UNIVERSITE DE NANCY I
U. E. R. DE MATHEMATIQUES

compilation
dans le projet civa

thèse

pour l'obtention
du titre de
docteur ingénieur

soutenue le 10 mars 1973

par

jacques ducloy

jury :

M. C.	PAIR	Président
M. M.	DEPAIX	Examinateurs
M. J. C.	DERNIAME	

UNIVERSITE DE NANCY I

U. E. R. DE MATHEMATIQUES

compilation
dans le projet civa

thèse

pour l'obtention
du titre de
docteur ingénieur

soutenue le 10 mars 1973

par

jacques ducloy

jury :

M. C.	PAIR	Président
M. M.	DEPAIX	Examinateurs
M. J. C.	DERNIAME	

Monsieur PAIR, Directeur de l'Institut Universitaire de Calcul Automatique, est à l'origine du projet Civa, je le remercie vivement de me faire l'honneur de présider ce Jury.

Je remercie Monsieur DEPAIX, Président de l'U.E.R. de Mathématiques de l'Université de Nancy I, de l'honneur qu'il me fait en acceptant de faire partie du Jury.

Que Monsieur DERNIAME trouve, dans ce travail, l'expression de ma profonde et amicale reconnaissance pour l'attention, les précieux conseils et l'efficacité qu'il m'a apportés dans la direction de mes recherches.

J'adresse mes remerciements à l'équipe Civa, spécialement à Monsieur VIAULT pour leur collaboration, à tout le personnel de l'IUCA pour leur sympathique aide technique, à Mademoiselle LE MARECHAL, à l'I.R.E.M. et particulièrement à Monsieur DEBERDT pour leur collaboration dans la réalisation de ce travail.

à françoise

sommaire

INTRODUCTION

introduction

Donner à l'utilisateur la possibilité d'employer une formulation unique pour représenter les différentes phases de la mise en oeuvre d'un travail (conception, compte-rendu d'analyse, programmation, mise au point, exécution, maintenance), tel est le but du projet Civa [1].

Pour atteindre ce but, deux actions étaient nécessaires :
définir un langage unique décrivant à la fois les résultats de l'analyse, les programmes et leur exploitation,
construire un système de compilation acceptant ce langage.

Notre rôle, dans ce projet, était double : contribuer à la définition du langage et réaliser un système de compilation. Ce rapport contiendra deux parties distinctes : l'une se rapporte à la réalisation, l'autre à un point précis de la définition du langage (aide à la mise au point et procédés de contrôle de l'exécution).

En ce qui concerne la réalisation, nous avons commencé par concevoir un système de compilation. Celui-ci définit les liaisons existant entre les classes et les modules pendant les différentes étapes de la compilation.

Pour la mise en oeuvre proprement dite, deux problèmes sont apparus : écrire un compilateur pour un langage en cours de définition, et travailler au sein

d'une équipe d'universitaires dont les activités sont variées et absorbantes, ce qui rend difficile la communication matérielle entre les personnes, les moments de disponibilité étant souvent disjoints. Nous avons donc défini un sous-ensemble "relativement sûr" du langage Civa et nous avons voulu que le travail fait puisse être réutilisable ; ceci nous a amené à dégager les fonctions essentielles d'un générateur et, par exemple, définir un système de procédures de génération. Pour résoudre les problèmes de communication, une solution modulaire s'impose ; nous avons défini un système de programmation s'appuyant sur l'existant, permettant d'écrire un compilateur de façon modulaire. Enfin, l'analyse du générateur proprement dit a été menée en utilisant le langage d'analyse décrit dans Civa.

1
système de compilation civa

I.1. ORGANISATION GENERALE DES SYSTEMES DE COMPILATION

I.1.1. Définition

Dans ce chapitre, on entendra par système de compilation un ensemble de services (processeurs tels que compilateur, assembleur, éditeur de liens), une organisation entre ces services (par l'intermédiaire de fichiers) et certaines règles (se traduisant par l'utilisation de cartes de commande) qui permettent de faire exécuter un programme écrit en langage source.
Il s'agit d'un sous-ensemble d'un système d'exploitation classique.

L'un des buts de Civa est la réutilisation de l'existant, dans et hors Civa (c'est-à-dire la possibilité d'utiliser des modules ayant été décrits en Civa, mais aussi la conservation de chaîne de traitement ou de modules, décrits dans un autre langage). On a donc dû veiller, dans la représentation des objets élémentaires, à ne pas trop s'écarter des normes communes ; de même, l'implémentation des modules et des classes, vue de façon macroscopique, devra tenir compte des normes traditionnelles de communication entre modules au sein d'une installation. Nous allons donc étudier ces "systèmes de compilation" de manière assez générale et ensuite définir un sous-ensemble propre à la compilation Civa.

I.1.2. Rappel sur les communications entre segments

Nous entendrons par segment, des ensembles d'instructions ou de valeur ayant une unité logique, écrits séparément mais devant être regroupés pour

une exécution. C'est le cas du programme principal et des sous-programmes en Fortran, des modules et des classes en Civa. Les communications entre segments seront de deux types : communication de contrôle (un programme passant le "contrôle" à un sous-programme) et communication de valeur.

1.1.2.1. Communication de contrôle

Le principal mode de communication de contrôle est l'utilisation d'instructions de branchement spécialisées, telle l'instruction BAL (Branch and Link) sur IO 070. Elle provoque le branchement à l'adresse indiquée et le rangement du compteur ordinal dans un registre :

```
ETI     BAL,R     A
```

provoque un branchement en A et le rangement de ETI+1 dans le registre R. Il sera alors très simple de revenir dans le programme appelant par une instruction de branchement indirect.

```
        B         *R
```

D'autres modes de communications de contrôle utilisent les services du moniteur ; nous en reparlerons ultérieurement.

1.1.2.2. Communication de valeur

Deux segments peuvent communiquer entre eux à deux moments distincts. Lors de l'exécution, il faudra transmettre des paramètres ; ce problème sera évoqué ultérieurement. Lors de l'édition de liens, il faut alors permettre à deux modules, compilés séparément, d'utiliser le même emplacement. Un emplacement est repéré dans un langage de programmation par un symbole ; la

plupart désignent des emplacements locaux à un segment, mais certains autres seront communs à plusieurs segments : on parlera alors de symboles externes. Un tel symbole doit être défini explicitement dans un segment : c'est alors une définition externe. Tout module se référant à ce module doit préciser que celui-ci est défini en dehors du segment : c'est alors une référence externe. (Les directives DEF et REF remplissent respectivement ces deux rôles dans un langage assembleur).

1.1.2.3. Exemple

L'instruction FORTRAN :

```
        CALL  SSP
```

pourrait être traduite par :

```
        REF     SSP
        BAL,RL  SSP
```

de même

```
        SUBROUTINE  SSP
```

par

```
        DEF     SSP
SSP     ...
```

et

```
        RETURN
```

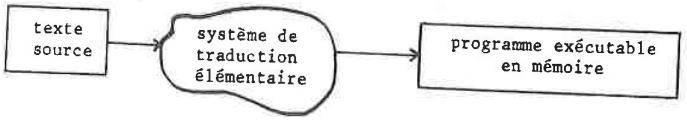
par

```
        B       * R
```

1.1.3. Eléments constitutifs d'un système de compilation

1.1.3.1. Système élémentaire

De façon très simple, nous pouvons dire que le rôle d'un système de compilation est de traduire un programme source en langage machine et de l'exécuter ; ce schéma, traduit par la figure ci-dessous, est rigoureusement celui utilisé dans des applications rudimentaires, (compilateur BASIC time-sharing, assembleur élémentaire sur n'importe quelle petite machine).



Ce schéma est souvent le seul connu du débutant qui "écrit un programme et le fait passer".

Un système de compilation évolué peut nécessiter plusieurs passages entre le texte source et l'exécution, chaque passage produit un texte intermédiaire, de plus en plus proche du langage machine ; le dernier passage résoud les problèmes qui se posent en mêmes termes pour tous les segments, quel que soit le langage source : résolution de références en avant, liaisons entre segment, allocation de mémoire. Il sera réalisé par l'éditeur de liens qui est commun aux divers compilateurs d'une installation.

1.1.3.2. Module de chargement, exécuteur

Un module de chargement est un fichier, utilisé par un exécuteur, pour

créer une ou plusieurs images-mémoire correspondant à une exécution.
L'exécuteur le plus simple est un chargeur non translatable ; le fichier est alors constitué d'un seul enregistrement : l'image exacte de la mémoire en début d'exécution.
Quelques améliorations peuvent être apportées à ce modèle très simple.

- Enchaînement automatique

  Certaines instructions permettent, à l'éxécution, de charger une autre image-mémoire, seules certaines zones de données étant sauvegardées. La procédure M:LINK sous BPM ou SIRIS 7 permet d'enchaîner deux modules de chargement, les pages dites communes étant préservées ; sur certains compilateurs Fortran existe une instruction permettant d'enchaîner en sauvegardant le CØMMØN non étiqueté.

- Recouvrement

  Il s'agit d'une amélioration du dispositif précédent permettant de charger des segments appelés à des moments différents sur les mêmes emplacements.

- Translation

  Un même module de chargement peut être exécuté n'importe où en mémoire.

Un exécuteur évolué apporte donc les avantages suivants :

- conservation du résultat d'une compilation (gain de temps considérable surtout pour des applications importantes) ;
- exécution d'un programme n'importe où en mémoire ;
- exécution d'un programme dans un espace restreint.

1.1.3.3. Description d'un module de chargement sur 10 070

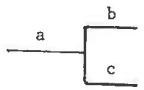
Il est impératif, sur n'importe quel système, de protéger certaines parties de la mémoire contre des débordements de l'utilisateur. En particulier, il est nécessaire que l'utilisateur ne puisse faire aucune action dans les zones réservées au moniteur, de même dans la plupart des cas, les instructions composant un programme n'ont pas, logiquement, à être modifiées pendant l'exécution ; la protection de certaines zones peut se faire par programme, mais cette solution est catastrophique du point de vue rendement ; c'est la raison pour laquelle la plupart des calculateurs sont dotés de dispositifs de protection "hardware". En particulier, sur 10 070, la mémoire est paginée, chaque page possède un code dit de protection d'accès, représenté par 2 bits.

00	tout accès autorisé
01	écriture interdite (lecture et exécution autorisées)
10	lecture seule autorisée
11	tout accès interdit

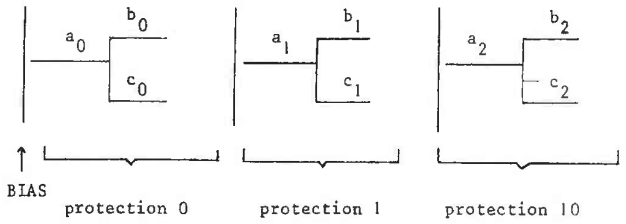
Une tentative d'écriture dans une page d'accès "01" provoque un déroutement.

Une image-mémoire est alors constituée de trois zones : variable, programme, constante.

Imaginons un programme ayant la structure de recouvrement :



il sera implanté en mémoire centrale de la manière suivante :



On trouvera dans le module de chargement :

- une description de l'arbre de recouvrement

  pour chaque segment

  - une table des définitions externes

    pour chaque zone de protection différente, par segment

    - l'image-mémoire correspondant au BIAS (limite basse de la mémoire utilisateur) fixée lors de l'édition de liens

      - un dictionnaire de translation (c'est-à-dire un octet par mot indiquant comment ce mot doit être éventuellement translaté.

1.1.3.4. Modules-objets, éditeur de liens

Un module-objet est un ensemble de directives de chargement. Une directive de chargement est, en général, une demande de chargement d'un mot comportant l'image binaire d'un mot mémoire avec des indications sur les translations éventuelles de ce mot. Le langage avec lequel est écrit un module-

objet est unique, c'est-à-dire que tout traducteur, compilateur, génère le même type de module-objet. Il s'agit en fait de l'ébauche d'une partie de l'image-mémoire définitive. (Une description plus complète sera donnée ultérieurement).

Le rôle de l'éditeur de liens est de créer un module de chargement à partir de plusieurs modules-objets.

Les modules-objets communiquent entre eux par des références et définitions externes (ce sont en fait des chaînes de caractères auxquelles une définition a été affectée dans un module).

Pour constituer un module de chargement, le programmeur fournit à l'éditeur de liens la liste des modules constitutifs (avec leur arbre de recouvrement). L'éditeur tente de satisfaire toutes les références entre ces modules ; si certaines demeurent insatisfaites, il va joindre au module de chargement certains modules de librairie contenant les définitions de ces références.

I.1.3.5. Traducteur simple, langage simple

Un traducteur simple est un processeur qui analyse un texte source écrit en langage simple pour produire un module-objet. Un programme écrit dans un langage simple ne contient que la description d'actions entreprises à l'exécution. En particulier, on ne trouvera aucune directive destinée au traducteur (hormis la description des variables).

Ce traducteur est en général à deux passages. Le premier sert à constituer une table de symboles ; pendant le deuxième passage, le traducteur génère, pour chaque instruction source, une séquence équivalente en langage machine

(ou plus précisément en module-objet). On peut trouver un troisième passage lorsqu'on fait une optimisation sur l'ensemble du texte ; le premier passage peut aussi disparaître lorsqu'on dispose d'un éditeur de liens qui satisfait les références en avant (celui du système BPM ayant cette propriété, l'assembleur symbol est à un passage). Il peut aussi disparaître dans les langages pour lesquels les déclarations figurent en tête du programme (Fortran).

I.1.3.6. Générateur, méta-traducteur, méta-langage

Un générateur est un programme qui génère un texte en langage simple à partir d'un fichier de données (générateur de tri, par exemple). Un méta-traducteur est un processeur qui génère un texte en langage simple à partir d'un texte écrit en méta-langage. Un méta-langage contiendra deux types d'instructions : des instructions du langage simple (qui seront simplement recopiées) et des descriptions de calculs à faire pendant la méta-traduction ; ces calculs vont permettre de définir de nouvelles instructions, ou de produire des textes différents en langage simple, en affectant certaines valeurs à des méta-variables. On peut remarquer que les directives propres au méta-langage sont simplement interprétées et qu'il n'en subsiste aucune trace dans les phases suivantes. On donnera ultérieurement une description précise de Métasymbol, méta-assembleur utilisé sur IO 070 ; nous pouvons citer Méta Cobol qui fonctionne de façon analogue en ayant Cobol pour langage simple.

