Le présent fichier PDF **Introduction et Bibliographie** concerne le document

Thèse de 3ème cycle en Informatique

Optimisation d'un compilateur incrémental et conversationnel de PL/I

Thèse de 1976 en français par Marie-Françoise Bruandet
A l'Université Scientifique et Médicale de Grenoble
Et Institut National Polytechnique de Grenoble.

Ce document au format A5 est disponible sous le numéro **AC 25989** dans l'inventaire de la collection de l'ACONIT, Association pour un conservatoire de l'informatique et de la télématique, Grenoble.

Le document comprend 3 parties, avec 12 chapitres. Il est le résultat du travail d'un « groupe mixte Université de Grenoble, Centre Scientifique I.B.M. France ».

Il traite de la réalisation d'un compilateur offrant « à l'utilisateur des facilités de modification au moment de l'édition du programme, ainsi que des possibilités d'interaction au moment de l'exécution du programme ».

Les extraits fournis dans le présent PDF sont une copie :

- **Des pages d'introduction,**
- **Du chapitre – 1 qui donne une brève description du compilateur incrémental et conversationnel,**
- **Du chapitre – 2 qui montre la nécessité d'une optimisation,**

- **De la conclusion (2 pages),**
- **De la bibliographie (2 pages).**

Pour utiliser ce document, Mme Bruandet, l'auteur de la thèse, l'Université de Grenoble, ainsi que la Base de Données ACONIT n° 25989.

Il est rappelé que la loi sur les droits d'auteur n'autorise que les reproductions partielles de documents à titre individuel et pour des motifs de recherche.

Fiche rédigée le 17/9/2019 par Jean Ricodeau.

CHAPITRE I

BREVE DESCRIPTION DU COMPILATEUR INCREMENTAL ET CONVERSATIONNEL DE PL/I

I - INTRODUCTION

Le compilateur incrémental et conversationnel de PL/1 a été réalisé par un groupe mixte 'Université de Grenoble, Centre Scientifique I.B.M. France'.

La structure de ce compilateur et les principes adoptés pour résoudre les différents problèmes ont été exposés par MM. M. GRIFFITHS et M. BERTHAUD [BERT-GRIFF] ; ont également travaillé sur ce projet Mme M. CHABRE et MM. M. PELTIER et Ph. JORRAND.

D'autres projets ont été réalisés pour d'autres langages par M. D. CLAUZEL et Mme V. BAJAR [BAJ] pour le langage FORTRAN, et MM. Y. CUNIN, B. BRETAGNOLLE, B. MAILLOT PILARD [CUN] [BRETA] pour le langage ALGOL 60.

Pour mettre en évidence les difficultés et pour qu'apparaissent plus clairement les solutions proposées ainsi que la nécessité d'une optimisation nous rappelerons les principaux éléments de ce compilateur.

Ce type de compilateur offre à l'utilisateur des facilités de modifications au moment de l'édition du programme, ainsi que des possibilités d'interaction au moment de l'exécution du programme. Pour avoir le maximum de possibilités à l'exécution, contrairement à un compilateur classique, ce compilateur ne produit pas à partir du programme source un code machine (qui peut être directement exécuté par l'ordinateur.

On considère un incrément du programme source à la fois (en général)

une instruction) et toute l'information nécessaire pour générer le code machine n'est pas disponible. Le programme est donc transformé en un code intermédiaire ou pseudo-code.

Ce pseudo-code est interprété. L'interpréteur lit le pseudo-code et exécute les actions indiquées par celui-ci.

L'objectif de ce compilateur incrémental et conversationnel est de permettre à l'utilisateur un "dialogue avec son programme".

Ce dialogue peut avoir lieu aussi bien au moment où il crée son programme à la console, c'est-à-dire à l'édition, que pendant l'exécution (partielle ou totale) de celui-ci. L'exécution peut d'ailleurs être demandée par l'utilisateur, même s'il existe des erreurs syntaxiques dans le programme.

Il peut aussi arrêter l'exécution et introduire une ou plusieurs instructions consécutives, les exécuter immédiatement sans les intégrer dans le programme source.

L'utilisateur travaille sur une partie de programme, que l'on appelle segment. Un segment est simplement une suite d'instructions. C'est sur un segment, et un seul à la fois, que l'utilisateur a la possibilité d'agir : c'est-à-dire le créer, le modifier ou l'exécuter.

Pour mémoire, nous rappelerons quelques autres caractéristiques du compilateur ; en particulier, un segment est limité à 255 incréments et le nombre d'identificateurs différents ne doit pas dépasser 128.

D'autre part, ce compilateur travaille dans l'environnement du système conversationnel LCMS [BELLINO] ; ce système a entre autres, comme tâche, d'assurer les communications avec le terminal, de gérer la mémoire, d'allouer des zones de travail à chaque terminal, et aussi de donner le contrôle, successivement à chacun des terminaux utilisés.

II - STRUCTURE DU COMPILATEUR

Le compilateur est formé de deux phases successives : un générateur et un interpréteur.

2.1 - Le générateur

L'unité de travail du générateur est l'incrément, ce qui signifie que le générateur analyse syntaxiquement un incrément indépendamment du contexte dans lequel se trouve cet incrément.

Le générateur remplit, d'autre part, les quatre fonctions suivantes :

- numérotation des incréments
- création d'un dictionnaire de commandes
- création d'une table d'identificateurs
- génération d'un pseudo-code.

