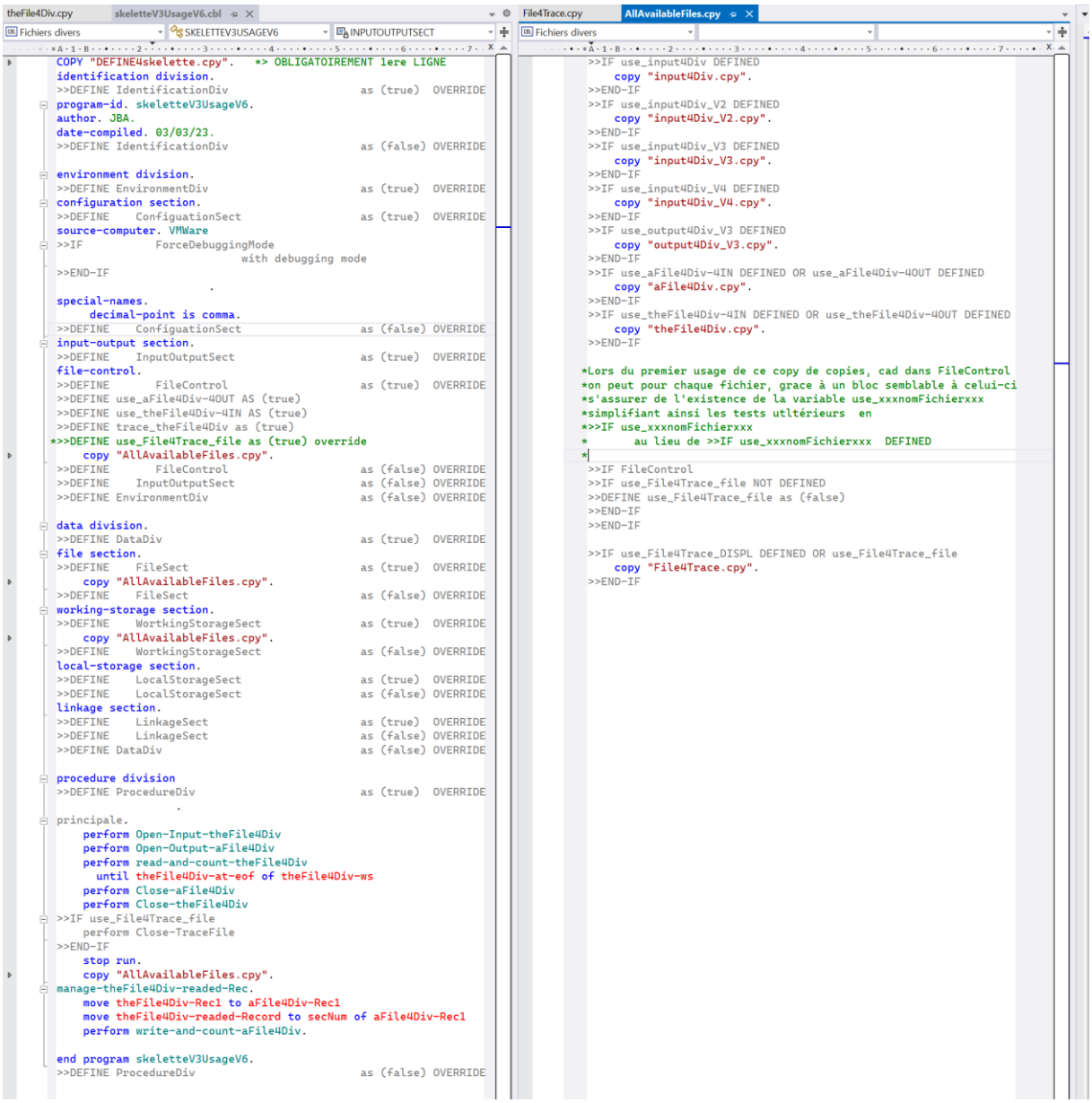


Fig. 32 Satisfaction d'un besoin propagé et satisfait par la compilation conditionnelle

Pour faciliter la perception à la lecture, les directives utilisées dans l'exemple

```
>>DEFINE use_aFile4Div-4OUT as (true)
>>DEFINE use_theFile4Div-4IN as (true)
>>DEFINE trace_theFile4Div as (true)
>>DEFINE use_File4Trace_file as (true) override
```

ont été placées à même le code,… mais *tel que conseillé Fig.* 15 *page 49* il est plus judicieux de préférer placer ces directives dans un COPY spécifique au programme ; le nom évoqué alors était **`nom-du-programme\DISPLAY-TO-SET.cpy`**.

5.4.4 Encore plus loin : l'émulation partielle de l'Orienté Objet

Puisque grâce à la compilation conditionnelle nous disposons de la possibilité de <***regrouper dans un COPY des définitions de zones mémoires et des définitions d'actions*** > et de <***lors de la phase de compilation, savoir répartir où il se doit chaque partie de ce qui a été regroupé***>, peut-on envisager de définir des classes {variables membres} + {méthodes}, regroupées en « package », … pour ensuite créer des objets instanciant l'une ou l'autre de ces classes, et les manipuler ? En 1998, le programmes COBOL étaient déjà considérés « legacy », leur conversion automatisée vers de l'orienté objet était déjà envisagée[58] … et pourtant, nombre de ces programmes « legacy » sont toujours présents.

Il existe du COBOL Orienté Objet mais, *sauf peut-être pour de tout nouveaux programmes,* le but ici n'est pas d'amener des cobolistes à utiliser celui-ci, probablement très complexe à ***mélanger*** aux millions (milliards ?) de lignes existantes de COBOL non OO. L'objectif est de proposer, en combinant la compilation conditionnelle et l'exploitation du code qui existe, une structuration factorisée qui au final présentera une vue semblable : « variables membres + unités de manipulation » regroupées au sein d'un même fichier de code.
Tout le paradigme OO ne sera évidemment pas réalisable, mais on peut malgré tout déjà donner une impression d'OO partiel : <des regroupements 'données'+'processus' par copy>, que comprendra un coboliste ignorant tout de la POO, et un ensemble de <copy regroupant les variables et les codes d'un concept> que comprendra un programmeur habitué à la programmation OO. (L'article de J Joiner et W.-T Tsai [58], même s'ils parlaient d'un processus automatique, contient une méthodologie d'identification des variables d'instance et des méthodes qui pourrait inspirer le travail manuel du programmeur connaissant son application pour arriver à la présente situation intermédiaire basée sur la compilation conditionnelle.)

Contrairement à l'article, l'exemple suivant n'explique pas de méthodes pour extraire de l'OO d'un programme « legacy », ce point est laissé à la connaissance métier du programmeur. Relevons que Krüger et al. [105] mentionnent une conclusion de Jordan et al. [97] après qu'ils eurent mené une observation en suivant deux ingénieurs logiciels expérimentés modernisant un système COBOL : « *leurs résultats suggèrent que la connaissance du domaine améliore l'efficacité et que les outils de recherche ne donnent pas de résultats pertinent.* » Le programmeur pourra malgré tout s'inspirer de techniques qu'il pourra trouver dans la littérature telle la publication de J. Joiner et W.-T Tsai (*ibid.*).

Il n'y a eu pour constituer l'exemple suivant aucun 'reverse engineering'. Le but est de présenter un canevas final d'une implémentation réaliste d'émulation OO vers laquelle on pourrait tendre. Il s'agit de montrer comment constituer un fichier définissant une « classe » pour que la « classe » contienne les variables d'instances, les variables de travailles, les méthodes et comment procéder pour que chaque partie trouve lors de la compilation sa/ses juste(s) place(s) dans le texte du programme.

La motivation est triple :

- Amener les cobolistes à <**mieux découper pour mieux rassembler**> leur code ;
- Présenter des codes plus compréhensibles aux jeunes programmeurs habitués à l'OO, et en convaincre que le COBOL n'est pas trop rébarbatif ;
- Avoir des codes mieux préparés à une migration, automatique ou humaine, vers un nouveau langage exploitant l'OO ; … voire même le COBOL OO pourquoi pas ?

Les captures d'écrans servant à concrétiser les propos sont basés sur le diagramme de classe suivant

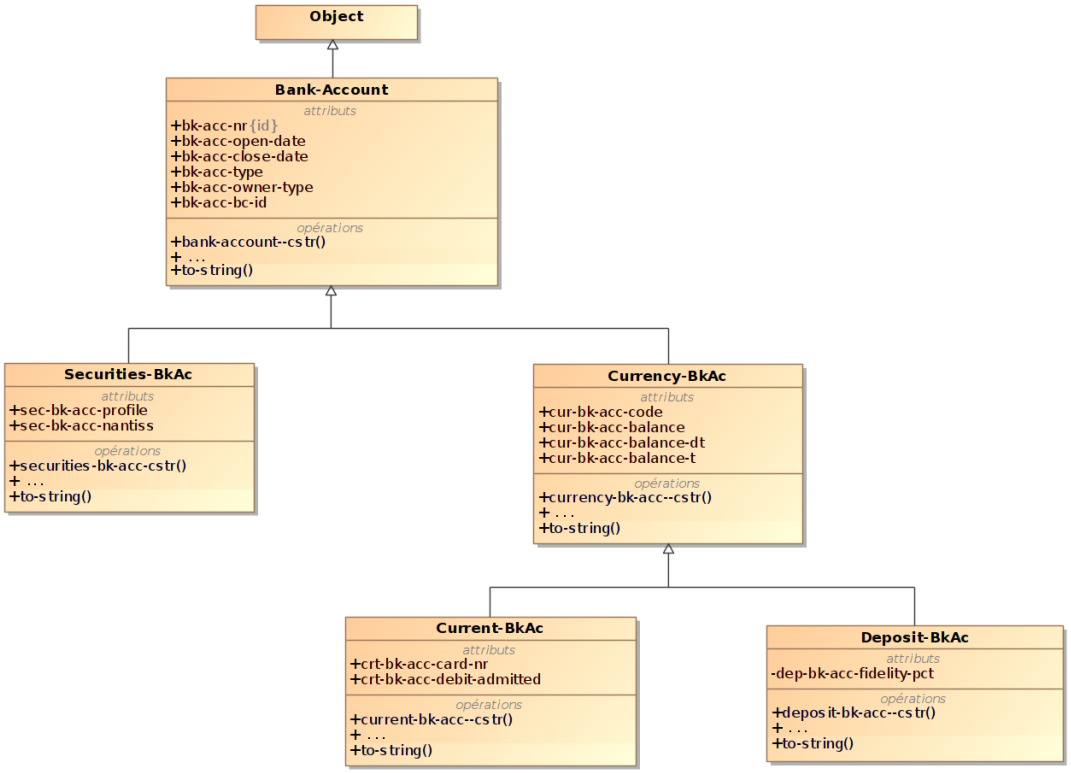


Fig. 33 Diagramme de classes utilisé pour illustrer l'émulation OO

5.4.4.1 La constitution d'une « classe » en utilisant la compilation conditionnelle

La définition d'une classe n'instancie pas un objet mais en défini les variables membres et en défini les méthodes, et comme cela a été montré, grâce à la compilation conditionnelle, un COPY COBOL peut regrouper des déclarations et des instructions.