Les limites indiquées ici entre méta-traducteur et traducteur simple sont relativement théoriques ; en particulier, certains méta-traducteurs ne font pas appel effectivement à un traducteur simple, mais produisent un résultat équivalent. (C'est le cas de Métasymbol qui produit directement du Module-objet ; cela va même plus loin, car, sous SIRIS 7, par exemple, l'assembleur Symbol n'est pas implanté et le méta-langage supplante définitivement le langage simple).

1.1.4. Notion de section

Nous avons vu qu'un programme à l'exécution pouvait se décomposer suivant trois zones différentes. Il va donc falloir décrire simultanément plusieurs zones distinctes en mémoire. Cela est possible grâce à l'utilisation de sections. Une section est un ensemble d'emplacements consécutifs regroupés de façon logique. Des directives permettent, aux processeurs de traduction, de générer des données dans différentes sections.

1.1.4.1. Section de contrôle

Une section de contrôle est une section entièrement définie à l'intérieur d'un module-objet ; beaucoup de modules-objets disposent donc d'au moins deux sections ; l'une, contenant les variables, sera sans protection, l'autre sera en protection d'écriture pour éviter une altération du code lors de l'exécution, par suite d'un débordement. Un autre intérêt des sections est que certains traducteurs autorisent d'ouvrir un nombre quelconque de sections et de générer alternativement dans l'une quelconque de sections, ce qui donne beaucoup de souplesse au traducteur.

1.1.4.2. Exemple d'utilisation de section de contrôle en Métasymbol

Nous donnerons ultérieurement une présentation sommaire de Métasymbol. Nous nous bornerons à préciser quatre directives :

```
        CSECT       i
```

où i est une protection d'accès (0, 1, 2) ; cette directive permet de déclarer une section et d'initialiser le compteur d'emplacement au début de cette section.

```
X       EQU         $
```

permet d'affecter à la constante d'assemblage X la valeur du compteur d'emplacement.

```
        DATA        exp
```

permet de générer l'expression "exp" dans le mot repéré par le compteur d'emplacement et d'incrémenter celui-ci de 1.

Cette directive est équivalente à :

```
X       EQU         $
        DATA        exp
```

La directive

```
        USECT       X
```

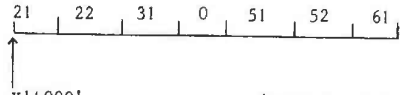
permet de revenir dans une section préalablement définie "X" repérant un emplacement quelconque de cette section, (le compteur d'emplacement prend pour valeur celle du dernier compteur d'emplacement positionné dans cette section).

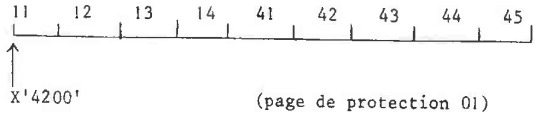
Envisageons les deux textes sources métasymbol :

```
        CSECT   1
X1      EQU     $
        DATA    11
        DATA    12
        CSECT   0
X2      DATA    21
        CSECT   0
X3      DATA    31
        USECT   X2
        DATA    22
        USECT   X1
        DATA    13
        DATA    14
        END
```

```
        CSECT   1
X1      DATA    41
X2      DATA    42
        CSECT   0
X3      DATA    51
        DATA    52
        USECT   X1
        DATA    43
        DATA    44
        CSECT   0
        DATA    61
        USECT   X2
        DATA    45
        END
```

Si nous faisons un seul module de chargement, l'image mémoire sera obtenue en mettant bout à bout les diverses sections de même protection. Si le module de chargement est implanté en X'4000', nous aurons :





Remarquons que le mot X'4003' contient 0, car une section occupe technologiquement un nombre entier de doubles mots.

1.1.4.3. Sections fictives

Une section fictive est une section repérée par un nom (c'est-à-dire une chaîne de caractères). Elle est telle que, si plusieurs modules-objets contiennent chacun une section fictive de même nom, celle-ci ne figure qu'une seule fois dans le module de chargement produit. La zone mémoire finalement générée dépendra des diverses directives données dans chaque module ; en particulier, le résultat sera imprévisible si deux modules assignent des valeurs différentes au même emplacement.

1.1.4.4. Exemple d'utilisation de sections fictives en Métasymbol

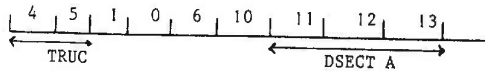
Envisageons les trois programmes sources

```
S1      DSECT   1
        RES     2
        DATA    3
TRUC    DSECT   0
        DATA    4
        DATA    5
        CSECT   1
        DATA    14
        DATA    15
        CSECT   0
        DATA    1
        END
```

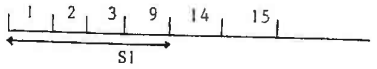
```
S1      DSECT   1
        DATA    1
        DATA    2
        CSECT   0
A       DATA    6
TRUC    DSECT   0
        DATA    4
        DATA    5
        USECT   S1
        RES     1
        DATA    9
        USECT   A
        DATA    10
        END
```

```
S1      DSECT   1
        RES     3
A       DSECT   0
        DATA    11
        DATA    12
        DATA    13
        END
```

Dans les mêmes conditions que dans l'exemple précédent, on aura :



X'4000'



X'4200'

1.1.4.5. Utilisation des sections dans les langages évolués

a) FORTRAN : Dans chaque module (programme ou sous-programme), on définit deux sections, l'une en protection 0 pour les variables et mémoires de travail, l'autre pour les constantes et le code généré en protection 1. Des sections fictives sont utilisées pour chaque CØMMØN étiqueté ; l'instruction

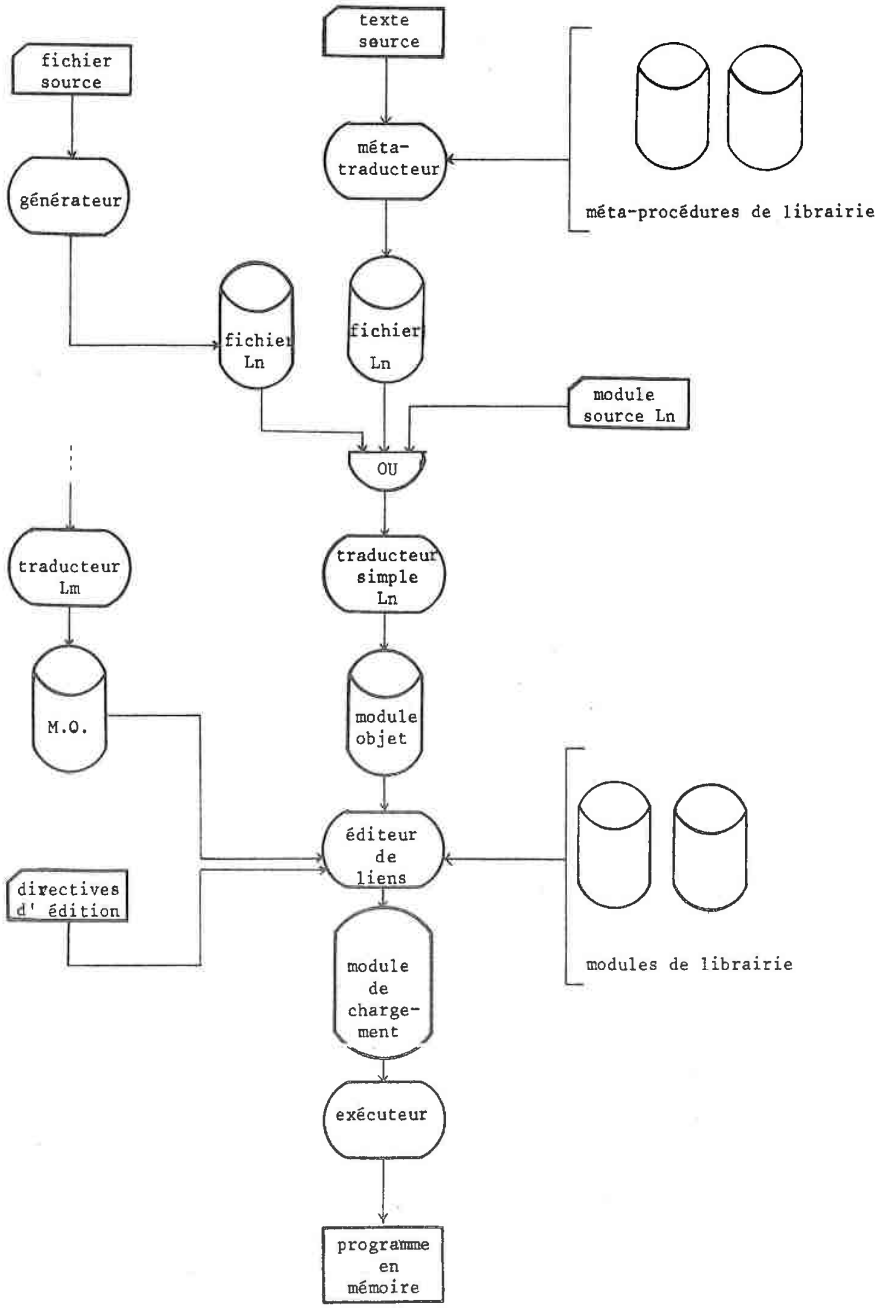
```
CØMMØN  /ETI/  X(200)
```

est équivalente à :

```
ETI  DSECT  0
     RES    200
```

b) COBOL : Un programme produit une section de commande et plusieurs sections fictives ; la section de commande contient les codes instructions, les sections fictives sont utilisées pour la Common Storage Section, les zones articles des fichiers et les DCB associés.

SCHEMA GENERAL D'UN SYSTEME DE COMPILATION



1.2. ORGANISATION DU SYSTEME DE COMPILATION CIVA

1.2.1. Système d'exploitation

Il est tentant, lorsqu'on réalise un ensemble relativement complet de services, de s'affranchir d'un système d'exploitation ; en effet, si les moniteurs de traitement par train ou les moniteurs élémentaires laissent une grande liberté de manoeuvre au programmeur, les systèmes plus évolués, tout en leur apportant une gamme plus élevée de services, lui amènent des restrictions.

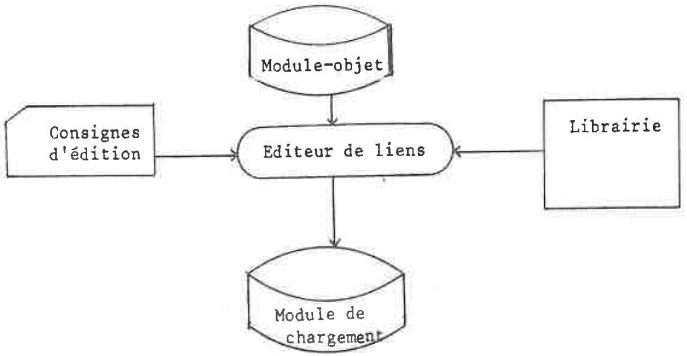
Civa doit être utilisable pour des exploitations de gestion supposant des installations importantes. Il paraît donc inconcevable de prévoir, au moins à long terme, de ne pas travailler sous un moniteur puissant (exploité en multiprogrammation, par exemple). Ceci est vrai, non seulement pour utiliser Civa, mais pour que sur un même site puissent cohabiter d'autres chaînes de traitement n'utilisant pas Civa. Or, un changement de système fait perdre un temps précieux ; donc, si nous voulions réécrire un système propre à Civa, il faudrait qu'il accepte les services déjà existants, ce qui est vraiment une trop grande tâche. Nous travaillerons sous un moniteur particulier (BPM et SIRIS 7 dans notre première application), et, chaque fois que le système imposera une limitation immédiate, on s'attachera plus à contourner la difficulté, même au prix d'un léger ralentissement de l'application, plutôt que d'essayer de modifier un moniteur, manoeuvre qui se révèle le plus souvent catastrophique. Remarquons toutefois qu'en acceptant un moniteur, nous sommes quand même libres de redéfinir n'importe quel processeur en vue d'un usage "Civa".

Nous allons donc simplement définir un système de compilation Civa s'appuyant sur un système d'exploitation existant. Ce système sera celui de l'installation dont nous disposons : SIRIS 7.

1.2.2. Choix d'un système de traduction

Le langage Civa est modulaire ; il est bon pour des questions de gain de temps que des modules écrits séparément voient le résultat de leurs compilations conservé séparément. Ceci permet alors d'insérer un module déjà compilé dans plusieurs chaînes de traitement, de ne recompiler qu'un seul module, si l'on ne modifie que celui-ci dans une chaîne de traitement ; enfin, le seul lien direct entre modules étant leur nom, il est logique de les traduire séparément. Cela impose pour l'exécution de rétablir les liens entre modules ; ceci peut être fait pendant ou avant l'exécution. C'est cette dernière solution que nous avons choisie. Nous aurons donc recours à un éditeur de liens.

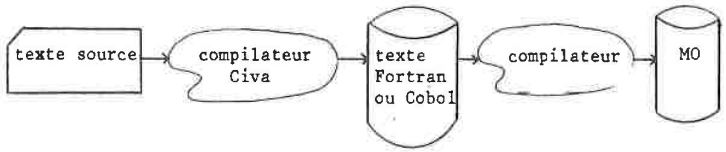
Celui-ci pourrait être propre à Civa, mais il doit être compatible avec les modules-objets déjà existants, de manière à pouvoir réutiliser des modules écrits en Cobol ou en Fortran. L'éditeur peut se définir entièrement par ses entrées et ses sorties :



Il semble donc inutile d'en réécrire un, il suffit de générer des modules-objets et de donner des consignes d'édition dans le format demandé par l'éditeur. Pour définir le système Civa, il nous suffit de définir un mode de production de modules-objets ; trois solutions sont possibles.

- génération de langage évolué, puis utilisation du compilateur correspondant pour générer du module-objet
- génération de Métasymbol
- génération directe de module-objet

1.2.2.1. Langage évolué intermédiaire



L'avantage principal de cette solution est la transportabilité du produit compilé ; le compilateur n'est qu'un générateur Cobol ou Fortran. Un avantage secondaire est une relative simplicité d'écriture : on peut laisser une bonne partie du travail au compilateur intermédiaire et se contenter d'un traitement de chaînes de caractères. (Le problème décisif est le choix du langage intermédiaire).