. Numérotation des incréments

Le générateur numérote les incréments, et le numéro du nouvel incrément est imprimé (au début de la ligne), avant que l'utilisateur ne tape cet incrément au terminal).

Lorsqu'une erreur "locale" est détectée dans un incrément, il y a impression d'un message d'erreur. Dans le cas d'une erreur, le générateur ignore cet incrément et ré-imprime, au début de la ligne suivante, le même numéro d'incrément comme nous le voyons dans l'exemple suivant.

```
SEGMENT essai;
 NEW SEGMENT:
001 E :BEGIN;
    $
E-SYNTAX ERROR
001 e:BEGIN;
002 DCL c FIXED BIN;
003 DCL m FIXED DEC INITIAL (2.98);
004 DCL b;DCL n;


006 LIST;                                  COMMANDE DEMANDANT L'IMPRESSION DU
 ***                                       PROGRAMME EDITE
 001     e:BEGIN;
 002     DECLARE c BINARY FIXED(15,0);
 003     DECLARE m DECIMAL FIXED(5,0) INITIAL(2.98);
 004     DECLARE b;
 005     DECLARE n;
 ***

006 AFTER 1;                               INDIQUE QUE L'ON VA EFFECTUER DES
006 DCL z;                                 MODIFICATIONS APRES L'INCREMENT 1
007 LIST;
 ***
 001     e:BEGIN;
 006     DECLARE z;
 002     DECLARE c BINARY FIXED(15,0);
 003     DECLARE m DECIMAL FIXED(5,0) INITIAL(2.98);
 004     DECLARE b;
 005     DECLARE n;
 ***

007 SKIP   2;                              INDIQUE QUE L'ON SUPPRIME L'INCREMENT 2.
007 LIST;
 ***

 001     e:BEGIN;
 006     DECLARE z;
 003     DECLARE m DECIMAL FIXED(5,0) INITIAL(2.98);
 004     DECLARE b;
 005     DECLARE n;
 ***
007 AFTER 2;
E-INCREMENT NOT FOUND
007 QUIT;
 ***QUIT SEGMENT essai
```

. Dictionnaire de commande .

L'analyse de l'incrément permet d'en connaître le type, instruction BEGIN, instruction d'affectation, etc....

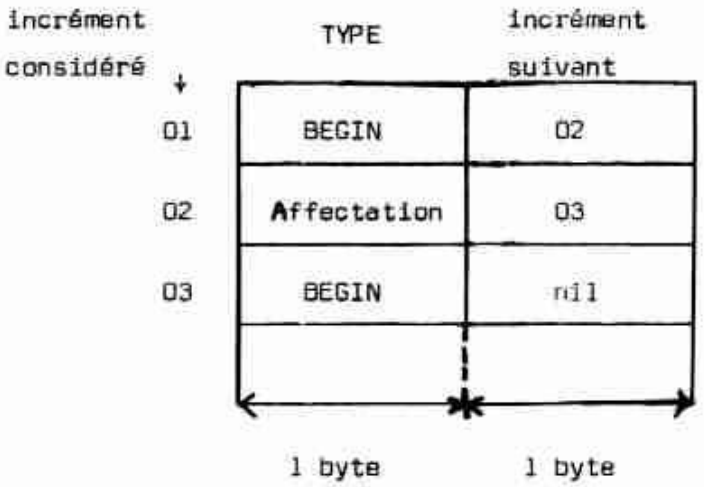
Cette information est conservée pour chaque incrément dans le dictionnaire de commande. A chaque incrément est associé le numéro de l'incrément suivant (ordre lexicographique du programme). L'ordre des instructions peut donc être modifié sans entraîner la reconstruction complète du dictionnaire de commande.

Exemple :

. Programme

```
01    BEGIN ;
02    x = 2 ;
03    BEGIN ;
04    .....
```

. Dictionnaire de commande



L'utilisateur peut modifier l'ordre lexicographique de son programme grâce aux commandes d'édition.

Si, dans le programme précédent, à l'incrément 04 on écrit :

```
04      AFTER   1 ;
04      a =     10 ;
05      .....
```

la commande AFTER 1 signifie que l'incrément 4 suivra désormais l'incrément 1 ; en fait, il sera ainsi inséré après l'incrément 1 ainsi que les incréments suivants.

Ce qui donnera pour le dictionnaire de commande :

	Type	incrément suivant
01	BEGIN [12]*	04
02	Affectation [17]	03
03	BEGIN [12]	05
04	Affectation [17]	02

*Les nombres entiers ainsi donnés sont la représentation dans le dictionnaire de commande du type des incréments.

. Création d'une table d'identificateurs.

Chaque identificateur est remplacé par un index, Deux occurrences d'un d'un même identificateur, même si elles correspondent à des variables différentes, ont la même index.

. Pseudo-code

Le pseudo-code, ou code intermédiaire, fourni par le générateur est organisé en blocs de 1 k octets, éventuellement placés en mémoire auxiliaire.

On accède au pseudo-code créé pour un incrément à l'aide du numéro de cet incrément. Ce code intermédiaire doit être compact et facile à interpréter (par exemple, les expressions sont écrites dans ce code en notation postfixée). De plus, ce pseudo-code est symétrique à quelques détails lexicographiques près ; on peut, si on le désire, recréer le programme source [CUN] . Une description du pseudo-code est donnée en annexe de ce chapitre.

2.2-L'interpréteur

L'interpréteur est constitué d'un ensemble de fonctions sémantiques qui sont activées par la commande "EXECUTE nom-de-segment ;".