Classiquement, les variables membres de l'objet sont accessibles via le référent qui porte le nom donné à l'objet. La notion de pointeur existe maintenant en COBOL, mais semble très peu utilisée « telle quelle » : il existe bien des passages et des réceptions de paramètre **`by reference`** mais si, pour un tel paramètre, dans la structure référencée les noms identifiants les zones sont univoques, ils peuvent être utilisés sans préciser le nom du référent. Pour rester dans les habitudes historiques du COBOL, il n'est pas fait usage de « pointeur » dans l'approche présentée, les variables membres seront présentes en mémoire dès où l'objet sera déclaré.

Un COPY va donc contenir la définition des variables membres qui prendront place où on instanciera un objet de cette classe ; comme elles feront partie intégrante de sa structure, elles devront avoir un numéro de niveau supérieur au niveau utilisé pour la déclaration de l'objet. Et ce devra être le cas pour chaque objet instanciant la classe. De plus, pour rester un minimum « encapsulé », et comme ce fut d'ailleurs déjà exploité avec les fichiers (**`Input4Div-ws`** de la Fig. 29, **`theFile4Div-ws`** ou **`TraceFile-ws`** de la Fig. 31) toutes les variables de travail nécessaires à l'exécution des « méthodes » de la classe sont également à définir dans ce COPY . En revanche, ces variables de travail ne seront pas à déclarer pour chaque objet, mais seulement une seule fois, dans la **`Wroking-Storage`** ou **`Local-Storage section`** du programme, et ce pour autant que les méthodes soient nécessaires, ce qui veut dire pour autant qu'un objet au moins instancie la classe dans une des sections '**`File`**', '**`Working-Storage`**', '**`Local-Storage`**' ou '**`Linkage`**' du programme.

L'instanciation d'un objet se fera très simplement en sollicitant, juste sous la déclaration de son identifiant, la copie du fichier définissant la classe qu'on souhaite lui voir attribuer. Ci-dessous un exemple d'instanciation, incluant une structure d'objets et sa redéfinition par un tableau d'objets :

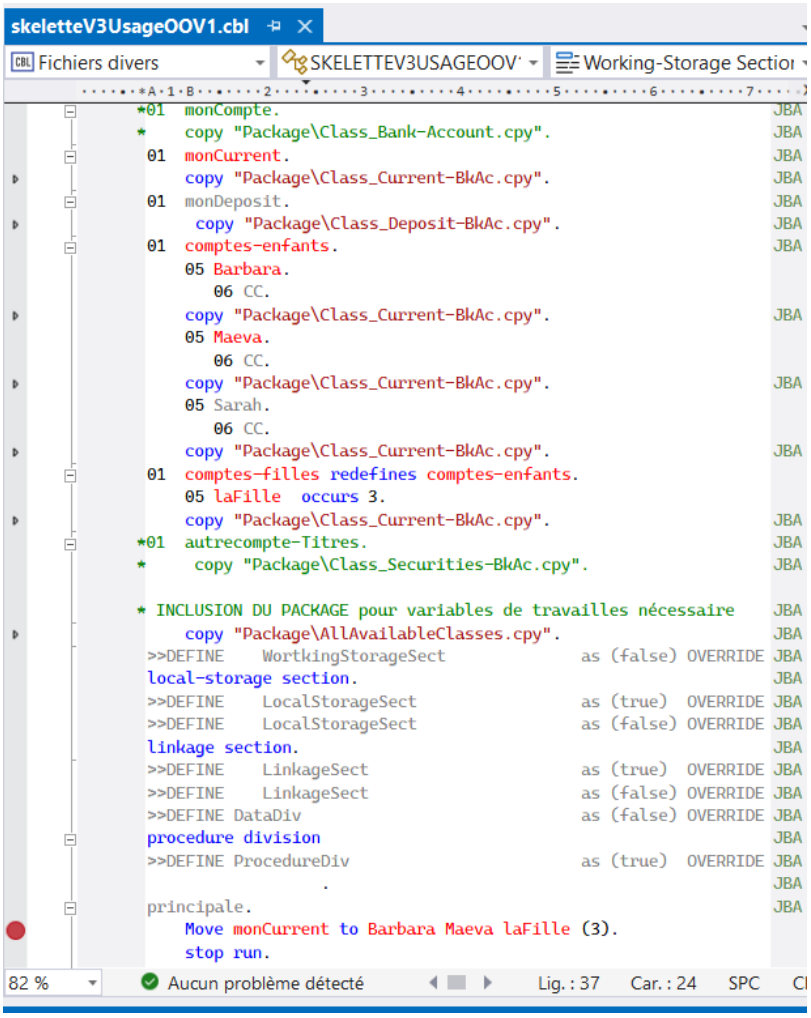


Fig. 34 Simplement invoquer [le copy définissant] la classe pour instancier un objet

Utilisant tous la même définition, chaque objet instanciant une classe sera composé d'une structure dont les noms des variables membres seront identiques ; chaque usage d'une variable membre dans une instruction devra donc être qualifié par le nom de l'objet pour lever toute ambiguïté. Il va sans dire que, même si c'est possible, il est inconcevable de prévoir pour chaque « méthode » (paragraphe(s) COBOL) une version par nom d'objet pour qualifier les variables membres. Pour remédier à cela, on va donc, en fonction des besoins, déclarer un, deux, trois… objets de travail dont les noms serviront à qualifier les variables membres dans les instructions. Il appartiendra au programmeur, avant l'appel de la méthode à exécuter, de copier son ou ses objets dans le ou les objets de travail nécessaires à la méthode, et éventuellement de recopier le ou les objets de travail dans son ou ses objets après l'exécution de la méthode.

Le squelette d'un copy définissant une « classe » doit donc répondre aux possibilités suivantes :

1. N'incorporer au texte du programme, en **`data division`**, que les variables membres (propres **b3** ou héritées **b3-h**) lorsque le copy est invoqué pour déclarer un objet ;
   si au moins une telle déclaration a lieu, ce fait sera mémorisé dans une variable de compilation

qu'on pourrait appeler *AtLeastOne_nomDeClasse* (crée par défaut à 'false' dès le premier usage du copy en a) actée à 'true' en b3

2. Incorporer au texte du programme en **`data division`** les variables b4 de travail, et en suffisance les b1 « objet(s) » b3 de travail regroupés dans une structure b2, si et seulement si au moins un objet a été déclaré comme décrit au point 1, donc
   si et seulement si la variable de compilation *AtLeastOne_nomDeClasse* existe à la valeur 'true'.
3. Incorporer au texte du programme, en **`procedure division`**, les paragraphes (« méthodes ») C, si et seulement si au moins un objet a été déclaré, donc
   si et seulement si la variable de compilation *AtLeastOne_nomDeClasse* existe à la valeur 'true'.

Réaliser le point 3 s'obtiendra simplement en ajoutant le copy en procedure division, il sera d'office inclus après l'éventuelle déclaration d'objet du point 1 ; toute évaluation de l'existence de *AtLeastOne_nomDeClasse* sera significative. Les points 1 et 2 vont demander plus d'attention :

+ il faudra que le point 2 survienne après le point 1 pour que le test sur *AtLeastOne_nomDeClasse* soit significatif. Cela s'obtiendra en plaçant le copy pour réaliser le point 2 le plus possible à la fin de la working-storage section, ou même de la local-storage section si elle existe. Tous les copy's définissant une classe doivent se trouver là à la fin pour satisfaire le point 2. Tous comme a été défini **`AllAvailableFiles.cpy`**, définissons un copy **`AllAvailableClasses.cpy`** et plaçons-le systématiquement le plus bas possible dans une de ces deux sections.
  Il reste malgré tout le problème qui surviendrait lorsque la première déclaration d'un objet d'une classe apparaitrait en Linkage Section, toujours localisée après les working-storage et local-storage sections, dans ce cas il appartiendra au programmeur de lui-même déclarer et positionner la variable AtLeastOne_nomDeClasse. C'est la seule exception détectée.
  Ici, contrairement aux fichiers qu'on choisit dans une liste en activant une variable, le simple usage d'une classe dans la définition d'un objet enclenchera automatiquement tout ce qui est nécessaire.
+ il faut, en son sein, pouvoir distinguer la finalité du copy lorsqu'il sera :
  - utilisé seul, isolé, pour attribuer les variables membres lors d'une déclaration d'objet (point 1)
  - utilisé parmi tous les autres dans **`AllAvailableClasses.cpy`** (point 2). Cela peut se distinguer aisément en positionnant au début de ce copy une variable de compilation précisant ce fait. Le choix étant arbitraire, **`copying_AllAvailablesClasses`** est choisi pour les exemples et visible Fig. 36.

La solution étant trouvée pour distinguer ce qu'on souhaite lors de l'incorporation des copy's, il reste la problématique de création des « objets » de travail lors du point 2 puisque, rappelons-le, les appels récursifs de copy ne sont pas autorisés, et on ne veut pas créer de dette technique due à la répétition du codage des déclaration des variables membres (une fois par objet de travail).