Fortran présente des avantages au niveau de la modularité ; la définition des variables structurées est acrobatique mais réalisable (par des "EQUIVALENCE") sur des éléments de type entier ou réel. La manipulation des chaînes de caractères devient très vite extrêmement lourde (à moins d'uti-

liser un mot pour stocker un caractère !). Le calcul sur des nombres décimaux n'est pas admis.

Cobol présente des avantages certains dans les traitements de gestion pure, est d'une modularité limitée et connaît des restrictions du point de vue transportabilité.

En fin de compte, aucun langage courant ne paraît pleinement satisfaisant en tant qu'intermédiaire ; cette solution a deux inconvénients supplémentaires : le premier est un allongement notable du temps de compilation ; le second, plus important, est la production d'un texte binaire qui sera assez loin du texte original : il en résulte une perte d'efficacité assez forte à l'exécution. Cette solution a donc été rapidement rejetée.

1.2.2.2. Production d'un texte intermédiaire en Métasymbol

L'utilisateur d'un Méta-assembleur semble présenter les mêmes aspects que la solution précédente (sauf la transportabilité). En fait, elle est beaucoup plus puissante si l'on utilise toutes les possibilités d'un méta-assembleur. Nous allons donner un exemple de ces possibilités en traduisant un module avec l'application de deux principes simples :

- tout symbole Civa produit, en langage intermédiaire, une méta-liste : celle-ci contient des codes identifiant l'élément (variable, entier, file, structure, etc.) et des constantes définissant l'emplacement ;
- les instructions Civa se traduisent par l'appel de procédure métasymbol qui génèrent un texte correct en fonction des paramètres.

Exemple :

Etant donné

```
module a ;
y struct (b entier, c file (5) car) ;
d file (5) car ;
d = c ;
fin
```

on pourrait définir l'équivalent Métasymbol :

```
     DEF         A
     MODULE      A                    MODULE est une procédure qui gé-
                                      nère toutes les initialisations :
                                      sauvegarde des registres de liai-
                                      son, par exemple

     CSECT       0                    ouverture d'une section en protec-
                                      tection 0 qui contiendra les va-
                                      riables et les mémoires de travail

Y    EQU         STRUCT, BA(ϕ), 9     Y est défini comme une liste,
                                      STRUCT est une métavariable défi-
                                      nissant le type "structure" ;
                                      BA(ϕ) représente l'implantation de
                                      Y ; 9, la longueur de la structure
                                      en octets

B    EQU         VAR, ENTIER, ϕ       VAR et ENTIER sont deux codes défi-
                                      nissant B comme variable entière,
                                      d'emplacement ϕ

     RES         1                    réservation d'un emplacement
                                      pour B

C    EQU         FILE, CAR, BA(ϕ), 5  C de même est défini comme une file
                                      de caractères de 5 octets
     RES, 1      5

D    EQU         FILE, CAR, BA(ϕ), 5  D est défini de manière identique
     RES, 1      5

     AFFECTATION D, C

     FINMØDULE                        procédure analogue à MODULE

     END
```

Ce programme serait en fait précédé d'une série de définition de type :

```
STRUCT      EQU   1
FILE        EQU   2
ENTIER      EQU   1
CAR         EQU   2
        ...
```

AFFECTATIØN serait une procédure déclarée, elle aussi, avant le programme et qui pourrait contenir les instructions :

```
AFFECTATION  CNAME
             PRØC
             LØCAL EMETTEUR, RECEPTEUR
EMETTEUR     EQU   AF(2)
RECEPTEUR    EQU   AF(1)
             ...
             DØ    EMETTEUR (1) = RECEPTEUR (1)
             ...
             DØ    EMETTEUR (1) = FILE
             ...
             GØTØ, EMETTEUR (2)   TRAITCAR, TRAIENT, ...
             ...
TRAITCAR     GØTØ, RECEPTEUR (2)  TRAITCARCAR, ...
             ...
TRAITCARCAR  DØ    EMETTEUR (4) = RECEPTEUR (4)
             LI,2  EMETTEUR (4)
             LI,3  EMETTEUR (3)          génération d'un
             STB,2      3                transfert de
             LI,2  EMETTEUR (3)          chaîne d'octets
             MBS,2      0
             ELSE
             ...
             FIN
             ELSE
             ...
             FIN
             ...
             ELSE
             ...
             FIN
             ...
             PEND
```


La méthode que nous avons présentée ci-dessus a un inconvénient : la manipulation de variables de nom STRUCT, FILE, CAR ..., devient interdite ; on domine cette difficulté en remplaçant :

```
      B      EQU          VAR, ENTIER, ♦
```

par :

```
      B      TYPE         VAR, ENTIER
```

où TYPE est une procédure telle que :

```
      TYPE   PRØC
             CNAME
             LØCAL  ETI1, ETI2
      ETI1   SET    SCØR (AF(1), VAR, FILE, STRUCT, ...)
      ETI2   SET    SCØR (AF(2), ENTIER, CAR, ...)
      LF(1)  EQU    ETI1, ETI2, ♦
             ...
             PEND
```

1.2.2.3. Comparaison entre la génération directe de code-objet et la production d'un texte intermédiaire en Métasymbol

D'après l'exemple précédent, nous pouvons imaginer la simplicité du processeur assurant la traduction de Civa en Métasymbol. La seule difficulté est la rédaction des procédures, et nous avons passé sous silence, dans l'exemple précédent, certaines difficultés techniques telles que :

- Apparition de symboles de même nom

```
module  a ;
y struct (a! entier, b! entier) ;
z struct (a! entier, b! entier  ;
```

traduire y = z ne présente aucune difficulté, mais

al *de* y = al *de* z ;

nous oblige à modifier nos définitions simples de symboles. Cette difficulté se résoud en numérotant les structures : par exemple, "al *de* y" devient !A1 et al *de* z, 2A1 ; mais ceci nous oblige à recréer une mini-table de symboles, ce dont nous nous étions complètement affranchis.

- Traduction d'expression arithmétique

y = a * b + c * d ;

pourrait être traduite par :

```
TRADEXP    Y, (PLUS, (PRØD, A, B), (PRØD, C, D))
           (où TRADEXP est une gigantesque procédure)
```

ou par

```
PRØD, !T1    A, B
PRØD, !T2    C, D
SØM, Y       !T1, !T2
```

où !T1, !T2 sont des zones de travail ; PRØD, SØM, des procédures générant de simples opérations. Finalement, la compilation est simple, mais le passage intermédiaire fait appel à des procédures Métasymbol relativement complexes.

Nous aurons donc deux problèmes importants.

Le temps de Méta-assemblage est très long par suite d'une multitude d'appels de procédures. Or, Métasymbol, outil prévu pour la rédaction d'éléments de systèmes ou de compilateurs, s'avère d'un emploi très coûteux : la simple compilation d'un programme comportant simplement six appels de procédures (chacune d'elles faisant elle-même un autre appel) demande

0.5 minutes dont 0.4 de traitement CPU ; encore s'agit-il de procédures dix fois moins complexes que celles utilisables dans le compilateur défini ci-dessus. Le temps complet de compilation du moindre module Civa demanderait donc au moins une minute. A titre d'exemple, la compilation d'un texte Fortran de deux cents instructions de complexité moyenne demande 0.2 minutes. Un rapport deux ou trois semble tolérable au niveau de la compilation, mais un rapport dépassant dix rendrait totalement inutilisable le compilateur Civa.

De plus, le produit généré par une série de procédures peut s'avérer, lui aussi, peu performant (par exemple au niveau de la traduction des expressions arithmétiques). Lorsqu'on cherche à améliorer cet aspect, on constate qu'il faut conserver une table des symboles et se rapprocher progressivement d'un processeur qui génère directement des instructions élémentaires. Il semble alors plus simple de générer directement ces instructions dans des modules-objets. Cependant, si pour cette application (compilateur Civa) nous rejetons la solution "langage intermédiaire Métasymbol", celle-ci nous paraît très intéressante lorsque l'on désire implanter un langage à peu de frais : pour évaluer ses performances, par exemple, en ayant la possibilité de modifier de façon simple le produit final en travaillant au niveau des méta-procédures.

1.2.3. Description des modules-objets sur IO 070

Sous les systèmes d'exploitation BPM IO 070 ou SIRIS 7, tous les processeurs de traduction (Assembleur, Métasymbol, compilateur Fortran ou Cobol)

génèrent, à partir d'un texte source, des modules indépendants aux références et définitions externes près, appelés "modules-objets" (ou R.O.M. : Relocatable Objet Module). Ces "modules-objets" constituent, en fait, de véritables "programmes de chargement" comportant déclarations, définitions et instructions. Ils sont interprétés par l'éditeur de liens qui produit un "Load Module" (Module de chargement translatable) ; celui-ci est un processeur à deux passages. Schématiquement, on peut dire que le premier sert à comptabiliser l'espace requis par les divers modules-objets et à établir certaines tables. Ceci étant fait, le deuxième passage interprète les instructions des divers modules-objets pour créer les éléments du module de chargement. Nous allons décrire ces modules-objets en nous intéressant principalement aux problèmes de section et en donnant des exemples sur les autres points. (On trouvera une description complète dans le manuel "Normes de programmation SIRIS 7").

1.2.3.1. Structure logique d'un module-objet

Un module-objet est contitué de "rubriques de chargement" ; ces rubriques sont de plusieurs types : déclaration, définition, instruction de chargement, évaluation d'expression... Elles ont deux rôles : définir les liens existants entre les divers modules, assurer la génération d'éléments de module de chargement. Ces rubriques sont pratiquement identiques sous B.P.M. ou SIRIS 7.

1.2.3.2. Structure physique d'un module-objet sous BPM

Chaque rubrique est une chaîne d'octets précédée d'un octet de contrôle ;

celui-ci permet d'identifier la rubrique, mais, pour certaines, il peut contenir des informations supplémentaires. Un module-objet sera donc une chaîne d'octets. Cette chaîne est divisée en enregistrements, dont la taille est inférieure ou égale à 104 octets, auxquels on adjoint un "mot de contrôle".

Le découpage en enregistrements est purement arbitraire ; en particulier une rubrique de chargement peut continuer d'un enregistrement sur le suivant. Le mot de contrôle se compose de quatre octets :

- Le premier précise si cet enregistrement est ou non le dernier ;
- Le second indique le numéro de séquence de l'enregistrement ;
- Le troisième est une clé de parité longitudinale (Checksum) obtenue par calcul. (Il y aura donc en fait deux protections sur un module-objet : la parité transversale sur neuf bits et cet octet ; celui-ci est particulièrement utile dans le cas de fichiers stockés sur disque). En effet, par suite d'erreur, un fichier peut écraser certains enregistrements d'un autre fichier (sans introduire une parité fausse sur neuf bits, évidemment), mais on ne retrouvera plus la parité longitudinale ; ceci permettra de détecter l'anomalie) ;
- Le dernier octet indique la taille utile de l'enregistrement.

1.2.3.3. Structure physique sous SIRIS 7

La chaîne d'octets est de la même manière découpée en enregistrements. Le système n'accepte pas le mot de contrôle d'enregistrement, mais il assure complètement la gestion des enregistrements logiques.

1.2.3.4. Rubriques de chargement

1.2.3.4.1. Rubriques de déclaration

On trouvera deux types de déclaration : les déclarations de liens et les déclarations de section.

a) Déclaration de liens

On entend par "lien" une définition ou une référence externe. Ces liens sont repérés par le système sous la forme d'une chaîne de caractères. La déclaration d'un lien donnera donc une rubrique du type

[code] [longueur] [nom]

où code est X'03' pour une définition externe et X'05' pour une déclaration externe ; "longueur" est la longueur de la chaîne de caractères "nom".

Exemple :

module truc	*module* aa
...	
aa ;	...
...	
fin module	*fin module*

Dans aa, on trouvera :

X'03', X'02', X'C1C1' (C1 est le code EBCDIC de A)

Dans truc, on trouvera :

X'05', X'02', X'C1C1'

b) Déclaration de sections

On trouvera trois types de sections : des sections fictives, des sections de contrôle standard et non standard. Une telle déclaration comporte deux renseignements : le mode d'accès à la section, et la taille en octets de celle-ci.

La différence entre une section de contrôle standard et une section de contrôle non standard est très simple sous BPM. Une section standard, si elle existe, est la seule section de contrôle, et sa déclaration se fait lorsqu'on y a terminé toutes les opérations de générations. Par contraste, on peut disposer de plusieurs sections non stadard, mais leurs déclarations doivent alors précéder toute instruction de génération dans celles-ci. Il est donc clair que tout processeur voulant générer plusieurs sections, pour un même module, devra utiliser des sections non standard (et donc, en règle générale, avoir au moins deux passages).

Sous SIRIS 7, on retrouve les mêmes types de sections (standard et non standard), mais on autorise la déclaration de la taille de la section après les instructions de génération ; les différences sont de ce fait beaucoup moins nettes, et il n'est pas improbable qu'une question de compatibilité entre BPM et SIRIS 7 ait entraîné cette distinction douteuse.

Fortran IV IO 070 génère par exemple deux sections non standard (une pour la zone code constante, en protection écriture et l'autre en zone variable). L'assembleur symbol génère une section standard par programme.

c) Numéro de déclaration

Lors des instructions de générations, il va falloir faire référence à certaines déclarations. Si nous voulons générer LW,8 A+1, A étant une référence externe, il faudra indiquer que la partie adresse de l'instruction est définie par rapport à une référence externe ; le problème est rigoureusement le même si A est définie dans une section quelconque du module. Pour cela, on associe à chaque déclaration un "numéro de déclaration" ; le numéro 0 est affecté à la section de contrôle standard, les autres déclarations portent comme numéro leur ordre d'apparition.