Les principaux éléments utilisés par l'interpréteur sont le dictionnaire de commande et le pseudo-code.

Une table des symboles fournit les fonctions d'accès à la pile d'exécution en fonction de l'index de l'identificateur.

La pile contiendra les attributs concernant les types des variables ainsi que leurs valeurs. Du fait de la structure de blocs du langage, il peut exister, pour un même identificateur, plusieurs déclarations (dans des blocs différents).

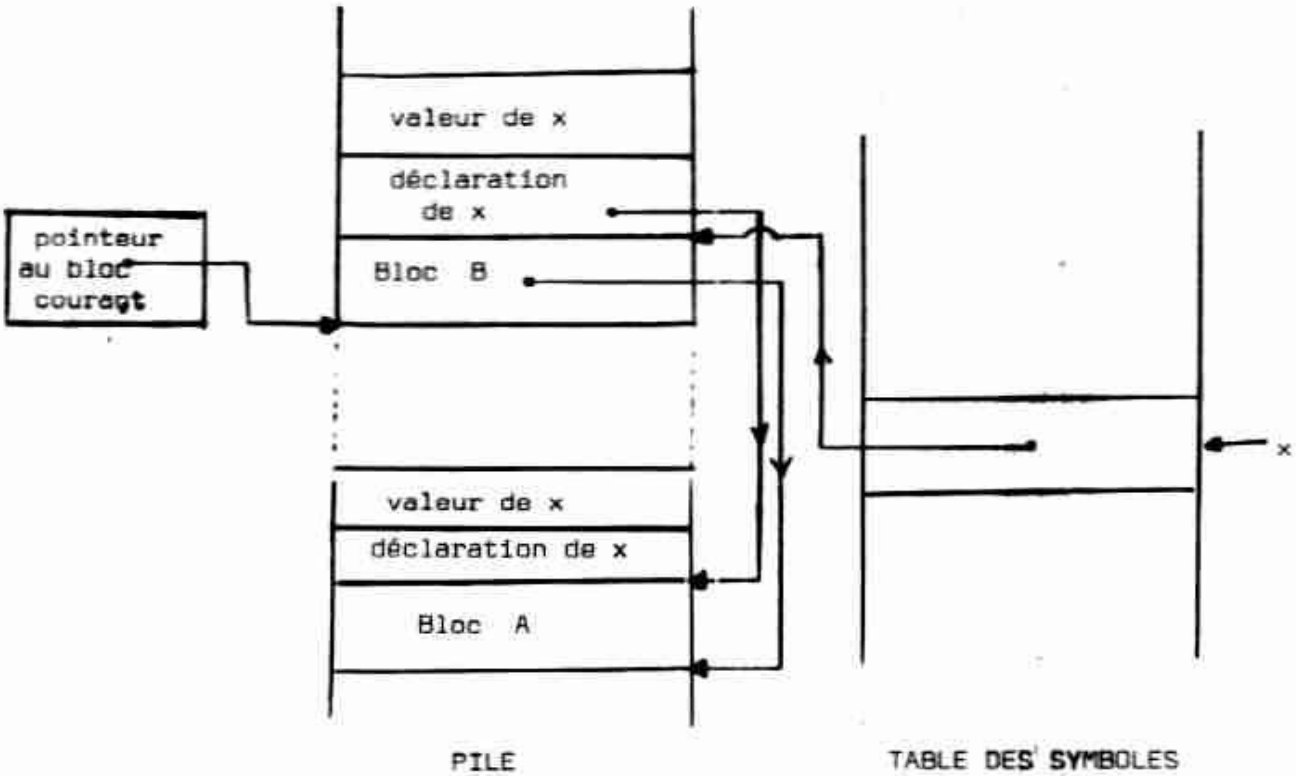
Pour un même nom d'identificateur, les déclarations sont chaînées : la représentation de la dernière déclaration traitée contient un pointeur à la précédente déclaration. La table des symboles permet, en fonction de l'index de l'identificateur, d'accéder à la dernière déclaration valide traitée. Ce chaînage sera utilisé pour la mise à jour de la table des symboles à la fin de l'exécution d'un bloc et pour la libération de la mémoire allouée aux variables automatiques de ce bloc.

De plus, la pile contient des informations donnant le type du bloc (BEGIN ou PROCEDURE) et le chaînage entre les blocs accessibles à un instant donné (chaîne dynamique des blocs).