Deux solutions sont présentées ici :

1. Pour cette première technique, le copy définissant la classe, tel qu'envisagé plus haut, contient tout. Elle semble complexe tant qu'on n'en connait pas le squelette ou qu'on est peu habitué à la compilation conditionnelle, mais elle respecte un peu plus l'esprit de la définition de classe puisque tout est maintenu **dans** le copy.
   a. Lorsque **`copying_AllAvailablesClasses`** n'est pas positionné, on sait qu'on définit un objet et que seules les variables membres doivent être générées ; de même on doit positionner *AtLeastOne_nomDeClasse* dont on veille à l'existence, 'false' par défaut
   b. Lorsque **`copying_AllAvailablesClasses`** est positionné, si un objet au moins avait été déclaré, on sait qu'on doit définir les variables de travail, et les objets de travail devant eux-mêmes contenir les variables membres, alors qu'il n'y a pas de récursivité autorisée. Comme la compilation conditionnelle ne prévoit pas de processus itératif, c'est <un jeu> de copy's multiples (autant que d'objets de travail nécessaires) de la définition de la classe

dans **`AllAvailableClasses.cpy`** combiné avec une variable de compilation les distinguant wrk_nomDeClasse_needed qu'on va y arriver.

- avant de prévoir l'inclusion des variables membres, on regarde s'il faut, et si oui, lequel ? inclure le nom d'un des objets de travail. Le tout *premier* étant précédé du nom de la structure englobante de toutes les variables et objets de travail ;
- on inclut la même (seule et unique) définition des variables membres propres reprenant celles héritées, comme pour tout objet (de travail, ou voulu par le programmeur) ;
- on regarde si on vient d'inclure le *dernier* objet de travail, si 'oui, il ne reste plus d'objet à inclure et on peut inclure toutes les autres variables de travail.

Veillons aussi à ne plus rien inclure au texte du programme si le copy est sollicité de manière excessive (**bx**).

La Fig. 37 est un exemple commenté de cette technique.

2. La deuxième technique est plus succincte, plus spontanément compréhensible, mais au contraire de la précédente, la définition de la « classe » ne tient plus dans un et un seul même copy. Ce sera toujours ce même copy de définition de classe qui sera repris partout où on devra définir un objet de la classe, mais lui-même sollicitera, *1* à *n* fois, un copy ne contenant que la définition des variables membres, celles-ci ne seront donc plus codées dans la définition de la classe, mais le fichier les contenant y sera intégré lorsqu'on déclare un objet, ainsi que sous chaque objet de travail requis par les méthodes de la classe.
   Cette technique ne requière qu'un seul copy de la définition de la classe dans le fichier **`AllAvailableClasses.cpy`** intégrant tous les copy's.

   La Fig. 38 en est un exemple commenté de cette deuxième technique.

Notons que **`AllAvailableClasses.cpy`** est aussi ajouté dans la procedure division pour y faire ajouter au texte du programmes les méthodes nécessaires, et que là aussi les copy's multiples possibles d'une même classe implémentée selon la première technique seront présentés plusieurs fois au compilateur pour inclure les méthodes des classes dont *AtLeastOne_nomDeClasse* existe et vaut 'true'. On évite très simplement d'inclure plusieurs fois les méthodes en remettant *AtLeastOne_nomDeClasse* à 'false' dès la fin du code prévu pour la procédure division, puisque cette variable de compilation ne sera plus utilisée. Toutes les sollicitations ultérieures du copy définissant cette classe ne modifieront en rien le texte du programme.
Á la fin de la partie destinée à être incluse dans la **`procedure division`** et pour autant que *AtLeastOne_nomDeClasse* valait 'true' :

```
      >>DEFINE    AtLeastOne_Deposit-BkAc as (false)     OVERRIDE JBA
    >>END-IF    *> AtLeastOne_Deposit-BkAc                       JBA
    >>DEFINE DisplayIt  as Save_DisplayIt_Deposit-BkAc    OVERRIDE JBA
  >>END-IF *> ProcedureDiv                                       JBA
```

Fig. 35 Dans la partie du copy prévue pour être incorporée en *PROCEDURE DIVISION*, penser à réinitialiser les variables

Le fichier **`AllAvailableClasses.cpy`** suivant (à solliciter en fin de **`data division`** et en fin de **`procedure division`**) répond à l'usage des deux techniques.

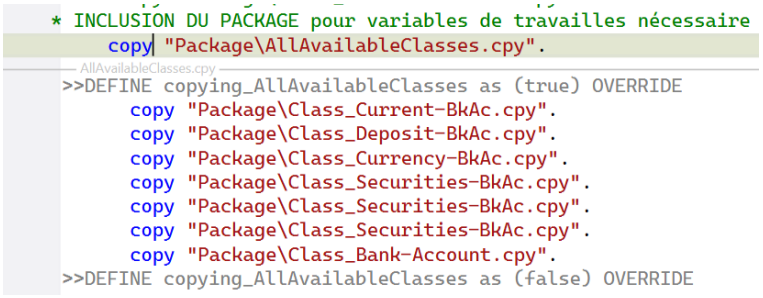


Fig. 36 Constitution d'un 'package' en assemblant de manière ordonnée les copy's des classes : toute classe fille doit précéder sa classe parent

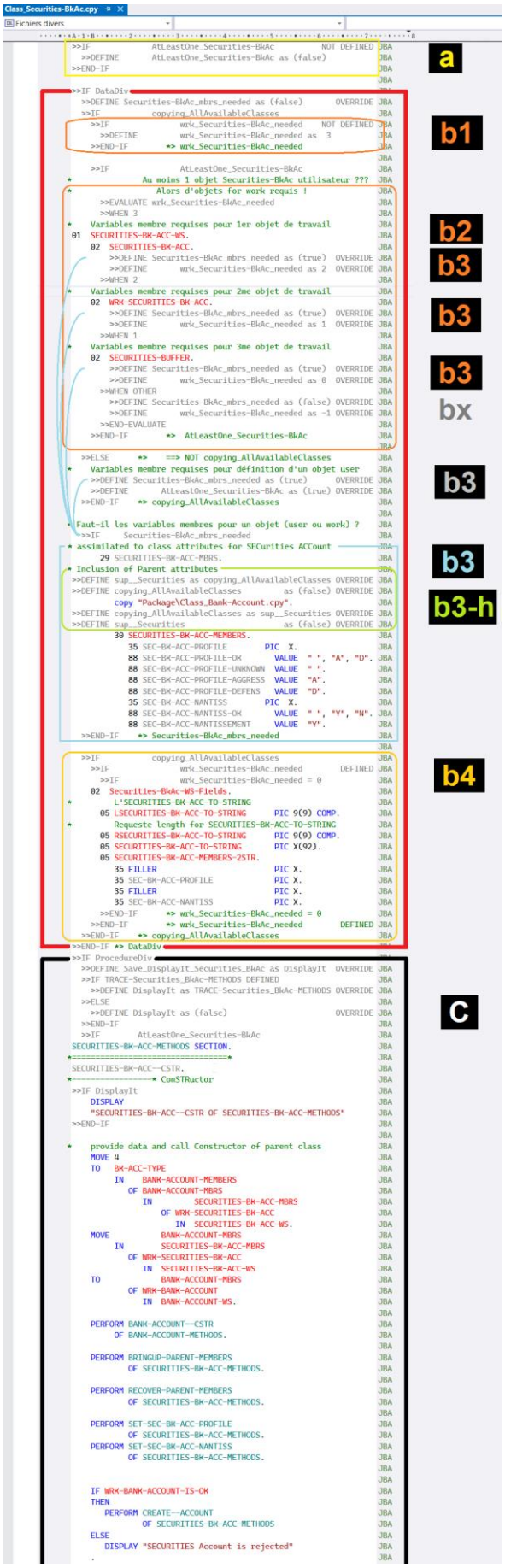


Fig. 37 Définition complète d'une classe selon la première technique

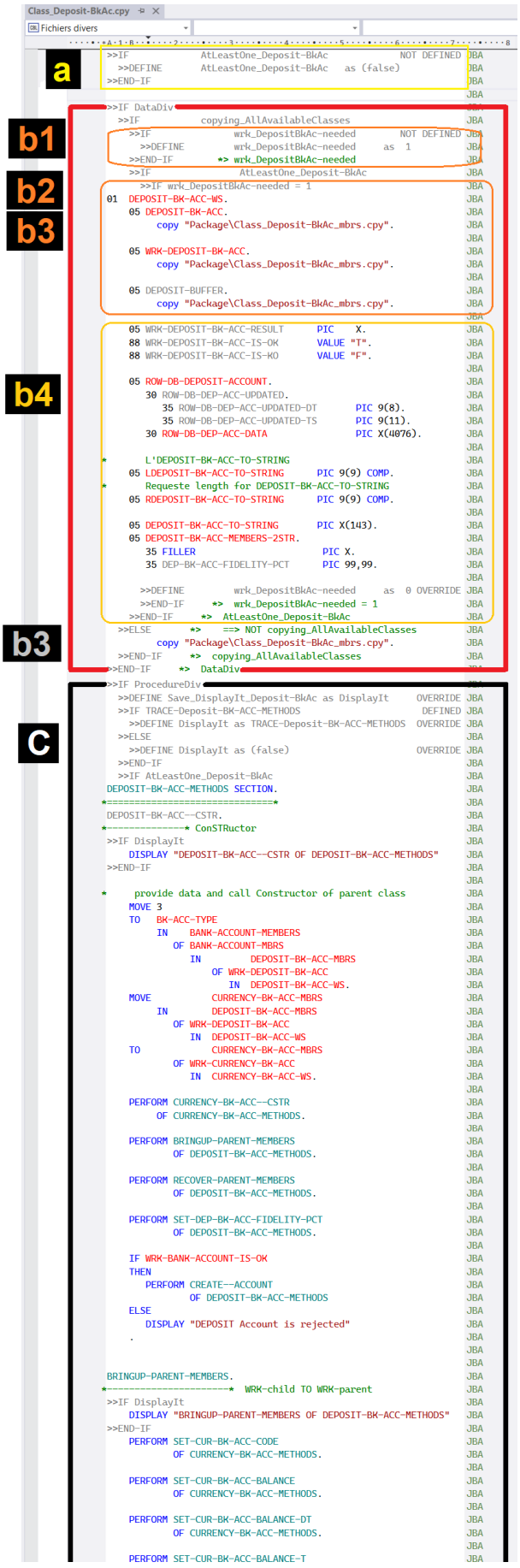


Fig. 38 Définition d'une classe selon la deuxième technique, les variables membres sont détachées

5.4.4.2 L'héritage

Grand principe de l'OO : l'héritage. Peut-on envisager de définir une classe (fille) pouvant spécialiser une classe (parent) ?