Exemple :

```
      CSECT    0
A     RES      1
      REF      B
      CSECT    0
C     RES      1
D     EQU      B+1
```

A est repéré par rapport à la déclaration 1 (première CSECT), B et D par rapport à 2, C par rapport à 3.

d) Cas des références en avant

L'éditeur de liens BPM ou SIRIS 7 accepte les références en avant. On aura donc le même problème de repérage de celles-ci dans un module-objet. Il n'y a, en fait, pas de déclaration explicite, la première référence à une "référence en avant" constitue sa déclaration ; pour cela, le compilateur attribue à cette référence un "numéro-de-référence-en-avant" purement arbitraire. Il utilisera celui-ci pour une déclaration ultérieure.

e) Définition d'un emplacement

Un emplacement pourrait être simplement défini à l'exécution par une adresse et une taille, par exemple, le mot (ou 4 octets) qui se trouve à l'adresse X'3000'. Lorsqu'on parlera de module-objet, un emplacement sera défini par

- son type : relatif à une déclaration
  relatif à une référence en avant ;
- son numéro de déclaration ou de référence en avant ;
- sa translation par rapport à cette déclaration ou son adresse relative ; cet adressage pourrait être défini en octet. En fait, nous serons obligés de manipuler des adresses de mot, d'octet, etc. Une translation est en fait un doublet (adresse, type de résolution).;
- sa taille en octets

Exemple :

```
      CSECT   0
A     TEXT    'A B C D E F G H I'
```

La chaîne de caractères 'CDE' est définie par

type	déclaration
numéro	1
translation	2
résolution	octet
taille	3 octets

1.2.3.4.2. Instructions de définition, évaluation d'expression

Les références en avant, les définitions externes doivent être définies ; les directives correspondantes sont de la forme :

< octet de contrôle > < numéro de nom ou de référence> < expression >

où

< expression > ::= <élément d'expression> * ,< fin d'expression>

et

< élément d'expression >::= <addition constante>|< addition référence en avant > | <addition numéro déclaration > ...

Le principe de calcul d'une expression est simple : un accumulateur est initialisé à 0, puis l'éditeur effectue les opérations indiquées.

Exemple :

```
        DEF     A
        CSECT   0
        RES     3
A       EQU     $
        ...
```

produira :

déclaration de A :

03	01	C1
déclaration de DEF	1 octet	code EBCDIC A

déclaration de section :

0C	000000C
déclaration de section	taille = 12 octets

définition de A :

0A	0001
définition de DEF	de numéro 1

ajouter la constante 12(octets) :

01	0000000C
addition constante	12

ajouter la déclaration 2 (section de contrôle) :

20	02
ajouter la déclaration déclaration	de numéro 2

fin d'expression : 02

02

...

1.2.3.4.3. Instruction de chargement

Il existe trois instructions de chargement : définition d'origine, chargement absolu et chargement translatable.

- La définition d'origine :

  (analogue aux déclarations précédentes) fixe la valeur du compteur d'emplacement utilisé par l'éditeur.

- Le chargement absolu :

demande le chargement d'une chaîne d'octets, celle-ci ne devant subir aucune modification, quelle que soit la translation du module de chargement. Cette instruction est de la forme

[octet de contrôle] [chaîne d'octets]

[octet de contrôle] = [0100] [nombre d'octets $0 \leqslant n \leqslant 16$]

Exemple : LI,4 3

serait traduit par

44 22400003

octet de contrôle

- Le chargement translatable :

cette rubrique sera utilisée pour charger des mots dont les 17 derniers bits contiennent une adresse translatable à résolution par mot, octet, dernier mot... Elle se présente sous deux formes : longue ou courte ; la forme courte s'applique dans les cas simples (résolution d'adresse par mot, numéro de déclaration < 64). L'octet de contrôle contient alors le numéro correspondant. Dans les autres cas, on utilisera une forme longue (qui revient, en fait, à ajouter 1 octet à l'octet de contrôle).

1.2.3.4.4. Exemple récapitulatif

Nous avons cherché, dans cet exemple, à donner une image relativement complète des divers éléments rencontrés dans un module-objet. Le programme source est en Métasymbol.

programme source			module-objet (en hexadécimal)	commentaires
	CSECT	0	0C	déclaration de section non standard
			00001C	elle possède 7 mots (X'1C'octets)
				son numéro de déclaration est 1
	DEF	AA	05 02C1C1	déclaration de DEF nom : AA
				son numéro de déclaration est 2
	REF	BB	03 02 C2 C2	même chose pour BB, réf. dont le numéro de déclaration est 3
	RES	2		
AA	EQU	s	0A 0002 0100 00 00 08 2001 02	définition de DEF de numéro 02 de valeur 8 octets à partir de la déclaration 1 fin d'expression
	RES	1		
	LI,3	1	04 2001 010000000C 02 44 22 30 00 01	définition d'origine de chargement déclaration n° 1 plus 12 octets générer les 4 octets absolus 22 (LI) 300001

STW,3	C	C9	charger le mot suivant, l'adresse étant définie par rapport à la référence en avant numéro 9 (arbitraire)
		3530 00 00	
LW,8	BB+2	83	charger le mot suivant, l'adresse étant définie par rapport à la déclaration 3
		32 80 00 02	
C RES	1	08	définition de la référence en avant
		0009	de numéro 9
		0100000018	24ème octet (X'18')
		2001	de la section 1
		02	fin d'expression
END	AA	OD	définition de l'adresse de lancement :
		20 02	déclaration numéro 2
		02	fin d'expression
		OE	fin de module-objet
		00	niveau de sévérité : 0

1.2.4. Utilisation des sections en Civa

Nous avons vu qu'il est très important de disposer de zones mémoires de protection d'accès différente, ceci pour des raisons simples de "sécurité" : il est beaucoup plus facile d'analyser une erreur lorsqu'elle se produit que lorsqu'elle a altéré la suite d'instructions, rendant un Dump mémoire complètement inefficace. Ceci semble donc imposer la présence de deux sections pour un module. Cette solution présente un inconvénient si l'on veut écrire un compilateur compatible BPM et SIRIS 7 ; en effet, sous BPM, la présence de deux sections (donc non standard) impose la connaissance de la taille exacte de chaque section avant d'entamer le processus de génération. Nous aurions donc un compilateur à au moins deux passages, sinon trois :

Passe 1 : à partir du texte source, production d'une chaîne codée dans laquelle les problèmes de reconnaissance d'instructions source et d'adressage sont résolus.

Passe 2 : à partir de la chaîne codée, production de code objet stocké sur un fichier de travail.

Passe 3 : génération finale de code objet : déclaration des sections puis recopie du fichier intermédiaire.

La passe 3 semble particulièrement lourde par rapport au travail qu'elle fournit réellement.

De plus, si nous examinons le "contenu" de ces sections, l'une (celle qui contient les codes instructions et les constantes) est complètement définie

par le compilateur ; l'autre ne contient qu'une simple réservation, aucune variable n'étant initialisée, lors de sa déclaration, en Civa. Il semble donc aisé de créer deux modules-objets, l'un après l'autre ; le premier est généré pendant la passe 2 et contient simplement la section en protection écriture, le second est généré ultérieurement et n'a besoin que d'un seul renseignement : la taille de la zone est variable.

Il reste cependant un problème : lors de la génération des instructions, on devra repérer les éléments de la zone variable. Il faut donc associer à la zone variable une définition externe. Nous verrons qu'à l'exécution un module en appelle un autre, grâce à une définition externe. Toute confusion de nom doit être impossible. Nous avons pris un moyen simple : si NØM est un nom de module, "NØM" repérera la zone en protection écriture et $ concaténé à NØM donc "$NØM" repérera la zone variable.

Exemple :

```
module  a ;
entier  x, y ;
x = y
fin module
```

produira : (nous allons représenter les modules-objets par leur équivalent en Métasymbol)

MODULE OBJET 1

```
     CSECT    1
     DEF      A
     REF      $ A
A    EQU      $
X    EQU      $ A
Y    EQU      $ A+1
     LW, 1    Y
     STW,1    X
     END
```

MODULE OBJET 2

```
     CSECT    0
     DEF      $ A
$ A  EQU      $
     RES      2
     END
```


Introduction de la préédition de liens [2]

Au lieu de générer séparément chaque module-objet de type "zone variable", il nous a paru beaucoup plus intéressant de prévoir un processeur qui génère la zone variable complète pour tous les modules dépendant d'un module directeur. On peut alors optimiser la répartition mémoire sans faire de recouvrement dynamique en zone variable. Ceci est fait par le préditeur de liens et, grâce à lui, même sous SIRIS 7, nous générons séparément les deux zones pour un seul module source.

1.2.5. Liaisons entre modules, classes

1.2.5.1. Relation entre modules

Un module doit pouvoir en appeler un autre ; deux modules sont compilés séparément et il doit donc exister un lien entre les deux : ce lien sera le nom du module appelé.

Pour chaque module existera une définition externe : le nom de ce module et autant de références externes que de modules différents appelés. Le problème des transmissions de paramètres sera étudié ultérieurement.

L'appel proprement dit sera réalisé à l'aide d'une instruction BAL ; par compatibilité avec les autres compilateurs existants, le registre de liaison sera le registre 15. Pour permettre un nombre quelconque de niveaux d'appel, chaque module préservera le contenu du registre de liaison dans une mémoire de travail :

Exemple :

module a

...

b ;

fin module

module b

...

fin module

on trouvera dans a

```
REF       B
BAL, 15   B
```

dans b

```
          DEF       B
          CSECT     1
B         EQU       s
          STW,15    α
          ...

          B         * α
```

où α est une mémoire temporaire

1.2.5.2. Relation entre modules et classes

L'aspect essentiel de la relation entre modules ou classes est la possibilité, dans un module, de repérer n'importe quel élément d'une classe du type : variable, structure, constante, procédure. Pour traduire une classe, plusieurs solutions sont possibles.

On peut penser déclarer chaque élément comme définition externe : ceci entraîne deux inconvénients :

- impossibilité de traduire "*de*" car cette notion n'existe pas au niveau

des définitions externes :

classe a

entier b

classe c

entier b

module truc

utilise a ;

x = b *de* a ;

- lourdeur de l'édition de liens et des diverses compilations comportant un nombre très élevé de liens.

Une autre solution consiste à recopier le texte des classes dans chaque module Civa et le recompiler à chaque fois (en traduisant tout ce qui concerne une classe par une section fictive). Une solution analogue est utilisée par Métasymbol avec SYSTEM, par Cobol avec CØPY. Cette solution a contre elle de multiplier, par un facteur considérable (deux, trois ou plus), le temps de compilation d'un module et ceci pour presque rien. On peut très bien imaginer qu'un module n'utilise qu'une très petite partie de la classe ; en particulier, la recompilation des procédures devient complètement absurde.

La seule chose qui importe pour un module utilisant une classe est l'implantation des éléments de cette classe (en dehors du type pour les variables). Il suffit donc que la classe transmette au module l'utilisant la liste de ses emplacements, autrement dit, sa table de symboles. Dans ce cas, une classe peut être constituée comme un module, la zone constante est repérée par une définition externe : le nom de la classe. Toute constante est repérée "de l'extérieur" par sa translation par rapport à cette définition externe ; de même, la zone variable sera repérée par une définition externe de type $NØM.

Exemple :

```
module a                    classe
utilise C1 ;                entier d, e ;
entier  x, y                procédure egal ;
e = x * y                   d = e  fin proc
egal ;                      fin classe
fin module
```

Dans le module a, on déclarera C1 et $C1 comme références externes et on utilisera la table des symboles de a ; par exemple, e sera repéré par $C1 + 1 et égal par C1 (egal est en début de zone constante).

Dans la classe C1, on déclarera C1 en définition externe et le prééditeur déclarera $C1. On trouvera des exemples plus précis à la fin de ce chapitre.

Une classe produit donc :

- un module-objet contenant sa zone constante repérée
  . par son nom, elle se repère à sa zone variable,
  . par $NØM, comme pour un module ;
- une table de symboles (chaque symbole est repéré par rapport à nom ou $NØM) ;
- un enregistrement dans le fichier descriptif à destination du prééditeur de liens.

Tout module qui utilise une classe

- si C est le nom de cette classe, déclare C et $C comme référence externe ;
- introduit la table des symboles de la classe dans sa propre table des symboles.

1.2.6. Schéma du système de compilation Civa

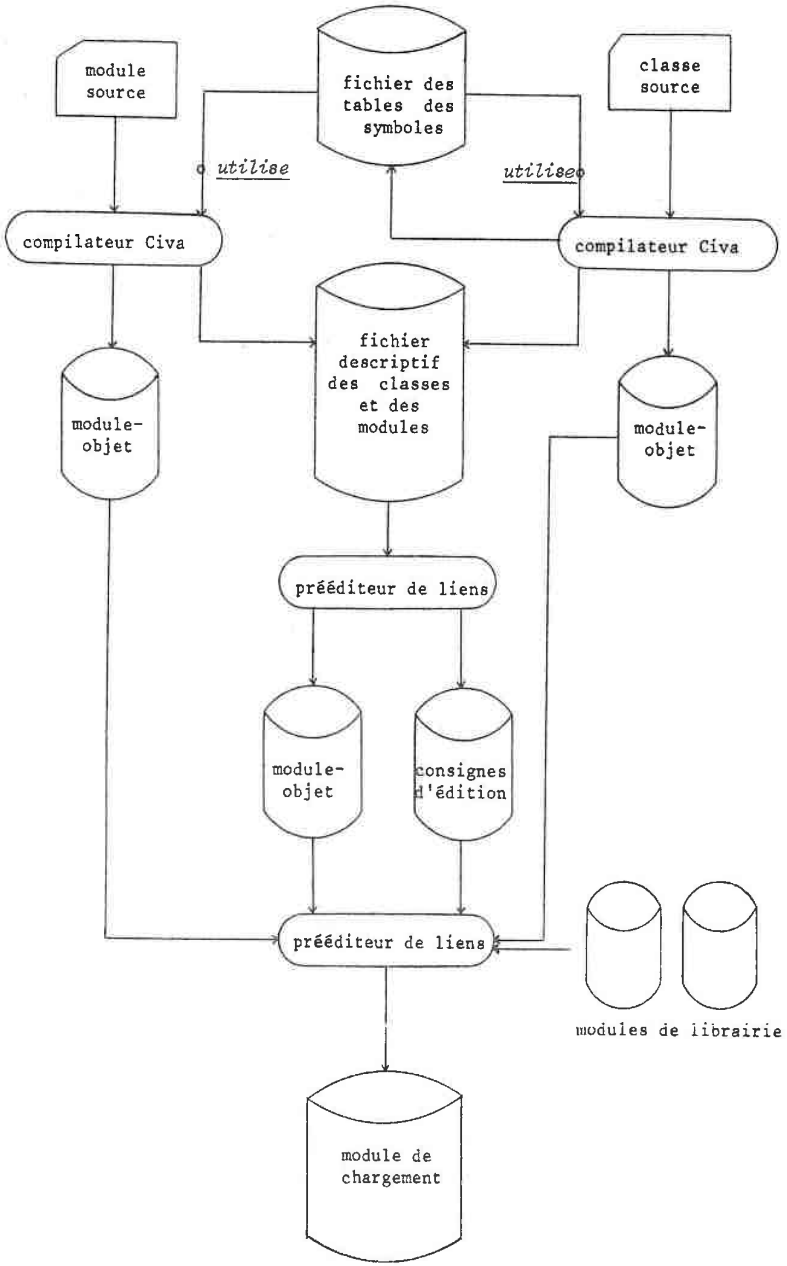
Comme dans tous les systèmes, nous aurons des modules-objets et des modules de chargement, ainsi qu'une librairie utilisable lors de l'édition de liens. Nous aurons, de plus, un fichier des tables des symboles des classes, véritable librairie au niveau de la compilation et un fichier descriptif de l'application permettant au prééditeur de liens de prévoir un recouvrement.