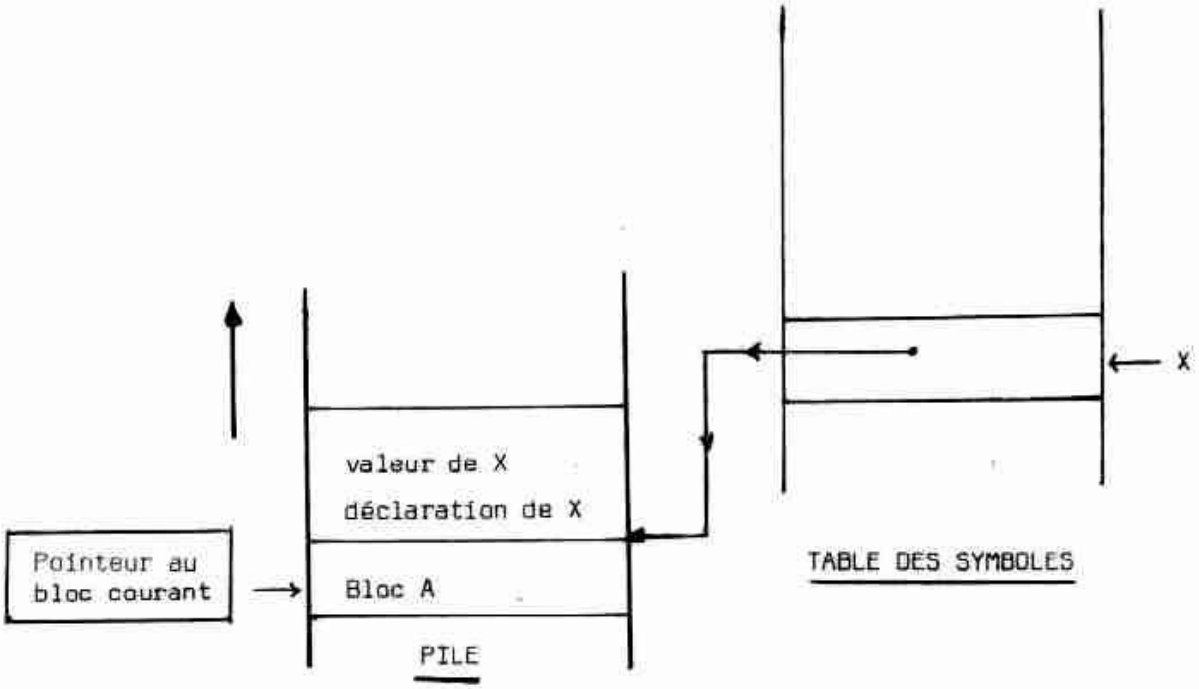
Exemple :

```
01   A : BEGIN ;
02   DCL  X ;
03   B : BEGIN ;
04   DCL  X  CHAR (9) ;
05   X : 'CHARACTER' ;
06   END B ;
07   .....
```

A l'exécution de l'incrément 5, l'état de la pile et de la table des symboles sera



A l'exécution de l'incrément 7, la pile ne contiendra plus aucune information pour le bloc B puisque celui-ci est inactif. L'état de la pile et de la table des symboles est le suivant :



Le dictionnaire de commande permet à l'interpréteur de déterminer l'ordre dans lequel les incréments doivent être exécutés.

L'interpréteur se charge donc :

- de l'adressage et de l'allocation de mémoire aux variables.
- des tests de compatibilité et des conversions des opérandes dans le cas d'expressions
- de l'exécution et de l'enchaînement des instructions.

Le travail de l'interpréteur est donc important par rapport à celui du générateur.

2.3-Niveau du langage implémenté

Le niveau du sous-ensemble de PL/I ainsi implémenté correspond au niveau du langage ALGOL 60 avec des éléments supplémentaires : notamment l'existence des variables structurées, la manipulation de variables non numériques (chaînes), la possibilité d'initialiser les variables lors de leurs déclarations.

Les entrées-sorties admises sont limitées au terminal (instructions GET et PUT avec la seule option LIST).

Les éléments admis sont les suivants :

- déclarations de variables arithmétique de classe DECIMAL, BINARY, de représentation FIXED ou FLOAT
- déclarations de chaînes de caractères ou de BIT
- déclarations de tableaux
- déclarations de structure
- les procédures
- l'instruction DO
- l'instruction IF
- l'instruction END et la fermeture multiple de blocs ou de groupes
- l'instruction GOTO
- l'instruction d'affectation

Ne sont pas pris en compte

- les ON-conditions
- les "multi-tâches"
- les entrées-sorties de type RECORD
- les données AREA et OFFSET

Les programmes ont été écrits en assembleur IBM/360 avec l'aide d'un système de macros [GRIFF.PELT] , [PELTIER] .

. Annexe au chapitre I

Le pseudo-code est donné sous forme d'une grammaire avec les conventions suivantes :

- Les caractères entre crochets sont indiqués dans le dictionnaire de contrôle
- Les caractères entre parenthèses sont dans le pseudo-code
- Les caractères entourés ni de parenthèses ni de crochets sont les éléments de la grammaire
- Une astérisque (*) indique une répétition de 1 à n fois.

Une nouvelle ligne indique une alternative, certains développements n'ont pas été donnés tels que "nombre de dim" qui est un entier.

La grammaire est donnée de façon indicative.

```
Incrément → [type-de-l'incrément]
            [étiquette*type-de-l'incrément] liste-d'étiquettes

Liste-d'étiquettes → Var (fin-de-la-partie-étiquette)
                     Var liste-d'étiquettes

Type-de-l'incrément → BEGIN
                      END
                      Déclaration
                      IF
                      ELSE
                      PROCEDURE
                      DO
                      Affectation
                      PUT LIST
                      GET LIST
                      CALL
                      RETURN
```

BEGIN → [12]
END → [7]
Endmult

Endmult → [8] (index)

Déclaration → [1] simpledec initial
[1] Arraydec initial
[1] structdec initial

Simpledec → (Index) Liste-de-type
Arraydec → borne-de-tableau
borne-de-tableau Arraydec

borne-de-tableau → (Index 5 nombre-de-dim) liste-de-type Ex^*

Ex → $Elément^*$ (9)
Element → Const
Var
Opérateur
crochet

Const → (5) liste-de-type (valeur)

Var → Id simple (Index)
Id indicé (Index nombre-d'élément) Ex^*
Id qual Id-liste

Idsimple → (1)
Idqual → (2)
Idindicé → (3)