C'est possible et de manière assez simple puisque, dans la lignée de toute déclaration d'objet « utilisateur », il suffira, dans la liste des variables membres de la classe fille …

1. Si on suit la technique présentée Fig. 37 : d'inclure un copy de la définition de la classe parent. Ce copy de la classe parent va inclure ses variables membres à la définition de l'objet fille, et va automatiquement mémoriser son propre *AtLeastOne_nomDeClasseParent* qui veillera le moment venu à l'intégration de ses propres variables et objets de travail ainsi qu'à l'intégration de ses propres méthodes.
   Comme on inclut là le copy de la classe entière, et que, localisé dans une série de variables membres (de la classe fille), on doit veiller à ce que ce copy n'intégrer que les variables membres (du parents), il est impératif de <mémoriser l'état avant> et de <restaurer l'état après> de la variable de compilation **`copying_AllAvailablesClasses`** lorsqu'on crée les objets de travail utilisés par une classe fille ; le copy de la classe parent sollicité pour ses variables membre dans la définition d'un objet de la classe fille, ne doit pas se confondre avec le copy effectué par **`AllAvailableClasses.cpy`**.
2. Si on suit la technique présentée Fig. 38 : les variables membres d'une classe ayant été isolées dans un copy ne contenant qu'elles (Fig. 39), c'est dans ce copy qu'on retrouvera, si nécessaire, l'incorporation du copy des variables membres de la classe parent s'il y en a une. C'est techniquement plus simple puisque les copy's contenant les variables membres n'effectuent aucun test sur les variables de compilation.

Quelle que soit celles des deux techniques choisies, ***pour manipuler ses variables membres héritées*** de la classe parent, la classe fille devra les copier dans l'objet de travail voulu de la classe parent, faire exécuter les méthodes (paragraphes) de la classe parent, et récupérer l'objet de travail du parent dans sa propre instanciation des variables membres héritées.

Le point critique de l'<émulation de l'héritage> sera l'attribution des numéros de niveau des variables membres. Une classe fille commencera la définition de sa structure de variables membres avec un numéro de niveau inférieur de 1 ('28' est donc obtenu et utilisé ici) au moins par rapport au plus petit numéro de niveau utilisé par la structure des variables membres de la classe dont elle hérite. Il est en effet impératif que la structure de la classe fille (**28**) puisse chapeauter tant les variables membres héritées (**29**) que les variables membres spécifiques (**29**).

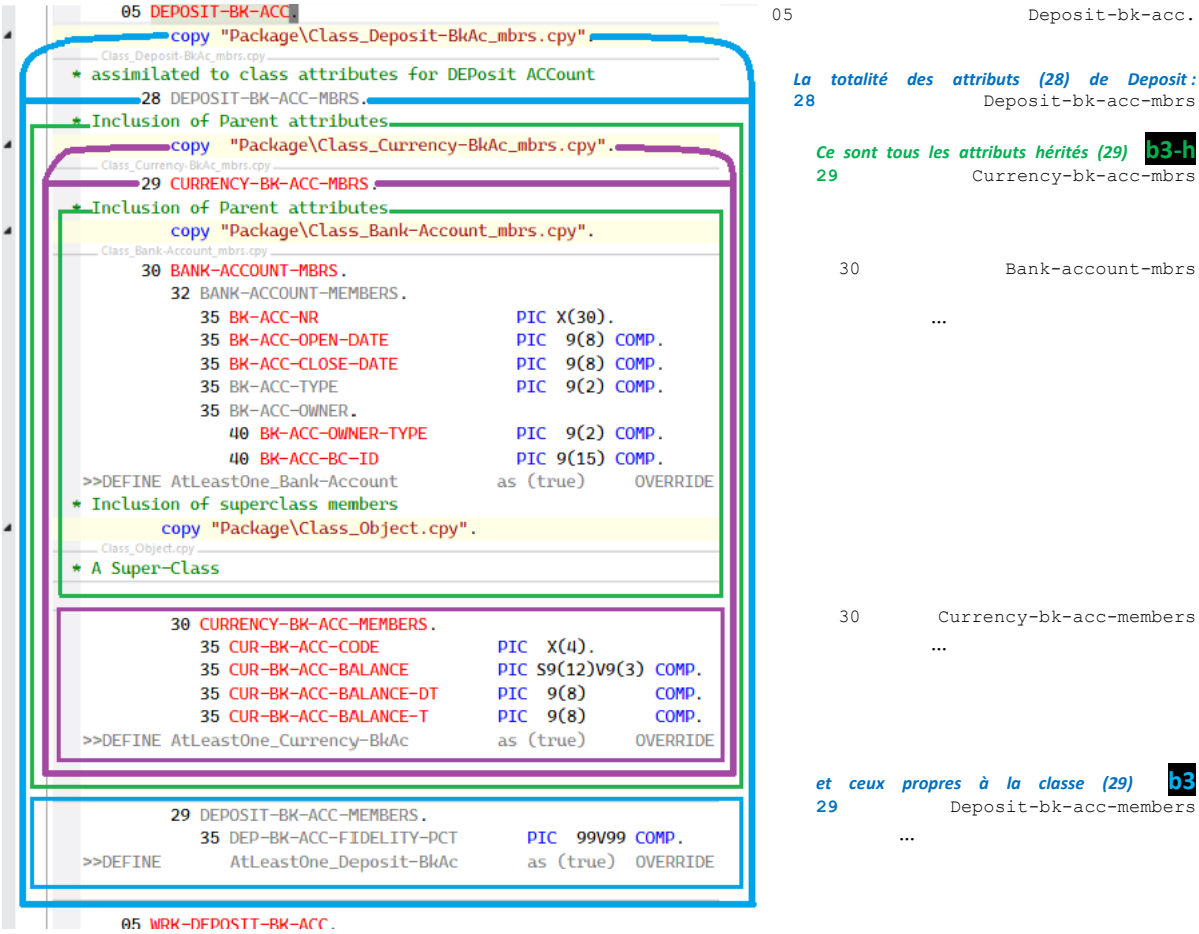


Fig. 39 Zoom sur la structuration des variables membres : héritées et propres

En résumé :

- Un peu à l'instar des « import » en java, la liste de toutes les classes utilisées doit être portée à la connaissance du programme (le copy qu'ici de manière arbitraire on a nommé **`AllAvailableClasses.cpy`**) en fin de data division, *et* en fin de procedure division ; ce copy doit au minimum contenir la liste des copy's définissant les classes utilisées en respectant la contrainte qu'un copy d'une classe fille doit y apparaitre avant tout copy d'une classe dont elle hériterait ;

  `AllAvailableClasses.cpy` pourrait être scindé en plusieurs copy's si on souhaite introduire une notion figurative de « packages »
- Dans la première technique, on répète dans **`AllAvailableClasses.cpy`** la définition de la classe autant de fois qu'il y a d'objets de travail ; (**`Securities`** est copié 3 fois)
- Dans la deuxième, on répète dans la définition de la classe l'intégration du copy des variables membres autant de fois qu'il y a d'objets de travail, plus une pour la déclaration des objets que voudraient les programmeurs. C'est le copy des variables membres (Fig. 39) qui incorpore les variables membres héritées et qui positionne *AtLeastOne_nomDeClasse* à 'true'.

5.4.4.3 Limitations

Tant qu'il s'agit du COBOL « classique » (pré-« OO ») utilisé depuis des décennies, COBOL ne connait pas le concept « objet » qu'on émule partiellement ici, il y a donc des limitations.

L'« encapsulation » n'existe qu'avec la bonne volonté du programmeur : pour le compilateur COBOL, c'est ***un*** texte source que la compilation conditionnelle transforme en ***un*** texte de programme (toutes les variables et tous les paragraphes sélectionnés) pour produire ***un*** exécutable.

La surcharge (overloading) n'est pas possible : les « méthodes » (paragraphes) n'ont pas d'argument, il n'y a donc qu'une seule signature possible par nom de méthode dans une section.

La déclaration d'un « objet » que nous obtenons en utilisant un copy définissant une « classe » va correspondre, aux yeux de COBOL, à une déclaration de structure on ne peut plus classique : adresse en mémoire et longueur en bytes, il n'y aura pas de métadonnées permettant de répondre à des questions telles que « de quelle classe est cette structure ? » ou « cette structure contient-elle une instance de la classe 𝒳 ? » Sans cette possibilité dont la réponse permettrait de lever l'ambiguïté pour choisir le paragraphe (la méthode) à utiliser : pas de polymorphisme. La redéfinition ('override') de nom de paragraphes existe bien (et c'est utilisé dans le code exemple : voir Fig. 40 pour la méthode **`TO-STRING`**), mais c'est obligatoirement dans des sections différentes, et l'ambiguïté doit être levée par le programmeur qui devra spécifier dans son code, au moment d'en solliciter l'exécution par le verbe **`perform`**, le **`nom-du-paragraphe of nom-de-section`**. Les instructions PERFORM COBOL ne sont que des branchements avec gestion automatique du retour, il n'y a donc pas de « late binding ».