La compilation d'un module produira un module-objet et un enregistrement dans le fichier descriptif.

La compilation d'une classe produira un module-objet, un fichier table des symboles et un enregistrement dans le fichier descriptif.

La préédition de liens produit un module-objet contenant toutes les déclarations de type $NØM, ainsi que des consignes à l'éditeur de liens.

L'exécuteur est celui du système standard.



Exemple récapitulatif

Dans cet exemple, afin de simplifier les modules-objets, on n'a pas tenu compte systématiquement des sauvegardes de registre. Les modules-objets sont écrits en équivalent Métasymbol.

```
module m1 ;
utilise c1 ;
entier x ;
i = 1 ;
m2 ;
x = i ;
fin module ;
```

1

```
module m2 ;
utilise c2 ;
entier x ;
i de c1 = i de c2 ;
k = n ; egal ;
carré ; cube ;
x = t ;
fin module ;
```

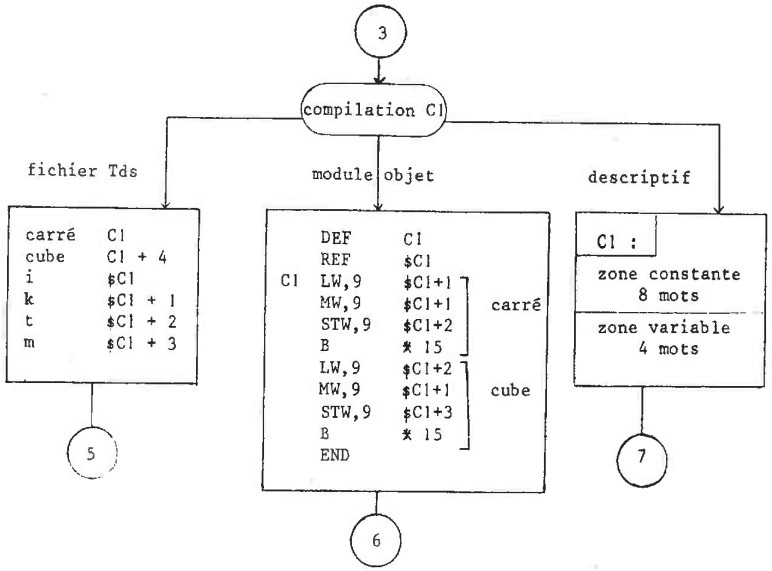
2

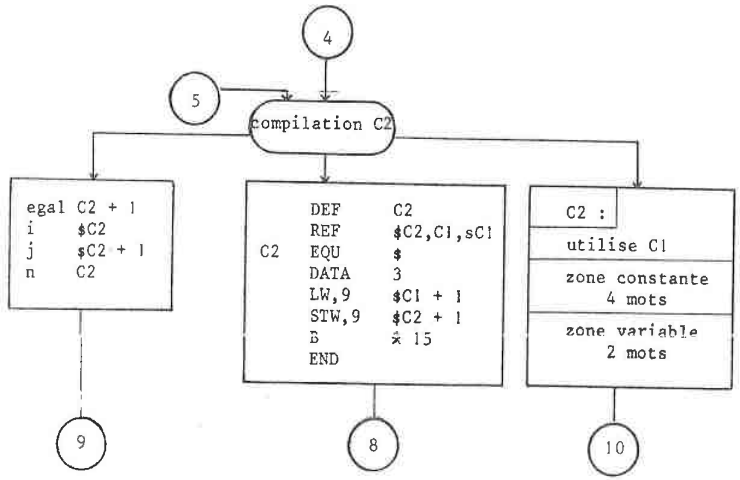
```
classe c1 ;
entier i, k, t, m ;
procedure carré ;
t = k * k ;
procedure cube ;
n = t * k ;
fin classe
```

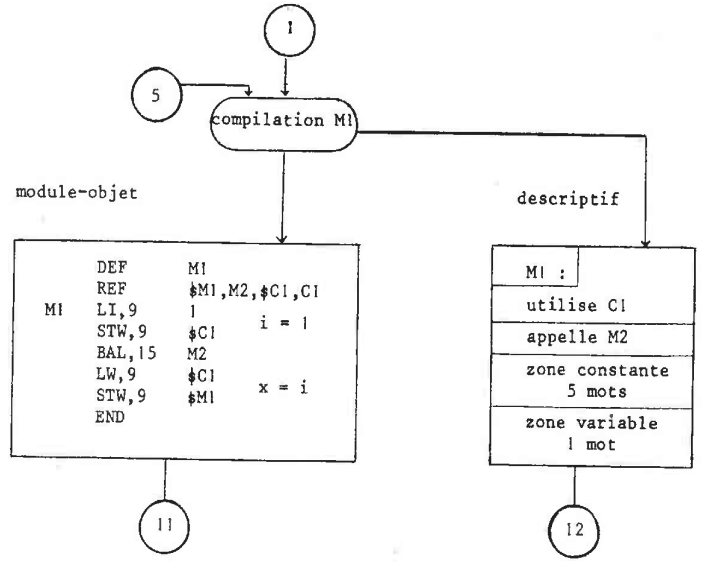
3

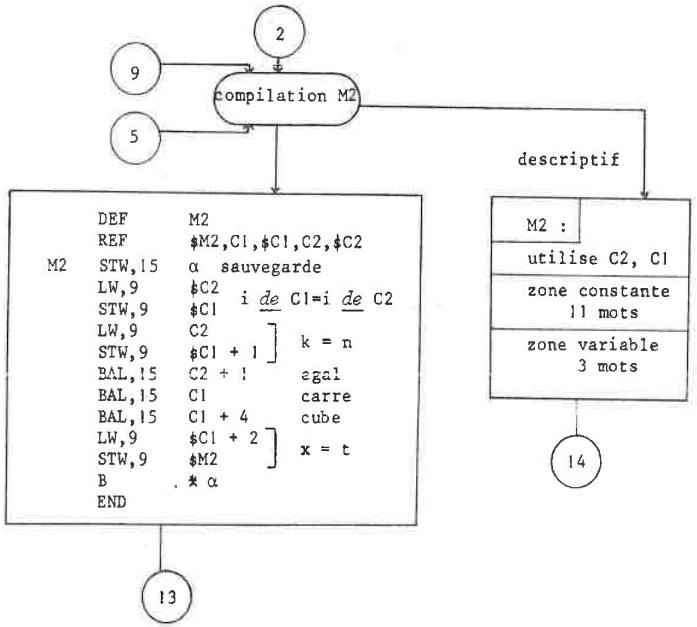
```
classe c2 ;
entier j, i ;
entier n = 3 ;
utilise c1 ;
procedure egal ;
j = k
fin classe
```

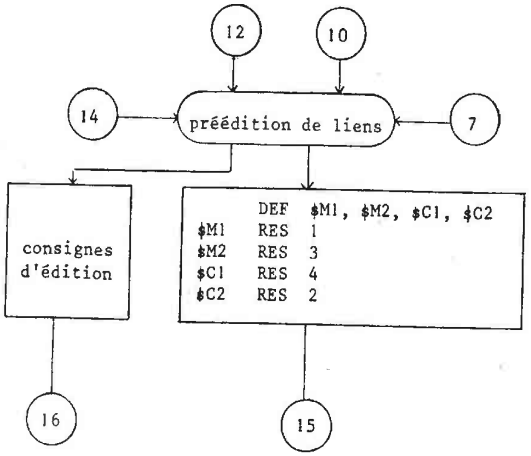
4

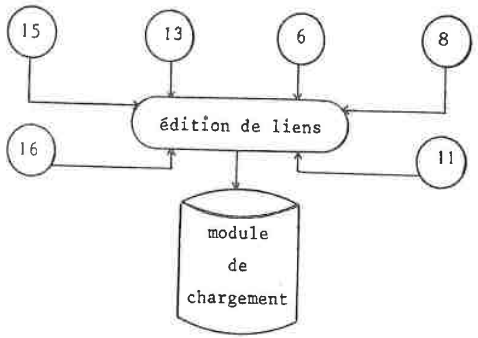

3
compilation C1
fichier Tds
module objet
descriptif
carré C1
cube C1 + 4
i $C1
k $C1 + 1
t $C1 + 2
m $C1 + 3
DEF C1
REF $C1
C1 LW,9 $C1+1
MW,9 $C1+1
STW,9 $C1+2
B * 15
carré
LW,9 $C1+2
MW,9 $C1+1
STW,9 $C1+3
B * 15
cube
END
C1 :
zone constante
8 mots
zone variable
4 mots
5
6
7


4
5
compilation C2
egal C2 + 1
i $C2
j $C2 + 1
n C2
DEF C2
REF $C2,C1,sC1
C2 EQU $
DATA 3
LW,9 $C1 + 1
STW,9 $C2 + 1
B * 15
END
C2 :
utilise C1
zone constante
4 mots
zone variable
2 mots
9
8
10


1
5
compilation M1
module-objet
descriptif
DEF M1
REF $M1,M2,$C1,C1
M1 LI,9 1 i = 1
STW,9 $C1
BAL,15 M2
LW,9 $C1
STW,9 $M1 x = i
END
M1 :
utilise C1
appelle M2
zone constante
5 mots
zone variable
1 mot
11
12


2
9
5
compilation M2
descriptif
DEF M2
REF $M2,C1,$C1,C2,$C2
M2 STW,15 α sauvegarde
LW,9 $C2 i de C1=i de C2
STW,9 $C1
LW,9 C2
STW,9 $C1 + 1
k = n
BAL,15 C2 + 1 egal
BAL,15 C1 carre
BAL,15 C1 + 4 cube
LW,9 $C1 + 2
STW,9 $M2
x = t
B * α
END
M2 :
utilise C2, C1
zone constante
11 mots
zone variable
3 mots
13
14





2
système de programmation permettant de décrire le compilateur civa

2.1 INTRODUCTION

Nous venons de décrire dans quel contexte se place le compilateur et quelles sont ses interfaces avec le système de compilation Civa. La réalisation de ce système constitue une application au sens de Civa, le compilateur en étant un sous ensemble ; pour cette opération, il nous a paru fondamental d'appliquer les mêmes règles que celles que nous indiquons dans la définition du langage Civa. Il était donc nécessaire de s'appuyer sur un langage permettant une description modulaire des actions et des informations.

Dans ce chapitre, nous allons définir les normes d'écriture de l'application. Le premier problème est donc le choix d'un langage, puis l'adaptation de ce langage à une utilisation Civa. L'application "réalisation Civa" sera donc constituée d'un ensemble de classes et de modules. Il restera à définir la tenue à jour de la documentation des textes de l'application, la gestion des bibliothèques de modules et de classes de l'application et les procédures de modification des textes de l'application.

2.2 CHOIX D'UN LANGAGE DE PROGRAMMATION

Sur un ordinateur relativement puissant, le choix de langage d'écriture est très vaste. Les diverses possibilités d'écriture existant sur IO 070 sont :

- langage évolué Fortran, Cobol, Algol,
- langage machine "universel" interprété,

- langage machine "universel" assemblé,
- langage d'assembleur 10070,
- langage d'assembleur perfectionné LP70,
- langage de méta-assembleur : Métasymbol.

2.2.1. Langage évolué

L'avantage essentiel de cette solution est l'universalité du produit obtenu. Elle présente cependant plusieurs inconvénients : un langage évolué est plus ou moins spécialisé dans un type de traitement, calcul scientifique ou gestion, et l'on est amené à faire beaucoup d'acrobaties pour l'adapter à un traitement tel que la compilation. (Ceci entraîne une perte d'universalité, les acrobaties étant souvent locales à une machine). De plus, le produit final est moyennement efficace tant du point de vue encombrement que de la rapidité, ce qui n'est guère acceptable pour un compilateur qui est le type de produit le plus souvent appelé sur une installation. Enfin, la plupart des machines proposent une série d'instructions spécialisées (manipulation de pile, traitement de chaîne d'octets) qui sont difficilement accessibles au niveau évolué. Nous devons avouer que l'élimination de cette solution aurait été plus délicate si nous avions disposé d'un langage tel que PL1.

2.2.3. Langage machine "universel" interprété

Les instructions machines se ressemblent étrangement d'un ordinateur à un autre. On pourrait donc penser écrire un software en utilisant un langage machine neutre, c'est-à-dire indépendant du calculateur ; un programme écrit

dans ce langage sera exécuté par un interprète qui représentera la machine virtuelle possédant ce langage. C'est le cas du compilateur FORTRAN IV sur le C.I.I. 10 070.