Id-liste → (nombre-d'identificateurs $index^*$)

Structdec → (Index 4) Elément-de-structure

Element-de-structure → liste-de-structure
liste-de-structure (virgule) élément-de-structure

Liste-de-structure → (Index niveau) liste-de-type
(Index niveau nombre-de-successeurs Index*)

Liste-de-type → Arithmétique-liste
Stringlist

Arithmétique-liste → (Arithmetic coded form 1) type (prec scalefactor longueur)

type → base Echelle mode

base → (BINARY)
(DECIMAL)

Echelle → (FIXED)
(FLOAT)

mode → (REAL)
(COMPLEX)

Stringlist → (4) type-de-longueur type-de-chaîne (nombre d'éléments char*)

Type-de-longueur → (VARYING)
(FIXED)

Type-de-chaîne → (BIT) caractéristique
(CHAR)

Caractéristique → (UNALIGNED)
(ALIGNED)

IF → [16] Ex

ELSE → [15]

GOTO → [18]

PROCEDURE → [13] Spécification

Spécification → [8] (nombre-de-paramètre)liste-de nom-de-point-d'entrée(Index*)

Liste-de-nom-de-point → (Index)liste-de-type fin-de-la-partie-entryname
(Index)liste-de-type list-de-nom-de-point d'entrée

DO → [9]
[10] DOwhile
[11] DOitération

DOitération → EX TO EX
EX BY EX
EX BY EX TO EX

TO → (1)

BY → (2)

DOwhile → WHILE EX
DOitération WHILE EX

WHILE → (4)

Affectation → [17] Var* egal (nombre-d'élément-en-partie-gauche) EX

Egal → [11]

PUTLIST → [23] Id-liste

GETLIST → [22] Id-list

CALL → [19] Var

RETURN → [20]

INITIAL → (253 valeur)

CHAPITRE II

NECESSITE D'UNE OPTIMISATION

I-INTRODUCTION

La structure décrite pour le compilateur incrémental implique un lourd travail à l'interprétation soit une vitesse d'exécution relativement faible.

En effet, c'est à l'exécution (travail de l'interpréteur) que l'on effectue la plupart des tâches : reconnaissance des noms, conversions, allocation de mémoires et exécution des instructions. La "compilation" (tâche du générateur) est réduite au minimum du fait du traitement incrément par incrément.

Les performances du compilateur ont été partiellement mesurées et on peut estimer que les temps d'interprétation sont 60 à 150 fois plus élevés que les temps d'exécution des mêmes programmes traités par le compilateur PL/I (F) sur IBM 360 [BERTHAUD].

Notre critère d'optimisation pour ce compilateur sera donc le critère temps. Ce critère nous paraissant très important, nous ne nous sommes pas occupé du critère espace de mémoire nécessaire au pseudo-code. Nous utilisons un espace de travail de 4096 bytes et le bloc de pseudo-code est mis sur une mémoire auxiliaire lorsqu'il est rempli.

II - SOLUTIONS PROPOSEES

Le nombre de tâches assurées par l'interpréteur dépend essentiellement des informations fournies par le générateur au moyen du pseudo-code et du dictionnaire de commande.

Une des solutions en vue d'accélérer l'interprétation est d'interpréter un pseudo-code plus complet. Les fonctions du générateur sont alors augmentées, et il se charge, non seulement de l'analyse syntaxique, mais aussi d'une partie de l'analyse sémantique statique (autrement dit certaines liaisons entre les différents incréments sont faites à la génération : instructions IF-THEN, association BEGIN-END, etc...).

Dans le compilateur, tel qu'il existe, on ne considère qu'un incrément ; chaque modification (suppression ou création d'un incrément) ne modifie que le pseudo-code et le dictionnaire de commande relatifs à cet incrément.

Considérons, non plus un incrément, mais un groupe d'incréments, par exemple un bloc. Effectuons les liaisons existant entre les incréments de ce groupe (en particulier l'association "utilisation-déclaration" des variables). On peut alors définir, pour le générateur, une unité de travail plus grande : le groupe d'incréments choisi. Le travail de l'interpréteur est ainsi allégé ; par contre, la modification d'un incrément, dans ce groupe peut obliger à recompiler tout le groupe en tenant compte de ce nouvel incrément.

Par exemple, si le groupe choisi est un bloc, la modification d'une déclaration oblige le générateur à réexaminer toutes les expressions du bloc, et à recompiler tous les incréments du bloc.

Nous sommes alors obligés, soit de recompiler systématiquement les incréments du bloc, soit d'analyser les incréments modifiés, afin d'examiner s'ils ont une interaction sur les autres incréments du bloc.

Il y a certaines modifications qui peuvent entraîner un temps de recompilation important. Par exemple, si le segment est constitué d'un bloc, la modification d'une déclaration de ce bloc entraîne la recompilation du segment complet. Une organisation de ce type a été réalisée pour le compilateur CPL1 [CPL1] ; de plus, pour que l'association utilisation-déclarations des variables soit faite à la compilation, les déclarations doivent être mises lexicalement avant leur utilisation (sauf si le compilateur est en plusieurs passages).

Pour conserver à l'utilisateur toutes les possibilités d'interaction au moment de la création de son programme, nous ne modifierons pas les fonctions du générateur. Ce dernier est d'ailleurs satisfaisant au point de vue rapidité. Notre effort d'optimisation s'est donc porté sur l'interpréteur.