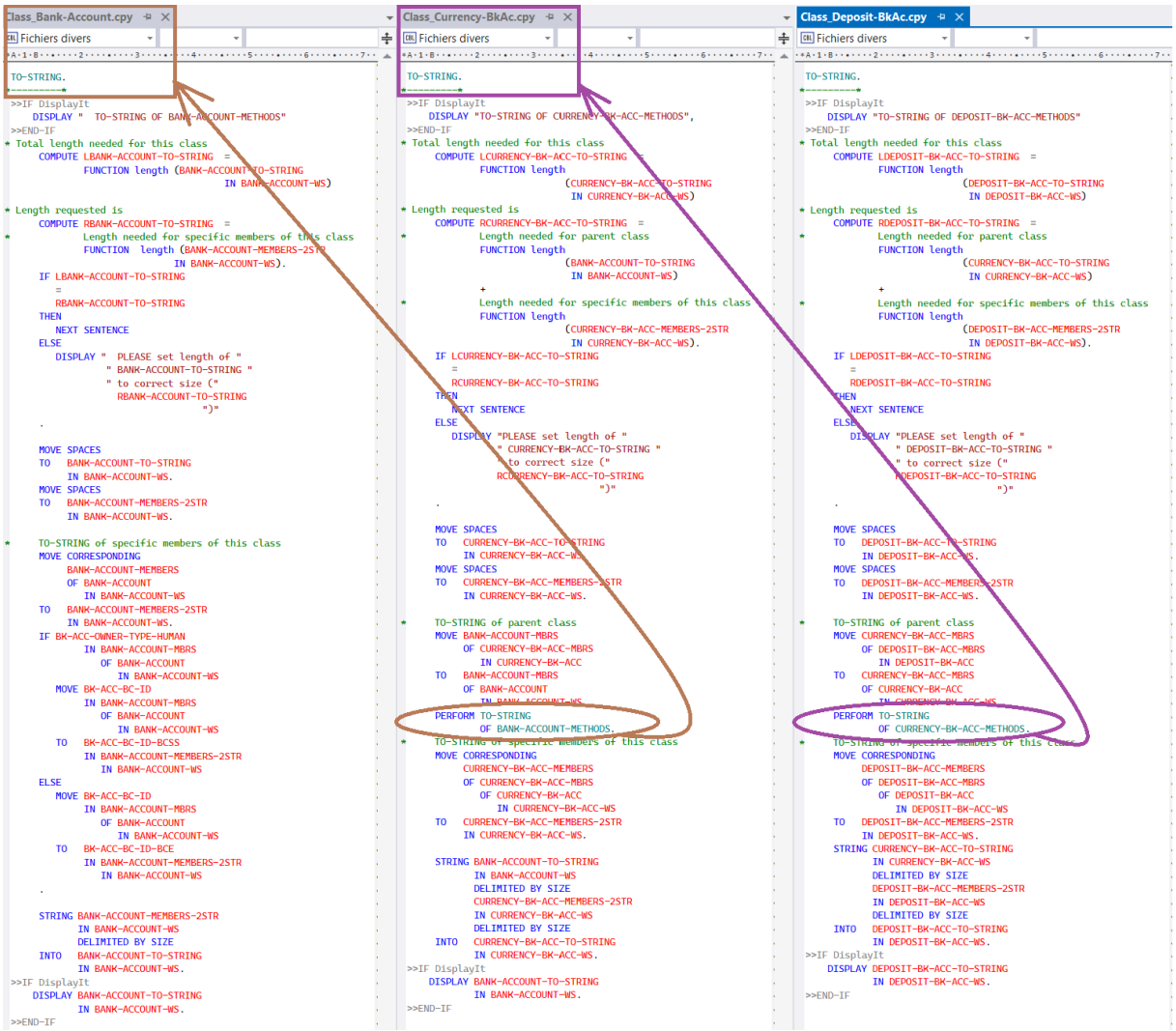


Fig. 40 Classe mère, classe fille, classe petite-fille

Remarques :

- Rappelons que la non-récursivité directe ou indirecte empêche un objet d'incorporer un objet de sa descendance ;
- Les variables membres d'une classe parent faisant partie de l'objet fille et devant être explicitement qualifiés, et les méthodes exécutées étant explicitement « performées » par l'objet fille, la possibilité de multi-héritage ne semble pas <à exclure>, le sujet sera proposé à analyse dans les *futur works*.

5.5 Résultats

La technique de la compilation conditionnelle est ancienne et éprouvée dans d'autres langages, ainsi que dans certains dialectes COBOL avec plus ou moins de facilités. Ses usages pour la réalisation de lignes de produits logiciels sont connus. Des lignes de produits logiciels sont donc maintenant réalisables en COBOL sans devoir recourir aux codes clones, ou en ne gérant ces SPL qu'au runtime. Conceptuellement, le résultat est donc positif.

Dans la pratique : le « design » du code est bien mieux constitué, et pourtant le texte du programme résultant est du COBOL classique compréhensible par tous cobolistes. De plus, *grâce à l'éditeur qui permet de faire ressortir en couleur le texte du programme de la ligne produit choisie*, la tâche de programmation n'est pas rendue plus compliquée. Les collègues sont ravis et demandeurs de pouvoir profiter de ce que cela apporte (entre autres les instructions de débogage « à la carte », le choix « à la carte » de ce qui est factorisé, ainsi que la facilité obtenue en regroupant dans un même copy <des paragraphes> et <les déclarations spécifiques qui leur sont nécessaire>). Ce résultat est donc aussi encourageant. En revanche les anciens programmeurs COBOL peinent à implémenter eux-mêmes ces usages. Par exemple et alors que c'est le bénéfice accessible le plus simplement : on retrouve en production des nouveaux DISPLAY qui n'ont pas été soumis à la condition minimum >>IF NOT FORPROD (manque de temps ? pas encore le réflex ? réticence au changement ?…).

L'émulation de l'orienté objet n'a par contre pas été confrontée au regard des collègues, ça ne reste pour l'instant qu'un petit défi personnel. Grace à l'autorisation d'utiliser la nuit le mainframe Unisys qui exploite sa compilation conditionnelle propre ($SET OMIT), le concept a malgré tout été codé, compilé, exécuté et testé avec succès. Le concept a été adapté et compilé avec succès en utilisant la norme 2002 implémentée par Micro Focus Cobol, il reste à en tester l'exécution. L'objectif d'avoir une structure de « copy pour définir une classe et instancier des objets » qui, en respectant les règles d'emploi, fournira aux bonnes places dans le texte du programme tout ce qui est nécessaire et seulement ce qui est nécessaire, est en tout cas atteint.

5.6 Discussion

La technique n'est pas compliquée et peut être enseignée en peu de temps, elle demande cependant de connaitre son compilateur : « respecte-t-il la norme ? dans son entièreté ou va-t-il plus loin (p.ex. en termes de nombre de niveaux d'imbrication de "COPY") ? ». Pouvoir disposer d'un éditeur de sources à coloration syntaxique et capable d'interpréter à la volée les instructions de compilation conditionnelle et un atout certain.

Le manque de recul et l'absence d'un échantillon de programmeurs plus jeunes sont sans doute des carences à l'évaluation de la recherche puisque seul l'usage de codes préparés a pu être évalué et pas la production de tels codes. Il semble cependant que l'essayer c'est l'adopter. L'usage de la compilation conditionnelle dans d'autres langages témoigne en tout cas en sa faveur.

Quant à l'émulation de l'OO en définissant des classes et en instanciant des objets, le « retour sur investissement » peut sembler moins évident. Elle a par contre l'avantage de pouvoir être progressive

(quelques classes à la fois) et étalée sur de nombreux releases : on reste sur la production d'un texte de programme en COBOL « classique » et donc pas de « big bang » de changement de paradigme.

En 2019 Paul Gazillo et Shiyi Wei [107] ont proposé, en vue de simplifier la compréhension des codes sources, que la compilation conditionnelle disposant de ses instructions spécifiques, puisse être remplacée par une classique compilation de source (sans langage ou balisage spécifique) qui profiterait de la puissance des compilateurs à rechercher, localiser et exclure du code mort. Ceci est inapplicable pour l'exploitation de la présente technique puisque le langage COBOL ne permet pas d'avoir des instructions d'évaluation « statement COBOL » en dehors de la procedure division.

Chapitre 6

Conclusion

Le langage COBOL souffre d'une mauvaise image : langage « legacy », verbeux par définition et par constitution, à la structure très spécifique (4 divisions, de multiples sections), … Il est pourtant présent pour assurer le core business de bien des entreprises, administrations, … N'intéressant pas, ou peu, les jeunes générations de programmeurs, il y a un risque, dans un futur plus ou moins proche, de se retrouver avec un déficit de programmeurs COBOL.

Présente pour nombre de langages, la compilation conditionnelle existe aussi pour le langage COBOL sous des formes variées, au gré des éditeurs de compilateurs. Depuis 2002 la norme COBOL définit une syntaxe et un fonctionnement précis pour la compilation conditionnelle en COBOL. Des grands éditeurs de compilateurs, tels IBM et Micro Focus, ne supportent cette norme que depuis environ 6 ans. Bien avant déjà, une compilation conditionnelle disponible avec les compilateurs d'Unisys, dont le compilateur COBOL, avait permis des usages un peu plus étendus que ce pour quoi elle était habituellement utilisée.

Imprégné du concept avec le compilateur Unisys, l'idée vint alors d'utiliser les possibilités de la compilation conditionnelle définie par la norme pour « travailler » les sources COBOL afin de factoriser et regrouper ce qui avait un sens commun. Un des buts étant d'offrir des possibilités de choix « à la carte » et pouvoir constituer des lignes de produits logiciels, afin aussi, et surtout, de tendre vers des fichiers de codes moins rébarbatifs mais au « look and feel » plus contemporain, et pourquoi pas vers de l'émulation de l'aspect de l'orienté objet.

La littérature parle très peu de `cobol et compilation conditionnelle`, mais elle parle de `cobol` et elle parle de `compilation conditionnelle`.
Sur base de la norme, de manuels de références, … une introduction sommaire au langage COBOL a été produite pour permettre de comprendre les exemples qui sont élaborés.
Ce fut ensuite la recherche d'informations sur la compilation conditionnelle et ses usages et sur les formes prises par des implémentations (dans des manuels de références COBOL plus spécifiquement). Ces diverses formes ont été comparées entre elles.

En utilisant la compilation conditionnelle telle que définie dans la norme COBOL 2002, des structures de code source COBOL accompagnées d'exemples de leurs usages dans les textes sources pour être incorporé au texte du programme ont été élaborées de manière progressive. Mélangeants déclarations et instructions, diverses structurations de fichiers sources incorporant de la compilation conditionnelle ont été produits. Le résultat est positif lorsqu'à l'usage d'un tel fichier, chaque bloc (entre directives de compilation conditionnelle) de code cobol qu'il contient, trouve précisément la place qui lui est destinée dans le texte du programme et contribue ainsi au bon fonctionnement du processus.

Ces exemples résultent de la transposition en COBOL de finalités de la compilation conditionnelle trouvées dans la littérature, innovent ou restituent l'équivalent de tâches dévolues aux compilateurs de langages OO notamment.

Le résultat est d'avoir une technique permettant de proposer des codes sources découpés, structurés, sélectionnés, assemblés … avec un « design » embrassant des concepts plus contemporains *-jusqu'à certains aspects de l'orienté objet.* Cela permet de tendre petit à petit vers un texte source permettant des choix (SPL) tout en produisant un texte de programme COBOL structuré, exempt de codes inutiles puisqu' ayant été modulé à la carte.

Quant à l'émulation de l'OO : dans la perspective d'une migration, soutenu par Krüger et al. mentionnant une conclusion de Jordan et al. (*Op.cit.*), il est possible que cela devienne encouragé par une hiérarchie. Une migration automatisée vers un véritable langage OO sur base de cette émulation pourrait certainement être le sujet d'une future recherche.
Une autre recherche intéressante concernant cette émulation de l'OO serait celle d'évaluer, et probablement confirmer, la possibilité effective de multi-héritage : le nom des « méthodes » (paragraphes) étant chaque fois qualifiés, il ne devrait pas y avoir de confusions.