Cette solution semble, à première vue, très satisfaisante : en effet, lors-lorsqu'on change de calculateur, il suffit de réécrire l'interprète, travail qui, du point de vue programmation, est relativement simple à réaliser. Elle présente de gros inconvénients : le premier est qu'il ne suffit pas de réécrire l'interprète, car il faut, en fait, refaire toute l'interface avec le système ; le second est qu'il n'existe pas de "langage neutre" vraiment universel. (Une définition d'un langage machine universel a été tentée avec L.M.U., son écriture et son adaptation sur 10 070 étant assez lourdes, nous avons éliminé cette solution). Il sera pratiquement impossible à quiconque de faire une modification s'il ne connaît pas ce code, ceci est particulièrement vrai pour l'utilisateur de Civa (au niveau système), qui hésitera à tenter la moindre modification s'il faut d'abord approfondir le fonctionnement de la machine virtuelle. Le dernier inconvénient est la longueur à l'exécution du résultat car chaque instruction élémentaire devra être interprétée, ce qui représente trois opérations élémentaires au minimum :

- décodage de l'instruction,
- exécution proprement dite,
- passage à l'instruction suivante.

Pour toutes ces raisons, cette solution a été rejetée.

2.2.4. Langage machine "universel" assemblé

C'est une variante de la solution précédente qui consiste à éviter le peu d'efficacité de l'interprète en essayant de trouver, pour chaque instruction du langage neutre, une ou plusieurs instructions du langage source qui feraient le même travail. Mais, au lieu de réaliser cette équivalence à l'exécution, on la réalise à l'aide d'un méta-assembleur en redéfinissant les instructions du langage simple par des méta-procédures. Elle impose que toutes les installations sur lesquelles le compilateur sera implémenté possèdent un méta-assembleur de même efficacité. De plus, il est souvent difficile de remplacer chaque instruction d'une machine par une ou plusieurs instructions d'une autre machine.

Supposons, par exemple, que nous prenions comme machine virtuelle une machine possédant un registre de base, il sera difficile de la simuler sur une machine n'en possédant pas, sans perte d'efficacité, et réciproquement. Dans la mesure où l'application est rédigée de manière modulaire, les liaisons entre modules étant relativement indépendantes d'une machine particulière, on peut se demander s'il n'est pas plus simple de réécrire séparément chaque module dans la logique d'une machine, plutôt que d'essayer d'établir une équivalence entre les deux.

Nous pouvons enfin signaler à titre d'exemple qu'il a été rédigé un simulateur de 360 sur 10 070 et il s'est avéré plus simple d'interpréter le langage machine 360 et de redéfinir un assembleur que d'écrire un assembleur acceptant le langage symbolique 360 pour générer des instructions machine 10 070.

2.2.5. Assembleur 10 070

On peut envisager d'écrire le compilateur en utilisant uniquement le langage assembleur 10 070. Cette solution présente des avantages certains : elle permet d'utiliser au mieux les possibilités d'un calculateur, mais elle est lourde à mettre en oeuvre et complètement intransportable. En effet, chaque instruction repère une action élémentaire et il faudra, pour exprimer une action simple se présentant dans des contextes légèrement différents, écrire un grand nombre d'instructions. La lecture d'un programme est peu commode, les bouclages, tests, sous programmes n'apparaissant qu'après une étude de toutes les instructions d'un programme. Cette solution n'est donc à envisager que lorsque c'est la seule possible.

2.2.6. Assembleur évolué du type LP70

Sa caractéristique est de proposer les mêmes avantages que l'écriture directe en assembleur (performance du produit à l'exécution), tout en ayant une écriture beaucoup plus facile et beaucoup plus lisible. Il a d'ailleurs été montré que la rédaction était à peu près deux fois plus rapide en LP70 qu'en assembleur. C'est une solution de ce type que nous aurions retenue si nous n'avions pas disposé d'un outil tel que Métasymbol.

2.2.7. Méta-assembleur : Métasymbol

Nous allons présenter sommairement les principales caractéristiques de ce langage. Il présente tous les avantages de LP70 au point de vue efficacité

possible du produit final. L'utilisation de Métasymbol suppose une préparation relativement longue pour définir les "méta-instructions de base" d'une application. C'est la raison pour laquelle on peut lui préférer LP70 ; dans les applications de taille moyenne, ce langage est beaucoup plus facile à assimiler. Cependant, pour une application importante, à condition que l'on passe un temps assez long pour définir un système de base, on aboutira à un résultat beaucoup plus modulaire et d'emploi aussi facile. Il permettra surtout des modifications extrêmement simples : n'importe quelle action ou information pouvant être décrite par une procédure et repérée uniquement par son nom, il suffit de modifier le texte d'une méta-procédure pour qu'un assemblage la répercuter sur tout le programme ; ceci permet également de faciliter l'adaptation du produit résultant à n'importe quelle installation, en modifiant simplement quelques méta-variables.

2.3. PRESENTATION DE METASYMBOL

2.3.1. Rappel sur la définition d'un méta-langage

2.3.1.1. Macro-génération

De façon très générale, un macro-processeur permet de créer un texte objet quelconque à partir de substitution dans des chaînes alphanumériques et de macro-procédures définies par l'utilisateur. Le modèle le plus simple de macro-processeur admet deux instructions : définition d'une macro-procédure et appel d'une macro-procédure. La rencontre d'une macro-instruction est, en général, caractérisée par l'apparition d'un caractère spécial :

Exemple :

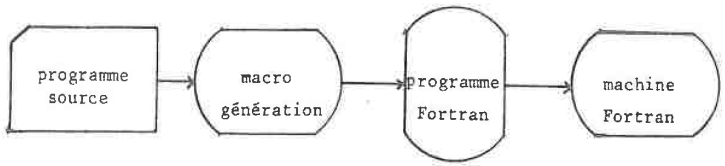
$MACRØ nom : chaîne de caractères $FIN constitue la définition d'une macro-procédure et $ (nom $) son appel. Ce modèle a un grand intérêt car, utilisé dans un langage de programmation, il permet de définir une seule fois des séquences répétitives. Utilisé avec le Fortran 10 070, on pourrait écrire :

```
$MACRØ TEST : IF (I.EQ.5) ØUTPUT K, L $ FIN ;
```

une ligne contenant $ (TEST $) serait traduite par le générateur ainsi :

```
IF (I.EQ.5) ØUTPUT K, L
```

Un programme écrit dans ce langage peut être considéré comme étant destiné successivement à deux machines : la macro-machine qui produit un texte Fortran (par exemple), et la machine Fortran qui exécute le programme résultant.



Ce dispositif peut être amélioré facilement en introduisant des paramètres, (les procédures cataloguées se comportent comme des macro-procédures sur les langages de commande).

2.3.1.2. Méta-générateur

Si, au liéu de faire du macro-générateur une simple machine à substitution, nous lui ajoutons toutes les instructions classiques d'une machine programmable, affectation, test, bouclage, nous obtiendrons alors un générateur beaucoup plus puissant que nous appellerons méta-générateur. Nous aurons alors deux véritables machines travaillant l'une après l'autre. Il sera alors possible de faire des calculs, d'analyser finement les appels de méta-procédures et de générer conditionnellement un texte différent adapté à chaque appel. Si nous reprenons l'exemple d'un méta-générateur produisant du Fortran, il est possible d'écrire, si "$ = X : cte" représente l'affectation de cte à la méta-variable X :

```
      $ = N : 1
      $META TEST :
      $SI N=1 $ALØRS
100   FØRMAT (x x x x)
      $FSI
      $=N : 0
      IF(...) WRITE (108, 100) ...
      $FIN
```

Le premier appel de TEST génère la définition du format 100, ce que ne font plus les appels suivants.

On peut alors introduire dans le langage de nouvelles instructions et de nouveaux types. Remarquons que la programmation devient un peu plus délicate et un texte un peu plus difficile à interpréter à la lecture, car on se trouve en présence d'un mélange d'instructions de deux machines.:

Exemple :

```
      DØ 1 I= 1, 100
      K = K+2                    (a)
1     S = S+K

      $DØ $1 I= 1, 100
      K = K+2                    (b)
$1    S = S+K

      DØ 1 I= 1, 100
      $ = K : K+2                (c)
1     $ = S : S+K
```

La séquence (b) donne, à l'exécution du programme Fortran, le même résultat que (a), mais il faut remarquer que la méta-machine génère cent fois :

```
K = K+2
S = S+K
```

En revanche, la séquence (c) ne produit rien à l'exécution Fortran, les instructions n'étant exécutées qu'une seule fois par la méta-machine. Pour éviter toute confusion de ce type, il est préférable de n'employer, dans le texte d'un programme, que des appels de méta-procédures à l'exclusion de toute autre instruction du méta-langage ; celles-ci sont réservées à la définition des méta-procédures.

Remarquons pour terminer que l'appel de méta-procédure présente un gros intérêt car elle permet de vérifier le type et la nature des paramètres formels d'une procédure. Elle permet surtout à l'utilisateur de repérer un paramètre par son nom et non par sa position. Si nous prenons le sous programme :

```
SUBRØUTINE SUB (X, Y, EPS, K, T)
```

il sera beaucoup plus simple d'écrire et de lire :

```
$ (SUB, X=1., Y=F(1, 1), T=25.0, K=0, EPS=0.001 $)
```

plutôt que :

```
CALL SUB (1., F(1, 1), 0.0001, 0, 25.0).
```

La méta-procédure SUB a alors pour seul rôle de générer un appel correct du sous programme SUB en générant, dans certains cas, des paramètres implicites. En cas d'erreur sur le nombre ou la position des paramètres, celle-ci est détectée dès la première phase et non au bout de trente minutes d'exécution par exemple.

2.3.1.3. Méta-langage spécialisé

Un méta-langage peut être indépendant d'un langage objet (MAG-15 sur Mitra 15, par exemple) ; il existe aussi des méta-langages dont les structure est étroitement liée à un langage donné (Méta-cobol). Le principe est le même, mais le méta-programme ne doit comporter que des méta-instructions, des appels de méta-procédure et des instructions du langage objet. Ils perdent un intérêt : la transportabilité des méta-programmes. On peut en effet imaginer, à la limite, un module composé uniquement d'appel de méta-procédures qui pourraient être générées en Cobol ou en Fortran, suivant le texte des méta-procédures. Ils gagnent en efficacité : ils peuvent alors reconnaître les déclarations, analyser le type des paramètres des appels de procédure et faire une action en conséquence, et les appels de méta-procédures peuvent alors apparaître comme de nouvelles instructions du langage.

Dans ce dernier cas, il arrive que les deux phases, méta-traduction et compilation, soient menées de front, c'est-à-dire que le méta-traducteur et le traducteur sont, en réalité, deux modules d'un seul processeur. Le traducteur "passe la main" au méta-traducteur lorsqu'il rencontre une méta-instruction. Pour l'utilisateur, tout se passe comme s'il existait deux machines, la méta-machine étant simplement plus évoluée qu'une méta-machine générale.

2.3.1.4. Méta-assembleur

Il y a deux manières de présenter un langage d'assemblage : un langage d'assemblage est accepté directement par une machine virtuelle. L'exécution d'une instruction de machine virtuelle assembleur a alors le même effet qu'une instruction machine ; seule existe la différence de description d'une instruction en langage d'assemblage ou en code binaire. Cette solution s'avère satisfaisante lorsque les instructions et les données forment deux ensembles disjoints ; elle pose certaines ambiguités cependant, en particulier, une construction dynamique d'instructions ne s'interprète pas très bien avec ce modèle, et, de même, l'insertion de réservation dans le texte d'un programme n'est pas claire. Dans un langage évolué, on a ou non le droit de mettre des déclarations n'importe où, ceci est décelé à la compilation et ne pose jamais d'erreur à l'exécution ; en assembleur, une séquence du type

```
    LW, 1     A
A   RES       1
    AW, 1     B
```

est bien acceptée par l'assembleur et provoque une erreur à l'exécution. On est donc souvent amené à présenter un assembleur comme une machine qui accepte un texte en langage d'assemblage et produit une image mémoire qui contient le programme objet. Une ligne du programme source n'est alors qu'une directive de génération, (d'autant plus que, sur de nombreux assembleurs, ce qui est couramment appelé instruction n'est en fait que l'appel d'une macro-instruction).

Il y a donc deux manières de présenter un méta-assembleur :

- un méta-assembleur est un méta-traducteur qui produit un texte en langage d'assemblage,
- un méta-assembleur est un assembleur évolué qui admet les instructions classiques d'un langage de programmation (bouclages, tests, appels de procédures, calculs sur variables) en plus des directives simples de génération.

Nous utiliserons cette deuxième solution pour présenter rapidement les diverses instructions de Métasymbol ; en fait, un programmeur de méta-assembleur se réfère au premier modèle lorsqu'il veut décrire de simples traitements, et au second s'il veut décrire des objets indépendants d'un traitement particulier ou des associations entre objets et traitements.

2.3.2. Métasymbol

Métasymbol est un méta-assembleur. Il accepte un texte source qui contient des méta-instructions et des directives de génération, et produit un module objet. Pour éviter toute confusion, nous désignerons par le mot de directive les instructions propres au méta-assembleur (méta-instruction et directives de génération). Nous réserverons le nom d'instruction pour celles de la machine réelle.

```
LW, 5     A
```

est une directive métasymbol ; c'est l'appel d'une macro-instruction GEN ; elle produira une instruction chargement.

2.3.2.1. Syntaxe

Une ligne Métasymbol est divisée en quatre zones séparées par des blancs. Les identificateurs sont des suites d'au plus 63 caractères alphanumériques dont l'un au moins est une lettre (les symboles s, : sont aussi autorisés pour constituer des identificateurs) ; on les appelle couramment symboles, le caractère "*" joue un rôle particulier s'il est employé (devant un symbole) en zone argument ; généralement, il sera la marque d'un adressage indirect.

2.3.2.2. Objets manipulés

Les objets manipulés par Métasymbol sont de trois types fondamentaux.

- Absolu

Un objet est absolu s'il ne dépend pas de l'emplacement où sera implanté le programme. Les constantes de types simples, classiques, entiers, files de caractères, réel simple précision sont des quantités absolues. Métasymbol traite tout objet absolu comme une chaîne de bits. L'expression 1+FS '1.' (FS' ' représente une constante réelle) n'a aucun rapport ni avec 2 ni avec FS'2';si la représentation hexadécimale de FS '1.' est X'40100000', 1+FS '1.' est représenté par X'40100001'.