Nous nous proposons de modifier l'interpréteur en séparant les deux notions : l'association "utilisation-déclaration" des variables et l'allocation de la mémoire aux variables.

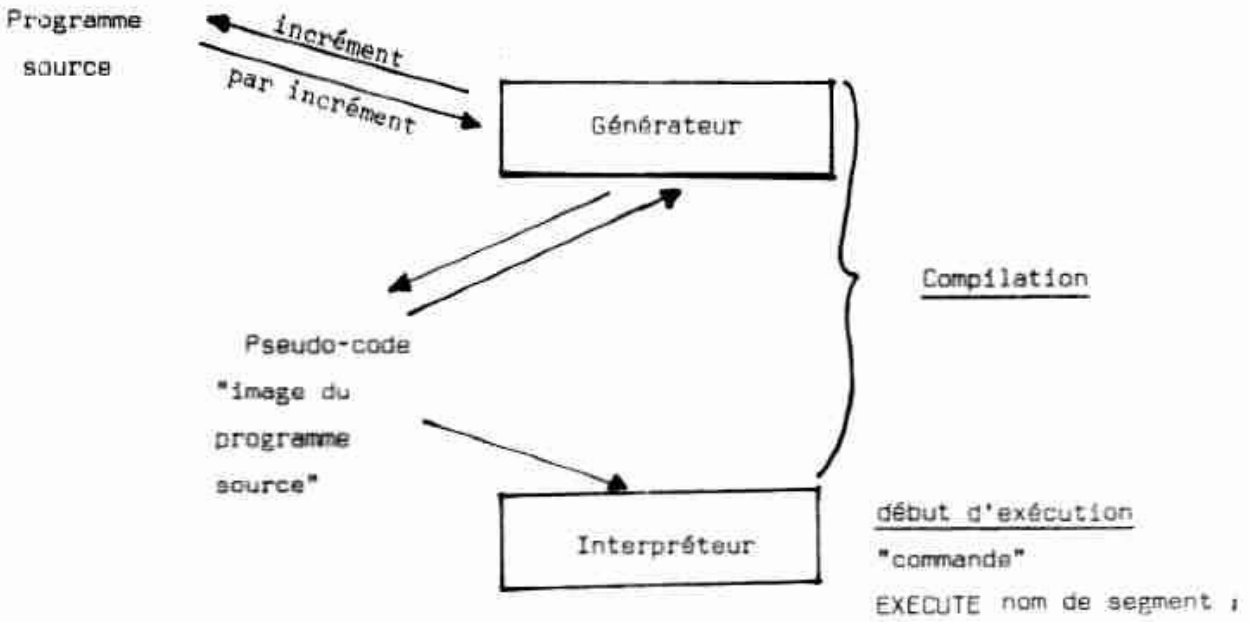
L'association des noms sera faite dans une première phase, appelée phase de pré-interprétation. Cette phase se situera après le générateur, lorsque l'utilisateur demande l'exécution de son programme. Au cours de cette phase, on effectuera certaines liaisons entre les incréments, en particulier l'association "utilisation-déclaration" des variables, dont les résultats seront transmis à la phase d'interprétation proprement dite, par l'intermédiaire d'un nouveau pseudo-code.

L'ancien pseudo-code sera conservé comme image du programme source.

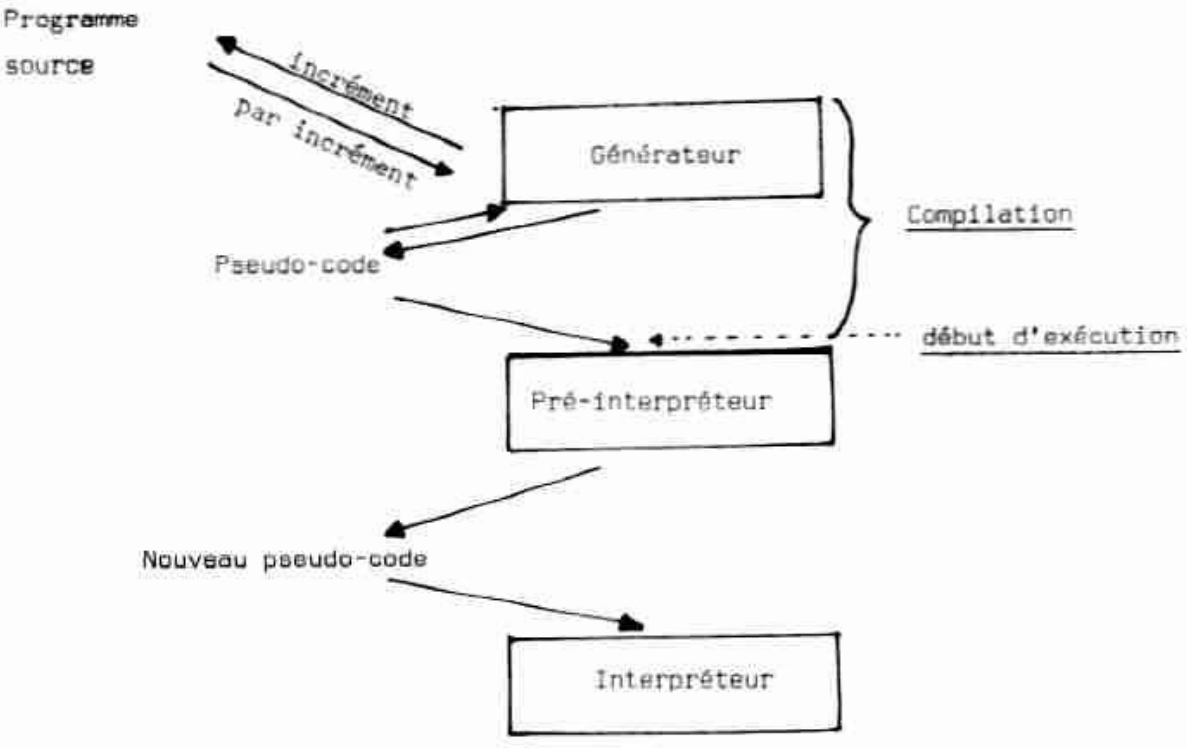
La phase d'interprétation sera chargée de l'enchaînement des incréments de l'allocation de mémoire aux variables et de l'exécution des incréments en utilisant le nouveau pseudo-code.