Bibliographie

[1] J. W. Backus *et al.*, « The FORTRAN automatic coding system », in *Papers presented at the February 26-28, 1957, western joint computer conference: Techniques for reliability*, in IRE-AIEE-ACM '57 (Western). New York, NY, USA: Association for Computing Machinery, févr. 1957, p. 188–198. doi: 10.1145/1455567.1455599.

[2] « Report to CONFERENCE on DATA SYSTEMS LANGUAGES Including INITIAL SPECIFICATIONS for a COMMON BUSINESS ORIENTED LANGUAGE (COBOL) for Programming Electronic Digital Computers ». U.S. Government, Department of Defense, avril 1960. Consulté le: 8 février 2023. [En ligne]. Disponible sur: http://www.bitsavers.org/pdf/codasyl/COBOL_Report_Apr60.pdf

[3] *Systems manual for 704 Fortran and 709 Fortran*. in IBM. 1960. Consulté le: 10 février 2023. [En ligne]. Disponible sur: https://w.bitsavers.orgw.bitsavers.org/pdf/ibm/fortran/FORTRAN_704_709_Systems_Manual-1960.pdf

[4] B. M. Leavenworth, « FORTRAN IV as a syntax language », *Commun. ACM*, vol. 7, n° 2, p. 72–80, févr. 1964, doi: 10.1145/363921.363932.

[5] G. Radin et H. P. Rogoway, « NPL: highlights of a new programming language », *Commun. ACM*, vol. 8, n° 1, p. 9–17, janv. 1965, doi: 10.1145/363707.363708.

[6] « ECMA STANDARD on FORTRAN », *Ecma International*, avril 1965. https://www.ecma-international.org/publications-and-standards/standards/ecma-9/ (consulté le 22 juillet 2023).

[7] *IBM Operating System/360 PL/I Language Specifications*. in IBM, no. S360-29. 1965. Consulté le: 22 juillet 2023. [En ligne]. Disponible sur: http://archive.org/details/bitsavers_ibm360pliCpecifications-Jul65_11929709

[8] *American National Standard FORTRAN, ANSI X3.9-1966*. 1966.

[9] *IBM System/360 FORTRAN IV Language*, vol. C28-6515-6. in IBM, no. S360-25, vol. C28-6515-6. 1966. [En ligne]. Disponible sur: http://www.bitsavers.org/pdf/ibm/360/fortran/C28-6515-6_FORTRAN_IV_Language_1966.pdf

[10] M. Richards, « BCPL: a tool for compiler writing and system programming », in *Proceedings of the May 14-16, 1969, spring joint computer conference*, in AFIPS '69 (Spring). New York, NY, USA: Association for Computing Machinery, mai 1969, p. 557–566. doi: 10.1145/1476793.1476880.

[11] *B2500 and B3500 SYSTEMS COBOL REFERENCE MANUAL*. in Burroughs Corporation, no. 1033099. 1970. Consulté le: 22 juillet 2023. [En ligne]. Disponible sur: http://archive.org/details/bitsavers_burroughsB00B3500COBOLSep70_21183699

[12] C. J. Shaw, « FORTRAN information bulletin », *SIGPLAN Not.*, vol. 6, n° 10, p. 50–65, oct. 1971, doi: 10.1145/1317448.1317452.

[13] F. P. Mathur, « A brief description and comparison of programming languages FORTRAN, ALGOL, COBOL, PL/1, and LISP 1.5 from a critical standpoint », JPL-TM-33-566, sept. 1972. Consulté le: 20 juillet 2023. [En ligne]. Disponible sur: https://ntrs.nasa.gov/citations/19720025540

[14] *B6700 SOFTWARE IMPROVEMENTS MARK II.3*. in Burroughs Corporation. 1972. [En ligne]. Disponible sur: http://www.bitsavers.org/pdf/burroughs/LargeSystems/B6500_6700/softwareNotes/B6700_SoftwareNotes_II.3_Oct72.pdf

[15] *B5500/B5700 SYSTEM SOFTWARE MARK XV.3.0*. in Burroughs Corporation. 1974. [En ligne]. Disponible sur: https://bitsavers.org/pdf/burroughs/LargeSystems/B5000_5500_5700/listing/B5700_TS-MCP_MarkXV.3.00_Feb74.pdf

[16] *Burroughs B 6700 System Software Improvements d Note Documentation (relative to MARK 2.6 Release)*. in Burroughs Corporation. 1974. Consulté le: 26 mars 2023. [En ligne]. Disponible sur: http://www.bitsavers.org/pdf/burroughs/LargeSystems/B6500_6700/softwareNotes/5000763_B6700_Software_Notes_2.6_Apr74.pdf

[17] « PROPOSED REVISION OF AMERICAN NATIONAL STANDARD COBOL ». U. S. DEPARTMENT OF COMMERCE National Bureau of Standards, janvier 1974. [En ligne]. Disponible sur: https://www.govinfo.gov/content/pkg/GOVPUB-C13-c57731b48da07c2c703d7ed9f589d8b9/pdf/GOVPUB-C13-c57731b48da07c2c703d7ed9f589d8b9.pdf

[18] G. N. Baird, « Program debugging using COBOL '74 », in *Proceedings of the May 19-22, 1975, national computer conference and exposition*, in AFIPS '75. New York, NY, USA: Association for Computing Machinery, mai 1975, p. 313–318. doi: 10.1145/1499949.1500010.

[19] *American National Standard Programming Language COBOL, ANSI X3.23-1974*.

[20] T. S. Chow, « A generalized assertion language », in *Proceedings of the 2nd international conference on Software engineering*, in ICSE '76. Washington, DC, USA: IEEE Computer Society Press, oct. 1976, p. 392–399.

[21] T. Stuart, « Adapting large systems to small machines », in *Proceedings of the ACM SIGMINI/SIGPLAN interface meeting on Programming systems in the small processor environment*, in SIGMINI '76. New York, NY, USA: Association for Computing Machinery, mars 1976, p. 144–150. doi: 10.1145/800236.807109.

[22] *ibm :: 370 :: OS VS :: cobol :: LY28-6486-2 OS VS COBOL Compiler Rel 2 PLM Nov76*. 1976. Consulté le: 22 juillet 2023. [En ligne]. Disponible sur: http://archive.org/details/bitsavers_ibm370OSVSOBOLCompilerRel2PLMNov76_47917176

[23] *IBM OS/VS COBOL Compiler*. in IBM. 1976. Consulté le: 22 juillet 2023. [En ligne]. Disponible sur: http://archive.org/details/bitsavers_ibm370OSVSOBOLCompilerRel2PLMNov76_47917176

[24] *Basic COBOL OS/3 - Programmer Reference*. in SPERRY UNIVAC, no. UP-8057 Rev. 2. 1977. Consulté le: 21 juillet 2023. [En ligne]. Disponible sur: http://archive.org/details/bitsavers_univacsyst57r2DBasicCOBOLProgrammerReference_8844598

[25] *Federal Standard Cobol Pocket Guide*. in United States: Commerce Department: National Institute of Standards and Technology (NIST). Commerce Department, 1977. Consulté le: 21 juillet 2023. [En ligne]. Disponible sur: https://doi.org/10.6028/NBS.FIPS.47

[26] E. C. R. Hehner, « On removing the machine from the language », *Acta Informatica*, vol. 10, n° 3, p. 229-243, sept. 1978, doi: 10.1007/BF00264318.

[27] *American National Standard Programming Language FORTRAN, ANSI X3.9-1978*. 1978. [En ligne]. Disponible sur: https://nvlpubs.nist.gov/nistpubs/Legacy/FIPS/fipspub69-1.pdf

[28] *TSO-3270 Display Support and Structured Programming Facility (SPF/TSO) Version 2.2 General Information Manual*, vol. GH20-1974-1. in IBM, vol. GH20-1974-1. 1978. [En ligne]. Disponible sur: https://www.computinghistory.org.uk/downloads/10755

[29] *VAX-11 COBOL-74 Language Reference Manual*. in Digital Equipment Corporation, no. AA-C985A-TE. 1979. [En ligne]. Disponible sur: http://www.vaxhaven.com/images/2/29/AA-C985A-TE.pdf

[30] H. Samet, « A Coroutine Approach to Parsing », *ACM Trans. Program. Lang. Syst.*, vol. 2, nº 3, p. 290–306, juill. 1980, doi: 10.1145/357103.357106.

[31] V. Santhanam, « Translating non-standard extensions to standard Pascal », in *Proceedings of the May 19-22, 1980, national computer conference*, in AFIPS '80. New York, NY, USA: Association for Computing Machinery, mai 1980, p. 877–882. doi: 10.1145/1500518.1500672.

[32] F. W. Stodola, « The PLUS programming language », *SIGPLAN Not.*, vol. 15, nº 1, p. 146–155, janv. 1980, doi: 10.1145/954127.954144.

[33] *RTE FORTRAN IV Reference Manual*. in Hewlett Packard, no. 92060–90023. 1980. Consulté le: 8 février 2023. [En ligne]. Disponible sur: http://www.bitsavers.org/pdf/hp/1000/RTE-III/92060-90023_Jul-1980.pdf

[34] D. M. Jones, « Introduction to SQURL », *SIGPLAN Not.*, vol. 16, nº 2, p. 69–76, févr. 1981, doi: 10.1145/954269.954279.

[35] P. H. Joslin, « System Productivity Facility », *IBM Systems Journal*, vol. 20, nº 4, p. 388-406, 1981, doi: 10.1147/sj.204.0388.

[36] E. W. Dijkstra, « How do we tell truths that might hurt? », *SIGPLAN Not.*, vol. 17, nº 5, p. 13–15, mai 1982, doi: 10.1145/947923.947924.

[37] A. S. Tanenbaum, H. van Staveren, E. G. Keizer, et J. W. Stevenson, « A practical tool kit for making portable compilers », *Commun. ACM*, vol. 26, nº 9, p. 654–660, sept. 1983, doi: 10.1145/358172.358182.