- Translatable

Une variable translatable repère un emplacement, tel que nous l'avons défini dans le chapitre 1. Est implicitement considérée comme translatable toute référence externe ou en avant. Cette notion est caractéristique d'un langage d'assemblage. En pratique, on confond la notion d'adresse et de quantité translatable.

- Listes

Les éléments peuvent être regroupés dans une structure de liste, un élément pouvant lui-même être une liste. Si L représente la liste

10, (11, 12), 13

L (1) représente la constante 10

L (2) représente la liste 11, 12

L (2, 1) représente l'élément 11

2.3.2.3. Calcul des expressions arithmétiques

Une expression est un élément du langage qui représente une valeur. Elle est entièrement calculée pendant la phase d'assemblage pour des opérandes de type absolu et se compose d'un terme ou d'une combinaison de termes séparés par des opérateurs arithmétiques, parenthésée de façon classique. Un terme est une méta-variable, une constante ou un élément simple de liste.

Les opérateurs peuvent être :

opérateur entier +, -, *, ** (décalage logique), / (division entière)
opérateur logique ¬ (NØN), & (ET), | (ØU), ||(ØU exclusif)
opérateur de relation <, < =,> , >=, =, ¬ =

Dans une expression, chaque terme est calculé et remplacé par sa valeur interne (binaire) ; ceci explique la remarque faite plus haut sur le calcul de 1+FS '1.'. Les opérations logiques sont faites digit par digit ; le résultat des comparaisons donnent les valeurs 0 si faux, 1 si vrai.

De nombreuses restrictions existent lorsqu'on manipule des variables translatables ; en effet, ces expressions ne sont évaluées que lors de l'édition de liens, seules les additions et soustractions sont autorisées. Il existe cependant un cas où une soustraction entre deux nombres translatables peut être évaluée : lorsqu'ils sont repérés par rapport à la même section, la différence est alors une quantité absolue, (la somme de deux nombres translatables repérés par rapport à la même section n'a aucun sens).

2.3.2.4. Directives d'attribution de valeur à un symbole

Un symbole peut désigner une constante absolue ou translatable. Un symbole désignant une constante est défini par la directive EQU, et ne peut être redéfini :

```
symbole     EQU     expression
```

Une méta-variable est affectée à l'aide de la directive SET et peut être redéfinie ; une méta-variable peut repérer une liste et un élément de liste peut être modifié par une directive SET.

Exemple :

```
A         EQU     5
B         SET     A ** 2          donne à B la valeur 20
A         SET     B               interdit, A désigne une constante
B         SET     A, B            donne à B la valeur 5,20
B (3)     SET     B               donne à B la valeur 5,20,(5,20)
B (3,1)   SET     B(3,1) + 1      donne à B la valeur 5,20,(6,20)
```

Nous insisterons sur le fait que les constantes ou variables ainsi définies ne laissent aucune trace directe dans le programme cible. Dans certaines directives de génération, un symbole en zone étiquette est considéré comme une constante désignant cette adresse.

2.3.2.5. Directives de rupture de séquence

Il existe plusieurs directives permettant de faire des ruptures de séquences (au niveau de la méta-machine) : GØTØ, DØ, WHILE et l'appel d'une procédure. Nous ne décrirons ici que la directive DØ ; elle se présente sous la forme :


```
[ symbole ] DØ expression
            suite de directives : SD1
            ELSE
            suite de directives : SD2
            FIN
            suite du programme  : SD3
```

Si l'expression a pour valeur 0, on exécute SD2 puis SD3 ; si elle a la valeur 1, on exécute SD1 puis SD3. (Si nous remarquons que la valeur d'une comparaison est 0 ou 1, cette directive se comporte comme une méta-instruction *$si $alors $sinon $fsi*).

Si l'expression a une valeur positive : n, SD1 est exécutée n fois, le symbole prenant les valeurs de 1 à n, puis on exécute SD3. (La directive se comporte alors comme un *$pour chaque* simple qui sera évaluée pendant la traduction du texte source).

Exemple :

```
LISTE   SET   0
I       DØ    5
LISTE   SET   LISTE, I
        FIN
```

donne à LISTE la valeur 0, 1, 2, 3, 4, 5.

2.3.2.6. Définition de procédures

Il existe deux types de procédures : les procédures de fonction ou de commande. Les directives FNAME et CNAME permettent de donner un nom à une procédure, le corps de celle-ci étant compris entre deux directives PRØC et PEND. Une référence à une procédure contient le nom de la procédure comme premier élément de la zone commande. La ligne de référence à une procédure

peut être repérée à l'intérieur de la procédure : chaque zone forme une liste et la zone étiquette est référencée par LF, commande par CF et argument par AF. La transmission des paramètres se fait donc en fonction de la position des paramètres formels dans une ligne de référence.

Exemple :

```
SØMME   CNAME
        PRØC
LF      SET     CF (2) + AF (1)
        PEND
```

définit la procédure SØMME

```
A       SØMME, 4   5
```

provoquera l'affectation de la valeur 9 à A.

Des fonctions système permettent de compléter l'analyse de la structure de la ligne de référence :

```
NUM   (LISTE)
```

donne le nombre d'éléments de la liste "LISTE",

```
SCØR  (symbole, test 1, test 2, ..., test i, ..., test n)
```

demande la comparaison de symbole avec successivement test 1 ... test n ; la recherche s'arrête dès qu'une égalité de symboles est rencontrée, la fonction prend alors le rang du test trouvé ou la valeur zéro si aucun test n'est égal au symbole. Symbole peut être un symbole explicite ou un élément d'une des listes AF, CF, LF référencées par une procédure.

Exemple d'emploi :

```
CALCUL   CNAME
         PRØC
         DØ       SCØR (AF(1), SØMME)
LF       SET      AF (2) + AF (3)
         ELSE
LF       SET      0
         FIN
         PEND
A        CALCUL   SØMME, 3, 4
B        CALCUL   3, 4
```

donnent à A la valeur 7 et à B la valeur zéro.

```
TCØR
```

est une fonction analogue à SCØR qui permet de rechercher le type d'un élément d'une liste de référence à une procédure.

2.3.2.7. Directives de génération

Ce sont les instructions d'assemblages classiques qui peuvent être considérées comme les instructions du produit fourni par Métasymbol. Ce ne sont plus, à proprement parlé, des méta-instructions.

Ces directives ont pour but de définir le module-objet cible. Elles comprennent les déclarations de section, de références ou définitions externes et les directives de génération proprement dites. Nous rappellerons que la directive CSET a le format :

```
[ symbole ]  CSECT  expression
```

expression donne la valeur de la protection attachée pendant l'exécution à

la section ; le symbole, s'il existe, désignera une constante translatable, premier emplacement de la section. Cette directive crée, dans le module objet, une déclaration de section de contrôle.

La directive ØRG demande à Métasymbol de créer une commande de définition d'origine ; elle a la forme suivante :

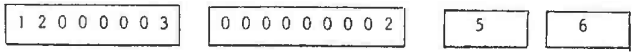
```
        ØRG     expression.
```

Les directives GEN et DATA demandent à Métasymbol de produire des commandes de chargement ; la directive GEN sert simplement à décomposer un emplacement mot, octet, demi-mot ou double mot en une suite de zones, une zone étant désignée par sa longueur en bits.

Exemple :

```
A     CSECT         0
      ØRG           A + 1
      DATA          2
      ØRG           A
      GEN,4,4,24    1,2,3
      ØRG           A + 2
      DATA          5
      DATA          6
```

donne au contenu de A, ... A + 4 les valeurs suivantes en début d'exécution, (les valeurs sont exprimées en hexadécimal).



Remarquons que les directives CSECT et DSECT sont implicitement suivies d'une directive ØRG fixant dans le module-objet la valeur de l'origine au

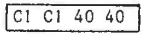
début de la section déclarée.

Deux directives permettent de générer des chaînes de caractères. Elles ont pour noms TEXT et TEXTC ; la directive TEXTC introduit un octet de comptage en tête de la chaîne.

Exemple :

```
        TEXT    'AA'
```

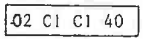
produira



(X'C1' est le code EBCDIC de A et X'40' celui du blanc)

```
        TEXTC   'AA'
```

produira



Il n'y a, dans Métasymbol, aucune définition d'instruction machine. Celles-ci sont définies par des directives CØM ; cette directive existe dans l'assembleur Symbol et se comporte alors comme une définition de macro-instruction.

La directive

```
        ETI     CØM, liste de zone   liste d'argument
```

a le même effet que

```
ETI   CNAME
      PRØC
LF    GEN, liste de zone   liste d'argument
      PEND
```

L'instruction machine chargement immédiat peut être définie par la directive :

```
LI    CØM,8,4,20          X'22', CF (2), AF (1)
```

et la rencontre de

```
LI,5  9
```

produira

```
22 50 000 9
```

Compteur d'emplacement

Pour permettre à l'utilisateur de définir des variables translatables représentant des adresses, Métasymbol tient à jour deux compteurs : l'un est dit d'emplacement au chargement (repéré par $), l'autre d'emplacement à l'exécution (repéré par $$). Dans la plupart des applications, ces compteurs auront la même valeur ; Métasymbol les initialise lors d'une directive ØRG, CSECT, USECT et les incrémente à chaque instruction, demandant la production d'une commande de chargement (GEN, DATA, TEXT, TEXTC, référence à une directive CØM), et donne au symbole figurant en zone étiquette la valeur du compteur d'emplacement à l'exécution.

Il est cependant des cas où l'on désire qu'un segment soit chargé quelque part et exécuté ailleurs (à la suite d'un transfert mémoire). Si une instruction de branchement interne, telle que :


```
BCR,3    ETI
```

figure dans ce segment, il est important que ETI repère l'endroit où l'on devra se brancher à l'exécution et non au chargement ; c'est pour cette raison que les deux compteurs sont différenciés, la direction LØC permettant de définir le premier.

Cette configuration est extrêmement rare, et nous ferons la confusion entre les deux compteurs d'emplacements que nous désignerons par $, convention généralement adoptée par les programmeurs Métasymbol.

2.3.3. Exemple d'une petite application Métasymbol

L'écriture d'un module de sortie de message est un problème que l'on rencontre souvent lorsqu'on écrit un compilateur ou un système.

2.3.3.1. Spécifications

On désire qu'un programmeur puisse lancer une écriture de message en spécifiant son numéro d'ordre ou l'adresse d'un mot contenant son numéro d'ordre. La chaîne de caractères constituant un message peut être d'une longueur comprise entre 1 et 32 caractères. Les chaînes sont rangées séquentiellement et on leur associe simplement une liste de pointeur, chaque pointeur repérant l'adresse de mot d'une chaîne.

2.3.3.2. Description des modules à décrire avant la programmation

La fonction AFA permet de savoir si un astérisque figure devant un paramètre d'une liste argument ; la liste LISTE contient les adresses des messages pendant l'assemblage et est utilisée dans la procédure d'écriture.

```
*
*         DESCRIPTIØN DE LA TABLE DES MESSAGES
*
*
          ØPEN   T1,T2,I,A,B
T1        CSECT  0              TETE DE LISTE DES POINTEURS
T2        CSECT  0              DEBUT D IMPLANTATIØN DES MESSAGES
I         SET    1
CHAINE    CNAME
*                               PRØCEDURE DE CØMPOSITION DE LA TABLE
          PRØC
          LØCAL  ETI,ETI1
*                               LES SYMBØLES ETI ET ETI1 PEUVENT ETRE
*                               REUTILISES PAR LE PRØGRAMMEUR
ETI       EQU    8
          USECT  T1
ETI1      TEXTC  AF             MET UNE CHAINE DANS LA TABLE T1
          USECT  T2
LISTE(I)  SET    ETI1
          DATA   ETI1           PØINTEUR DANS T2
          USECT  ETI            RETØUR A LA SUITE DANS LE PRØGRAMME GENERE
I         SET    I+1
          PEND
*
*
*         PROCEDURE D ECRITURE
*         ELLE UTILISE LA PROCEDURE M:PRINT (MESS, A) OU A EST L ADRESSE
*         D UNE CHAINE DE CARACTERES
ECRIT     CNAME
          PRØC
          DØ     AFA(1)=0.
*                               L ADRESSE DE LA CHAINE EST CØNNUE A
*                               L ASSEMBLAGE
          M:PRINT (MESS, LISTE (AF(1))
          ELSE
*                               CALCUL DE L ADRESSE A L EXECUTION
          STW,1  A              SAUVEGARDE REGISTRE
          LW,1   AF(1)
          LW,1   T1+1,1
          STW,1  B
          M:PRINT (MESS,* B)
          LW,1   B              RESTAURATION
          FIN
          PEND
A         RES    1
B         RES    1
          CLØSE  T1,T2,I,A,B
```


Les directives ØPEN/CLØSE permettent de définir des symboles locaux à une suite d'instructions ; LØCAL permet de définir un symbole dont la durée de vie est l'évaluation d'une procédure. (ETI est différent, à chaque appel, de CHAINE ; I et LISTE désignent le même objet à chaque appel).

L'utilisation de ces procédures est alors particulièrement facile :

définition d'une table

```
          CHAINE      'ABC'
          CHAINE      'MESSAGE'
          CHAINE      'XYZ'
```

demande d'écriture

```
          ECRIT       2
```

provoque l'impression de MESSAGE

Si l'adresse J contient le numéro d'ordre d'une chaîne, il suffira d'écrire

```
          ECRIT       * J
```

Remarques :

Un programme étant écrit, les messages de la table peuvent être modifiés : une modification d'un message entraîne la modification d'une carte.

La suite de messages peut être rallongée sans aucun problème, il suffit de rajouter des lignes de référence à CHAINE.

L'application terminée, on constate que l'on appelle très souvent ECRIT avec la forme indirecte et l'on désire gagner de la place en mémoire, il suffit de remplacer

```
STW,1    A          par          BAL,RL     SSPECRIT
...                              DATA       AF (1)
LW,1     A
```

dans la procédure, SSPECRIT étant un sous programme de quelques intructions, et le reste du programme est inchangé.