L'enchaînement de ces différentes phases est représenté par les schémas suivants :

Compilateur incrémental et conversationnel



Compilateur incrémental et conversationnel avec pré-interpréteur



Nous pourrions envisager également une optimisation au niveau de l'incrément [BERTHAUD].

Chaque fois qu'un incrément nouveau est exécuté, nous pourrions conserver les instructions machines nécessaires à son exécution dans le nouveau pseudo-code. C'est-à-dire que pour chaque incrément on conserve, après la première exécution, son image en langage machine (programme objet).

Cela suppose que l'on puisse connaître quel est l'état de l'incrément ainsi que ceux qui sont déjà été exécutés.

Cette étape pourrait être envisagée après la première optimisation que nous proposons.

III - PROBLEMES A RESOUDRE

Du fait de la définition incrémentale du compilateur, l'interpréteur exécute un certain nombre d'opérations. Passons en revue et analysons les principales opérations responsables de l'augmentation des temps d'exécution.

- Les déclarations sont évaluées à chaque exécution d'un bloc. Cette opération, du fait que les déclarations des identificateurs peuvent être situées après leurs utilisations, oblige à analyser tous les incréments de ce bloc.

- L'association entre les références à un identificateur et sa déclaration n'est faite qu'à l'interprétation. Ce qui implique la recherche de la déclaration d'un identificateur (apparaissant dans une instruction donnée) ; l'analyse des conversions à effectuer sur les valeurs est faite chaque fois que l'instruction est exécutée.

Par exemple, l'interprétation de l'instruction

x = 1 ;

(l'identificateur x ayant été déclaré FIXED BIN) provoquera la conversion de la constante 1 à la représentation de x, toutes les fois que l'instruction d'affectation sera exécutée.

Nous nous proposons de traiter les éléments précédents, avant l'exécution, dans la phase de pré-interprétation.

Nous conserverons le pseudo-code fourni par le générateur et créerons un nouveau pseudo-code (introduisant certaines liaisons entre les incréments, en particulier entre utilisations et déclarations des variables).

Nous distinguerons deux niveaux d'optimisation : une optimisation au niveau de la structure du segment (à l'aide du dictionnaire de commande), une autre au niveau des incréments (à l'aide d'un dictionnaire d'attributs).

Le premier niveau d'optimisation a été réalisé par [BERTHAUD].

Nous effectuons le processus d'identification des variables dans la phase de pré-interprétation à l'aide du dictionnaire d'attributs, dans lequel nous conserverons les attributs des variables. L'accès à ce dictionnaire se fera au moyen d'une table des symboles.

Pour l'allocation de la mémoire aux variables AUTOMATIC, nous utiliserons une pile (au moment de l'exécution). Une table des blocs nous permettra de savoir quels sont les blocs actifs à un instant donné.

Nous décrirons dans les chapitres suivants le rôle et la structure des éléments que nous créons (nouveau pseudo-code, dictionnaire d'attributs, table des symboles, pile, table des blocs).

De plus, du fait que l'on analyse les incréments avant l'exécution, nous pourrons traiter d'autres problèmes dans cette phase.

En particulier, le traitement des déclarations peut être long et complexe, puisque les déclarations peuvent contenir des expressions. Ces expressions définissent les longueurs des chaînes, les bornes des tableaux, les facteurs d'itérations lors d'initialisation avec l'attribut INITIAL. Les déclarations peuvent aussi faire intervenir d'autres identificateurs lorsqu'elles contiennent les attributs LIKE, DEFINED et BASED (pour les deux derniers attributs, leur évaluation ne se fait que lors de l'utilisation). Lorsque les identificateurs, intervenant dans ces déclarations, sont déclarés dans le même bloc (que les déclarations que l'on veut traiter), un problème se pose : dans quel ordre faut-il traiter ces déclarations ? Nous déterminerons, lorsque cela est possible, l'ordre d'exécution des déclarations.

Nous pourrons traiter les variables STATIC ; il est possible, avant l'interprétation, plus précisément avant l'allocation de mémoire aux variables AUTOMATIC, de connaître le nombre de ces variables ainsi que la place qui leur est nécessaire, et la réserver en début d'exécution.

Nous pourrons également traiter les variables implicites. Nous effectuons dans cette phase l'identification des variables, ce qui nous permet de déceler les variables pour lesquelles il n'existe pas de déclarations explicites. Nous effectuons, pour cet identificateur, une déclaration implicite en utilisant les attributs par défaut.

Après ces pages d'introduction et de présentation de l'outil, **Le présent PDF saute ici un grand nombre de pages descriptives des travaux effectués,** pour aller directement à la conclusion (2 pages), laquelle est suivie par la bibliographie (2 pages).