[38] *IBM VS COBOL for OS/VS rel 2.4*, vol. GC26-3857-3. in IBM, no. S370-24, vol. GC26-3857-3. 1983. [En ligne]. Disponible sur: http://bitsavers.trailing-edge.com/pdf/ibm/370/OS_VS/cobol/GC26-3857-3_IBM_VS_COBOL_for_OS_VS_Rel_2.4_Aug83.pdf

[39] J. R. Buchanan, R. D. Fennell, et H. Samet, « A Database Management System for the Federal Courts », *ACM Trans. Database Syst.*, vol. 9, nº 1, p. 72–88, mars 1984, doi: 10.1145/348.318587.

[40] R. R. Ragan, « CYBIL: cyber implementation language », *SIGPLAN Not.*, vol. 20, nº 5, p. 21–30, mai 1985, doi: 10.1145/988327.988331.

[41] J. E. Sammet et J. Garfunkel, « Summary of Changes in COBOL, 1960-1985 », *Annals of the History of Computing*, vol. 7, nº 4, p. 342-347, oct. 1985, doi: 10.1109/MAHC.1985.10033.

[42] *American National Standard Programming Language COBOL, ANSI X3.23-1985*. 1985. [En ligne]. Disponible sur: https://www.govinfo.gov/content/pkg/GOVPUB-C13-d0cd47d3539e1d225361316057506135/pdf/GOVPUB-C13-d0cd47d3539e1d225361316057506135.pdf

[43] T. Lahey, « The Fortran 8X standard », *SIGPLAN Fortran Forum*, vol. 6, nº 3, p. 27–30, déc. 1987, doi: 10.1145/41560.41565.

[44] P. C. Capon et P. J. Jinks, *Compiler Engineering Using Pascal*. London: Macmillan Education UK, 1988. doi: 10.1007/978-1-349-10401-7.

[45] *A Series CANDE Operations Reference Manual*. in Unisys. 1991. [En ligne]. Disponible sur: https://doc.lagout.org/science/0_Computer%20Science/0_Computer%20History/old-hardware/burroughs/A-Series/MCP_3.9/86001500-000_A_Series_CANDE_Operations_Reference_Manual_Sep91.pdf

[46] *A Series COBOL ANSI-74 Programming Reference Manual*. in Unisys. 1991. [En ligne]. Disponible sur: https://bitsavers.org/pdf/burroughs/LargeSystems/A-Series/MCP_3.9/86000296-000_A_Series_COBOL_ANSI-74_Programming_Reference_Manual_Volume_1_Basic_Implementation_3.9.0_Sep91.pdf

[47] *Fortran 90 ISO/IEC 1539 : 1991 (E)*. 1991. [En ligne]. Disponible sur: https://wg5-fortran.org/N001-N1100/N692.pdf

[48] K. Lester, « Proving Correctness of Executable Programs », in *Computer Systems and Software Engineering: State-of-the-art*, P. Dewilde et J. Vandewalle, Éd., Boston, MA: Springer US, 1992, p. 325-353. doi: 10.1007/978-1-4615-3506-5_12.

[49] L. Osborne, « Teaching C with UNIX for college credit to professional programmers », *SIGCSE Bull.*, vol. 24, nº 4, p. 43–48, déc. 1992, doi: 10.1145/141837.141852.

[50] D. J. Salomon, « Four Dimensions of programming-language independence », *SIGPLAN Not.*, vol. 27, nº 3, p. 35–53, mars 1992, doi: 10.1145/130854.130859.

[51] *HP 3000 Computer Systems HP FORTRAN 77/iX Reference*. in Hewlett Packard, no. 31501-90010. 1992. Consulté le: 16 juillet 2023. [En ligne]. Disponible sur: https://support.hpe.com/hpesc/public/docDisplay?docId=emr_na-c01711105

[52] *HP FORTRAN 77/iX Reference*. in Hewlett Packard, no. 31501-90010. 1992. [En ligne]. Disponible sur: http://www.3kranger.com/HP3000/mpeix/en-mpeX/Fortran/31501-90021.pdf

[53] *SPARCompiler C_2.0 Programmers Guide*. in Sun Microsystem, no. 800-6578–11. 1992. Consulté le: 20 juillet 2023. [En ligne]. Disponible sur: https://usermanual.wiki/Document/800657811SPARCompierC20ProgrammersGuide199210.4099637115

[54] E. H. Spafford et S. A. Weeber, « Software forensics: Can we track code to its authors? », *Comput. Secur.*, vol. 12, nº 6, p. 585–595, oct. 1993, doi: 10.1016/0167-4048(93)90055-A.

[55] D. A. Wheeler, « Ada, C, C++, and Java vs. the Steelman », *Ada Lett.*, vol. XVII, nº 4, p. 88–112, juill. 1997, doi: 10.1145/260541.260547.

[56] « Information technology — Programming languages — Fortran — Part 3: Conditional compilation, ISO/IEC 1539-3 WORKING DRAFT ». 23 octobre 1997. Consulté le: 16 juillet 2023. [En ligne]. Disponible sur: https://www.iso.org/standard/29926.html

[57] A. V. Deursen et T. Kuipers, « Rapid System Understanding: Two COBOL Case Studies », présenté à 6th International Workshop on Program Comprehe++nsion, IEEE Computer Society, juin 1998, p. 90-90. doi: 10.1109/WPC.1998.693319.

[58] J. Joiner et W.-T. Tsai, « Re-Engineering Legacy Cobol Programs », *Commun. ACM*, vol. 41, p. 185-197, mai 1998, doi: 10.1145/276404.276410.

[59] S. S. Somé et T. C. Lethbridge, « Parsing Minimization when Extracting Information from Code in the Presence of Conditional Compilation », in *Proceedings of the 6th International Workshop on Program Comprehension*, in IWPC '98. USA: IEEE Computer Society, juin 1998, p. 118.

[60] D. Epstein, « More than just CoCo », *SIGPLAN Fortran Forum*, vol. 18, nº 3, p. 15–20, déc. 1999, doi: 10.1145/340103.340118.

[61] S. Baker, « The making of Orbix and the iPortal suite », in *Proceedings of the 22nd international conference on Software engineering*, in ICSE '00. New York, NY, USA: Association for Computing Machinery, juin 2000, p. 609–616. doi: 10.1145/337180.337484.

[62] Y. Hu, E. Merlo, M. Dagenais, et B. Lagüe, « C/C++ Conditional Compilation Analysis Using Symbolic Execution », in *Proceedings of the International Conference on Software Maintenance (ICSM'00)*, in ICSM '00. USA: IEEE Computer Society, oct. 2000, p. 196.

[63] P. Clements et L. Northrop, « Software product lines - practices and patterns », présenté à SEI series in software engineering, août 2001. Consulté le: 30 juillet 2023. [En ligne]. Disponible sur: https://www.semanticscholar.org/paper/Software-product-lines-practices-and-patterns-Clements-Northrop/9f9fb8a7bbf8e14ffcc82c4ccc0c58885b3dfbfa

[64] R. Lammel et C. Verhoef, « Cracking the 500-language problem », *IEEE Softw.*, vol. 18, nº 6, p. 78-88, déc. 2001, doi: 10.1109/52.965809.

[65] « Information technology — Programming languages — COBOL ISO/IEC 1989:2002(E) ». 27 janvier 2002. Consulté le: 21 juillet 2023. [En ligne]. Disponible sur: https://web.archive.org/web/20020127140324/http://www.ncits.org/tc_home/j4htm/cobolv200112.zip

[66] L. Bass, P. Clements, et R. Kazman, « Software Architecture In Practice », 2003.

[67] M. Ernst, G. Badros, et D. Notkin, « An Empirical Analysis of C Preprocessor Use », *Software Engineering, IEEE Transactions on*, vol. 28, p. 1146-1170, janv. 2003, doi: 10.1109/TSE.2002.1158288.

[68] R. H. Follett et J. E. Sammet, « Programming language standards », in *Encyclopedia of Computer Science*, GBR: John Wiley and Sons Ltd., 2003, p. 1466–1470.

[69] A. Garrido et R. Johnson, « Analyzing multiple configurations of a C program », in *21st IEEE International Conference on Software Maintenance (ICSM'05)*, sept. 2005, p. 379-388. doi: 10.1109/ICSM.2005.23.

[70] N. Veerman, « Towards lightweight checks for mass maintenance transformations », *Science of Computer Programming*, vol. 57, nº 2, p. 129-163, août 2005, doi: 10.1016/j.scico.2005.01.001.

[71] *Digital Cobol Reference Manual HP*. in Hewlett Packard, no. AA–Q2G0H–TK. 2005. Consulté le: 21 juillet 2023. [En ligne]. Disponible sur: https://manualzz.com/doc/6634209/digital-cobol-reference-manual

[72] *IBM XL Fortran Advanced Edition V10.1 for Linux*, vol. SC09-8019-00. in IBM, vol. SC09-8019- 00. 2005. Consulté le: 19 juillet 2023. [En ligne]. Disponible sur: http://www1.univ-ag.fr/c3i/xlf_compiler_reference.pdf

[73] L. A. Clarke et D. S. Rosenblum, « A historical perspective on runtime assertion checking in software development », *SIGSOFT Softw. Eng. Notes*, vol. 31, nº 3, p. 25–37, mai 2006, doi: 10.1145/1127878.1127900.

[74] B. Stroustrup, « Evolving a language in and for the real world: C++ 1991-2006 », in *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, in HOPL III. New York, NY, USA: Association for Computing Machinery, juin 2007, p. 4-1–4-59. doi: 10.1145/1238844.1238848.

[75] R. Koschke, « Identifying and Removing Software Clones », in *Software Evolution*, T. Mens et S. Demeyer, Éd., Berlin, Heidelberg: Springer, 2008, p. 15-36. doi: 10.1007/978-3-540-76440-3_2.

[76] R. M. Stallman et Z. Weinberg, *The C Preprocessor*, Free Software Foundation, Inc. 2008.

[77] B. Adams, K. De Schutter, A. Zaidman, S. Demeyer, H. Tromp, et W. De Meuter, « Using aspect orientation in legacy environments for reverse engineering using dynamic analysis-An industrial experience report », *J. Syst. Softw.*, vol. 82, nº 4, p. 668–684, avr. 2009, doi: 10.1016/j.jss.2008.09.031.

[78] S. Apel et C. Kästner, « Virtual Separation of Concerns - A Second Chance for Preprocessors. », *JOT*, vol. 8, nº 6, p. 59, 2009, doi: 10.5381/jot.2009.8.6.c5.