CHAPITRE III

CONCLUSION

Ce travail nous a permis de faire une analyse très précise des différentes fonctions du compilateur. En particulier, nous avons essayé de distinguer les éléments statiques et dynamiques. Cette distinction permet de construire le dictionnaire d'attributs afin d'effectuer, en particulier, l'association "utilisation-déclaration" des variables avant l'exécution, ce qui est applicable à tout autre langage (de même type global) pour la construction d'un compilateur interprétatif.

Dans cette version nous avons gagné un facteur deux (en temps d'exécution) par rapport à une version n'ayant pas de phase de pré-interprététion. On ne peut pas alourdir outre mesure cette phase de pré-interprétation sans supprimer des possibilités conversationnelles, ni augmenter trop le nouveau pseudo-code, ce qui fait que la charge de l'interpréteur reste encore lourde. L'intérêt de la phase de pré-interprétation est de permettre de traiter certains éléments absents de la première version telles que classes de mémoire et variables implicites.

De plus, pour les fonctions de conversions à appliquer aux valeurs, il semble que le temps passé pour les calculs de précisions (p,q) soit important (car les tests préalablesn'entraînent pas de gain de temps considérable). Il aurait été intéressant, puisque les valeurs de p et q sont connues lors de la pré-interprétation, de conserver comme pseudo-code le code machine nécessaire à leurs calculs, ou même de les calculer.

Une solution pour améliorer les performances de tels compilateurs sans en enlever les possibilités d'interaction est d'effectuer une optimisation au niveau de l'incrément lui-même. Une fois que l'on a interprété un incrément, on peut connaître les instruction "machines" nécessaires pour réaliser son exécution. On pourrait les conserver comme pseudo-code et les utiliser lors d'une activation ultérieure.

Le compilateur interpréteur semble un excellent outil pédagogique. Il permet aux étudiants de connaître rapidement la syntaxe de chaque instruction du langage : une instruction n'est acceptée par le générateur que si elle est correcte "localement" du point de vue syntaxique. Dans la conception des différents messages d'erreurs, il est possible d'en prévoir un certain nombre forçant l'étudiant à plus de rigueur dans la conception de ses programmes. Par exemple, on peut lui demander de déclarer explicitement les variables ou, tout au moins, le prévenir (par un message n'interrompant pas l'exécution) des types pris pour les variables non déclarées. Pour un langage comme PL/1 où beaucoup de conversions sont possibles, de tels garde-fous permettraient à l'étudiant débutant d'éviter, lors de l'exécution, des erreurs qui sont souvent difficiles à déceler.

Les compilateurs interprétatifs peuvent avoir un avenir dans l'apprentissage du langage de programmation et aussi dans la mise au point de programmes du fait des possibilités d'interaction permises à l'exécution.

Du fait de leur "inefficacité" les compilateurs interpréteurs ne sont pas compétitifs avec les compilateurs "batch", et l'on ne peut pas utiliser ces compilateurs pour une utilisation fréquente. Il faut donc que l'utilisation d'un compilateur interpréteur soit compatible avec l'utilisation d'un compilateur "batch".

BIBLIOGRAPHIE

[BAJAR] V. BAJAR, D. CLAUZEL
Le compilateur FORTRAN IV conversationnel
(Notes internes IMAG
Novembre 1970)

[BELLINO] J. BELLINO, Ph. POTIN
LCMS (sous système conversationnel fonctionnant sous OS/360)
Etude du Centre Scientifique IBM (n° FF0106).

[BERGE] C. BERGE
Graphes et hypergraphes
(Ed. Dunod).

[BERT.GRIFF] M. BERTHAUD, M. GRIFFITHS
Incremental compilation and conversationnal interpretation
Annual Review of Automatic Programming
(Vol.7 - Part 2, 1973).

[BRETA] B. BRETAGNOLLE
Compilateur incrémental d'Algol 60
(Notes Internes 1972-73).

[CPL 1] B. LORHO, M. VATOUX
CPL 1 - 1ère Partie - Analyseur générateur
Sept. 1971 - cahier n° 6.

C. BLAIZOT, L. BLAIZOT
CPL 1 - 2ème Partie - Interpréteur conversationnel
Mai 1972 - Cahier IRIA n° 10.

[CUN] Y. CUNIN, PILARD
Décompilateur algol incrémental
1969-70 - Projet de 3ème année ENSIMAG.

[GRIFFITHS 1] M. GRIFFITHS, M. PECCOUD, M. PELTIER
Incrémental Interactive compilation
(IFIP Edinbourg 1968).

[GRIFFITHS 2] M. GRIFFITHS
Cours de compilation
Université de Grenoble.

[GRIFFITHS 3] M. GRIFFITHS
Compiler Construction
Springer Verlag - Berlin Heidelberg - New-York (1974).

[GRIFF-PELT] M. GRIFFITHS, M. PELTIER
A macro-generable language for the 360 computers
(Computer . bulletin - Vol 13 n° 11, novembre 1969).

[PAIR] C. PAIR
Cours de compilation
Ecole d'Eté d'Informatique (Neuchâtel, septembre 1972).

[RAMAMOORTHY] C.V. RAMAMOORTHY
Analysis of graphs by connectivity considerations
(Journal of A.C.M., Vol. 13, n° 2, April 1966, p211-222).

[WARSHALL] S. WARSHALL
A theorem on boolean matrices
(Journal of A.C.M., Vol. 9, 11-12, Janvier 1962).