[79] M. Ceccato, T. R. Dean, et P. Tonella, « Recovering structured data types from a legacy data model with overlays », *Inf. Softw. Technol.*, vol. 51, nº 10, p. 1454–1468, oct. 2009, doi: 10.1016/j.infsof.2009.04.017.

[80] J. R. Cordy, « Excerpts from the TXL cookbook », in *Proceedings of the 3rd international summer school conference on Generative and transformational techniques in software engineering III*, in GTTSE'09. Berlin, Heidelberg: Springer-Verlag, juill. 2009, p. 27–91.

[81] S. Apel, C. Kästner, A. Größlinger, et C. Lengauer, « Type safety for feature-oriented product lines », *Autom Softw Eng*, vol. 17, nº 3, p. 251-300, sept. 2010, doi: 10.1007/s10515-010-0066-8.

[82] C. Kästner, P. G. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, et T. Berger, « Variability-aware parsing in the presence of lexical macros and conditional compilation », in *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, in OOPSLA '11. New York, NY, USA: Association for Computing Machinery, oct. 2011, p. 805–824. doi: 10.1145/2048066.2048128.

[83] D. Le, E. Walkingshaw, et M. Erwig, « #ifdef confirmed harmful: Promoting understandable software variation », présenté à 2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2011), IEEE Computer Society, sept. 2011, p. 143-150. doi: 10.1109/VLHCC.2011.6070391.

[84] L. Moonen, « Robust Parsing Using Island Grammars Revisited », 2011.

[85] M. Völter et E. Visser, « Product Line Engineering Using Domain-Specific Languages », présenté à Proceedings - 15th International Software Product Line Conference, SPLC 2011, août 2011, p. 70-79. doi: 10.1109/SPLC.2011.25.

[86] « Extracting Software Product Lines: A Case Study Using Conditional Compilation », présenté à 2011 15th European Conference on Software Maintenance and Reengineering, IEEE Computer Society, mars 2011, p. 191-200. doi: 10.1109/CSMR.2011.25.

[87] C. Kästner, « Virtual Separation of Concerns: Toward Preprocessors 2.0 », *it - Information Technology*, vol. 54, févr. 2012, doi: 10.1524/itit.2012.0662.

[88] R. Schadek, « DMCD A Distributed Multithreading Caching D Compiler », Universität Oldenburg, 2012. Consulté le: 20 juillet 2023. [En ligne]. Disponible sur: https://uol.de/svs/lehre/abschlussarbeiten/2012/dmcd-a-distributed-multithreading-caching-d-compiler

[89] N. Volanschi, « Safe clone-based refactoring through stereotype identification and iso-generation », in *Proceedings of the 6th International Workshop on Software Clones*, in IWSC '12. Zurich, Switzerland: IEEE Press, juin 2012, p. 50–56. Consulté le: 4 août 2023. [En ligne]. Disponible sur: https://www.researchgate.net/profile/Nic-Volanschi/publication/241633102_Safe_clone-based_refactoring_through_stereotype_identification_and_iso-generation/links/00b7d523d5b61298fc000000/Safe-clone-based-refactoring-through-stereotype-identification-and-iso-generation.pdf

[90] B. Zhang, « Extraction and improvement of conditionally compiled product line code », in *2012 20th IEEE International Conference on Program Comprehension (ICPC)*, juin 2012, p. 257-258. doi: 10.1109/ICPC.2012.6240498.

[91] S. Apel, D. Batory, C. Kästner, et G. Saake, « Feature-Oriented Software Product Lines: Concepts and Implementation », in *Feature-Oriented Software Product Lines: Concepts and Implementation*, S. Apel, D. Batory, C. Kästner, et G. Saake, Éd., Berlin, Heidelberg: Springer, 2013, p. 3-15. doi: 10.1007/978-3-642-37521-7_1.

[92] E. Bodden, T. Tolêdo, M. Ribeiro, C. Brabrand, P. Borba, et M. Mezini, « SPLLIFT: statically analyzing software product lines in minutes instead of years », in *Proceedings of the 34th ACM SIGPLAN*

Conference on Programming Language Design and Implementation, in PLDI '13. New York, NY, USA: Association for Computing Machinery, juin 2013, p. 355–364. doi: 10.1145/2491956.2491976.

[93] M. Lillack, J. Müller, et U. W. Eisenecker, « Program slicing to understand software generators », in *Proceedings of the 5th International Workshop on Feature-Oriented Software Development*, in FOSD '13. New York, NY, USA: Association for Computing Machinery, oct. 2013, p. 41–48. doi: 10.1145/2528265.2528268.

[94] M. Coughlan, *Beginning COBOL for Programmers*, 1st éd. USA: Apress, 2014.

[95] C. Bucholdt et M. Lillack, « Documentation of Recovered Architecture for Variability in Legacy Generator Systems », in *Proceedings of the 9th International Workshop on Variability Modelling of Software-Intensive Systems*, in VaMoS '15. New York, NY, USA: Association for Computing Machinery, janv. 2015, p. 19–26. doi: 10.1145/2701319.2701323.

[96] I. Chivers et J. Sleightholme, « Introduction to Programming Languages », in *Introduction to Programming with Fortran: With Coverage of Fortran 90, 95, 2003, 2008 and 77*, I. Chivers et J. Sleightholme, Éd., Cham: Springer International Publishing, 2015, p. 17-46. doi: 10.1007/978-3-319-17701-4_3.

[97] H. Jordan, J. Rosik, S. Herold, G. Botterweck, et J. Buckley, « Manually Locating Features in Industrial Source Code: The Search Actions of Software Nomads », présenté à 2015 IEEE 23rd International Conference on Program Comprehension (ICPC), IEEE Computer Society, mai 2015, p. 174-177. doi: 10.1109/ICPC.2015.26.

[98] W. Fenske, J. Meinicke, S. Schulze, S. Schulze, et G. Saake, « Variant-preserving refactorings for migrating cloned products to a product line », présenté à 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE Computer Society, févr. 2017, p. 316-326. doi: 10.1109/SANER.2017.7884632.

[99] A. Iosif-Lazar, J. Melo, A. Dimovski, C. Brabrand, et A. Wasowski, « Effective Analysis of C Programs by Rewriting Variability », *The Art, Science, and Engineering of Programming*, vol. 1, janv. 2017, doi: 10.22152/programming-journal.org/2017/1/1.

[100] B. A. Malloy et J. F. Power, « Quantifying the transition from python 2 to 3: an empirical study of python applications », in *Proceedings of the 11th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, in ESEM '17. Markham, Ontario, Canada: IEEE Press, nov. 2017, p. 314–323. doi: 10.1109/ESEM.2017.45.

[101] S. A. Masri, N. U. Bhuiyan, S. Nadi, et M. Gaudet, « Software variability through C++ static polymorphism: a case study of challenges and open problems in eclipse OMR », in *Proceedings of the 27th Annual International Conference on Computer Science and Software Engineering*, in CASCON '17. USA: IBM Corp., nov. 2017, p. 285–291.

[102] J. Melo, F. B. Narcizo, D. W. Hansen, C. Brabrand, et A. Wasowski, « Variability through the Eyes of the Programmer », présenté à 2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC), IEEE Computer Society, mai 2017, p. 34-44. doi: 10.1109/ICPC.2017.34.

[103] A. S. Nuez-Varela, H. G. Prez-Gonzalez, F. E. Martnez-Perez, et C. Soubervielle-Montalvo, « Source code metrics », *J. Syst. Softw.*, vol. 128, n° C, p. 164–197, juin 2017, doi: 10.1016/j.jss.2017.03.044.

[104] D. Watson, *A Practical Approach to Compiler Construction*. in Undergraduate Topics in Computer Science. Cham: Springer International Publishing, 2017. doi: 10.1007/978-3-319-52789-5.

[105] J. Krüger, W. Gu, H. Shen, M. Mukelabai, R. Hebig, et T. Berger, « Towards a Better Understanding of Software Features and Their Characteristics: A Case Study of Marlin », in *Proceedings of the 12th International Workshop on Variability Modelling of Software-Intensive Systems*, in VAMOS '18. New York,

NY, USA: Association for Computing Machinery, févr. 2018, p. 105–112. doi: 10.1145/3168365.3168371.

[106] R. Lämmel, « The Notion of a Software Language », in *Software Languages: Syntax, Semantics, and Metaprogramming*, R. Lämmel, Éd., Cham: Springer International Publishing, 2018, p. 1-49. doi: 10.1007/978-3-319-90800-7_1.

[107] P. Gazzillo et S. Wei, « Conditional Compilation is Dead, Long Live Conditional Compilation! », présenté à 2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER), IEEE Computer Society, mai 2019, p. 105-108. doi: 10.1109/ICSE-NIER.2019.00035.

[108] J. Krüger, M. Mukelabai, W. Gu, H. Shen, R. Hebig, et T. Berger, « Where is my feature and what is it about? A case study on recovering feature facets », *J. Syst. Softw.*, vol. 152, n° C, p. 239–253, juin 2019, doi: 10.1016/j.jss.2019.01.057.

[109] M. Marques, J. Simmonds, P. O. Rossel, et M. C. Bastarrica, « Software product line evolution: A systematic literature review », *Information and Software Technology*, vol. 105, p. 190-208, janv. 2019, doi: 10.1016/j.infsof.2018.08.014.

[110] L. Carvalho *et al.*, « Re-engineering Legacy Systems as Microservices: An Industrial Survey of Criteria to Deal with Modularity and Variability of Features », in *Handbook of Re-Engineering Software Intensive Systems into Software Product Lines*, R. E. Lopez-Herrejon, J. Martinez, W. K. Guez Assunção, T. Ziadi, M. Acher, et S. Vergilio, Éd., Cham: Springer International Publishing, 2023, p. 471-494. doi: 10.1007/978-3-031-11686-5_19.

[111] *Enterprise COBOL for z/OS 6.2 Language Reference*. in IBM, no. SC27-8713– 01. 2023. Consulté le: 21 juillet 2023. [En ligne]. Disponible sur: https://www.ibm.com/docs/en/cobol-zos/6.2?topic=pdf-version-documentation

[112] *Intel® Fortran Compiler Classic and Intel® Fortran Compiler Developer...* in Intel. 2023. Consulté le: 20 juillet 2023. [En ligne]. Disponible sur: https://www.intel.com/content/www/us/en/docs/fortran-compiler/developer-guide-reference/2023-2/overview.html