2.4. ADAPTATION DE METASYMBOL A UNE ANALYSE MODULAIRE

ANALOGIES AVEC LE PROJET CIVA

Nous devons nous rapprocher le plus possible de la structure d'une application décrite en Civa ; il est donc intéressant d'isoler, dans le système de compilation Métasymbol, une organisation qui permet de définir des objets analogues à ceux manipulés en Civa : module, classe, procédure, de manière à ce que le programme Métasymbol se rapproche le plus possible du résultat de l'analyse. Si nom est une notion Civa, nous désignerons par Xnom un objet de l'application système de compilation permettant d'écrie le software Civa.

2.4.1. Définition d'un Xmot clé

En Civa, tout mot clé ou variable réservée est souligné pour éviter toute ambiguïté et permettre au programmeur de définir n'importe quel identificateur ; dans le système X-Civa, tout mot clé commence par X ou Z ; un identificateur est une suite de caractères alphanumériques dont le premier est une lettre différente de X ou Z.

Pour définir un code, il sera préférable d'utiliser les fonctions SCØR plutôt que des directives EQU, le code pouvant alors être réutilisé en tant que variable.

2.4.2. Définition des sections utilisables

Quatre sections de contrôle sont déclarées dans la "classe système" associée au compilateur : elles ont pour noms

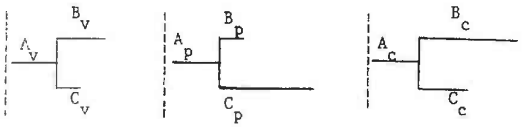
- P section associée au programme ; elle sera en "protection écriture exécution autorisée",
- PI sera de même protection ; elle permet d'inclure des procédures internes,
- V contiendra les variables,
- C contiendra les constantes.

Pour entrer dans une nouvelle section, on utilisera une procédure

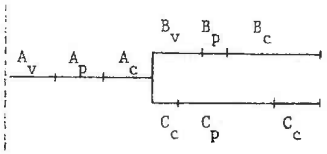
XGNS { P / PI / V / C }

intérêt de cette procédure : les noms de sections sont normalisés et il est plus facile, pour le lecteur, d'interpréter XGNS P plutôt que USECT ETI ou ETI est une adresse qui se trouve "quelque part dans le programme". Les symboles P, PI, V, C sont réutilisables par l'utilisateur ; en mode mise au

point, les sections seront dans des zones de protection différentes ; en mode exécution, elles seront dans une même zone (ce qui permet de gagner une place non négligeable en mémoire, surtout si l'on fait du recouvrement).



mise au point avec protection différente



exécution sans protection différente

2.4.3. Notion de Xmodule

Le seul lien existant entre deux modules est leur appel. Cet appel se caractérise par le nom du module appelé. La même règle sera appliquée dans notre application : les modules du compilateur ou Xmodules sont assemblés séparément, un module est connu de l'extérieur par son nom qui est une définition externe.

Pour tous les Xmodules, les opérations d'initialisation sont souvent les mêmes, (sauvegarde des registres, transmission des paramètres). Il est donc intéressant de définir trois méta-modules permettant de fixer la notion de Xmodule : XMØDULE, XFINMØD, XAPPEL.

Déclaration d'un Xmodule

On écrira :

```
          DEF      nom
nom       XMØDULE, "nom" [liste d'adresse]
```

L'idéal aurait été de définir une procédure du type

```
XMØDULE  nom, liste des adresses des paramètres formels
```

qui puisse faire toutes ces opérations. Malheureusement, Métasymbol n'autorise pas de déclaration du type

```
DEF    AF (1)
```

pour des raisons technologiques assez compréhensibles. De même, il n'est pas possible de passer d'un identificateur à la chaîne de caractères pouvant le représenter.

Une fin de module sera simplement déclarée par

```
XFINMØD
```

un appel de module, par

```
REF   nom              (uniquement pour le premier appel)
XCALL nom  [liste de paramètres]
```

Un paramètre effectif est défini par : une adresse ou (VAL, valeur).

Ces procédures gèrent les sauvegardes éventuelles, assurent les liaisons et les récupérations de paramètres. En mode mise au point, elles permettent de définir des points remarquables, par exemple, il est possible de deman-

der l'édition du nom d'un module lorsqu'il est appelé.

Exemple :

```
        XMØDULE   CNAME
                  PRØC
                  ...
                  DØ  XMISEAUPØINT
                  M:PRINT (MESS, A)
                  XGNS    C
        A         TEXTC   CF (1)
                  XGNS    P
                  FIN
                  ...
                  PEND
```

2.4.4. Définition d'une Xclasse

Il existe une directive particulière de METASYMBOL : SYSTEM qui permet d'insérer, dans le texte d'un programme, une suite de directives métasymbol préalablement définie dans une partition d'un fichier ; elle se comporte comme l'appel d'un sous programme ouvert. Pour définir l'équivalent d'une classe, il suffit de constituer une suite de directives métasymbol et de la ranger dans une partition ayant le nom de la classe, les déclarations d'emplacement figurant dans une section fictive de même nom.

Exemple : un module de compilateur qui serait décrit en Civa par

```
module m1 ;          module m2 ;          classe c ;
utilise c ;          utilise c ;          entier i, j, k ;
i = 1 ;              j = 2 * i ;          fin classe
m2 ;                 fin module
k = j + 3 ;
fin module
```


s'écrirait en Métasymbol, dans notre système de compilation

```
      DEF      M1              DEF      M2          C   DSECT        0
M1    XMØDULE, "M1"        M2  XMODULE, "M2"        I   RES          1
      SYSTEM   C               SYSTEM   C           J   RES          1
      LI,1     1               LW,1     I           K   RES          1
      STW,1    I               MI,1     2               END
      REF      M2              STW,1    J
      XCALL    M2              XFINMØD
      LW,1     J               END
      AI,1     3
      STW,1    K
      XFINMØD
      END
```

Pour des raisons techniques, nous imposerons que les directives SYSTEM figurent en tête des modules, (avant la déclaration XMØDULE qui contient une directive XGNS P permettant de sortir des sections fictives introduites par les SYSTEM).

La classe Système qui comporte toutes les déclarations de procédures telles que XGNS, XMØDULE, les définitions de registres... doit être utilisée par tout module ou par toute classe ; elle a pour nom XCIVA.

2.4.5. Notion de procédure

La plupart des procédures seront interprétées comme des méta-procédures Métasymbol. Ceci est possible car les procédures en Civa sont toujours internes à un module ou à une classe, et il en sera de même dans notre réalisation. Une restriction est introduite : toute déclaration de procédure doit précéder son appel.

En fait, une méta-procédure se ramènera souvent à un appel d'un sous programme dès que cela s'avèrera plus efficace, la transmission des paramètres

se faisant alors le plus souvent par des registres.

Exemple : procédure de transfert de chaîne d'octets

```
MØVE    emetteur, recepteur, longueur
```

Une chaîne est désignée par son adresse, sa longueur.
Une adresse est repérée par son adresse d'octet ou une adresse de mot contenant cette adresse précédée de * ; de même une longueur, par sa valeur ou une adresse de mot

```
MØVE    (* ADA, *ADB, *L)
```

produira

```
LW, R1   ADA
LW, R2   ADB
LW, RB   L
BAL, RL  SSPMØVE
```

car il faut beaucoup plus de cinq instructions pour générer un mouvement de chaînes d'octets de longueur quelconque. En revanche,

```
MØVE    BA (A), BA (B), I
```

pourra produire

```
LB, I    A
STB, I   B
```

car il est possible de savoir si une adresse d'octets est sur une frontière de mot en testant pour A : BA (WA (AF (I))) = AF (I).

Une remarque s'impose cependant : l'écriture des procédures de base souvent appelées doit être faite par des programmeurs qualifiés, car chacune demande de l'attention. Par contraste, l'utilisation peut se faire sans précaution, tout le travail d'analyse d'un traitement élémentaire ayant été fait une seule fois.

Nous n'avons pas porté notre effort plus loin pour le rapprochement entre les instructions Civa et les traitements possibles par Métasymbol, mais nous avons surtout cherché à définir les procédures de base pour la réalisation d'un générateur.

2.5. GESTION DES MODULES ET DES CLASSES DE L'APPLICATION COMPILATION

Le compilateur sera composé d'un grand nombre de modules et de classes. Métasymbol est un produit extrêmement coûteux en temps d'assemblage. La modularité d'écriture et de compilation impose une édition de liens importante. Le texte source sera aussi important et les manipulations de cartes trop nombreuses sont exclues : on a donc été amené à définir quelques principes de gestion des éléments de l'application.

2.5.1. Inventaire des modules à gérer et organisation

2.5.1.1. Inventaire

- Module du compilateur

Il devra exister sous forme compressée pour éviter les manipulations intem-

pestive de cartes. Il sera aussi conservé sous forme de module-objet pour éviter des assemblages inutiles. Si le module est un module directeur utilisé plusieurs fois, il sera conservé sous la forme de module de chargement.

- Classes du compilateur

Elles existent uniquement sous forme compressée ; en effet, ce sont des fichiers SYSTEM qui doivent être complètement assemblés dans tout module. Les sous programmes appelés par celles-ci seront considérés comme des modules et conservés sous les deux formes compressée et module-objet.

- Modules de librairie Civa

Les modules de la librairie Civa sont conservés sous les trois formes source, module-objet et module de librairie.

2.5.1.2. Organisation

Il est important de noter que l'installation sur laquelle nous travaillons ne comporte pas, pour l'instant, de mémoire de masse sur disque et qu'aucun fichier disque ne peut être permanent.

Sous BPM (système sous lequel les premiers essais ont été faits), à chaque élément (module compressé, module-objet) était associé un fichier ; si nom est le nom d'un module, S:nom est le nom du fichier compressé, B:nom le nom du fichier sous forme de module-objet et nom le fichier module de chargement, s'il y a lieu.

Sous SIRIS 7, les fichiers partionnés répondent parfaitement à notre type d'application, au moins en ce qui concerne les modules-objets. Les modules-

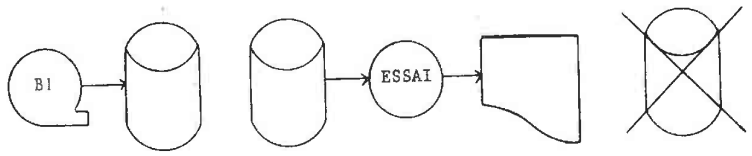
objets associés au module du compilateur seront regroupés en un fichier partitionné, chaque fichier ayant comme clé le nom du module, de même pour les modules de librairie. La même solution a été choisie pour la conservation des modules et classes sous forme compressée.

2.5.2. Sauvegardes au niveau de l'application

C'est un des points les plus importants si l'on veut minimiser le coût de la phase mise au point.

Deux bandes multifichiers B1 et B2 permettent d'avoir une exploitation relativement souple, (B2 est en principe une copie de B1).

Un essai sera caractérisé par trois étapes :



- restauration des fichiers venant de B1 sur disque,
- essai proprement dit,
- destruction des fichiers sur disque.

Une mise à jour sera décomposée en deux phases :

première phase : mise à jour proprement dite

- restauration des fichiers à partir de B1,
- mise à jour sur disque,
- sauvegarde des fichiers disques sur B1,
- destruction des fichiers sur disque ;

deuxième phase :

si la mise à jour a été correcte, copie de B1 sur B2, sinon copie de B2 sur B1 (éventuellement sélective).

Ceci ne cause aucun problème sous SIRIS 7, le nombre de fichiers proprement dits étant fixe ; il est facile de cataloguer une fois pour toutes les Job restauration, sauvegarde, destruction des fichiers sur disque, copies.

Sous BPM, vu l'absence de fichier partitionné, le nombre de fichier évolue et les sauvegardes et restaurations deviennent délicates.

Pour la restauration, il faut toujours connaître l'emplacement exact des fichiers sur bande pour minimiser les mouvements de bande.

Pour la sauvegarde, il faut être sûr que tous les fichiers à sauvegarder existeront lors de l'opération de sauvegarde, et ceci est pratiquement imprévisible dès qu'une erreur s'est produite pendant la mise à jour. Supposons par exemple que les fichiers F1... Fi existent avant la mise à jour, que le fichier Fk (k < i) doit être modifié et les fichiers Fi+1..Fn introduits, le module de sauvegarde prévoiera la sauvegarde de F1 jusqu'à Fn ; si la mise à jour du fichier Fk s'est mal passée, le module de sauvegarde

s'arrêtera sur Fk-1 et tous les autres seront perdus. Pour remédier à cet inconvénient très fréquent sur toute application manipulant de nombreux fichiers, il a été écrit un processeur spécialisé qui fait une copie de tous les fichiers existant sous un numéro de compte, sur une bande multifichiers. L'option inverse a été prévue, c'est-à-dire la création sur disque de tous les fichiers existant sur une bande multifichiers. Les Job de sauvegarde et de restauration sous BPM s'écrivent alors :

```
! JØB          RESTAURATIØN
! ASSIGN       M:EI, (LABEL, X), (INSN, B1)
! TRBD         (BADI)

! JØB          SAUVEGARDE
! ASSIGN       M:EØ, (LABEL, X), (ØUTSN, B1)
! TRBD         (DIBA)
```

Il est intéressant de noter que ce processeur une fois écrit a été réutilisé sous BPM par toutes les applications utilisant de nombreux fichiers. Il semble donc qu'une des premières opérations à faire, lorsqu'on décide d'implanter une application modulaire sur un système donné, est d'étudier les moyens de sauvegarde et de restauration complète des fichiers de l'application, et d'écrire un programme spécialisé s'il y a lieu, le temps perdu par l'écriture de ce programme étant largement compensé par les économies ultérieures qu'il permet de réaliser.

2.5.3. Modification des textes des modules et des classes de l'application compilation

2.5.3.1. Modification locale à un Xmodule

Une anomalie dans le fonctionnement du compilateur a été décelée ; le remède à apporter est une modification d'une ou plusieurs instructions d'un Xmodule. Deux solutions sont possibles :

- modifier le texte source : ceci demande alors le ré-assemblage d'un Xmodule et l'édition de liens du compilateur ; cette perte est légère tant que le compilateur est en phase d'essai, module par module, mais ceci s'avérera coûteux lorsque le compilateur sera en version définitive. Il importe donc que cette nouvelle compilation soit justifiée et que la modification ait été testée auparavant.

- modifier le programme objet : on utilisera alors les services d'aide à la mise au point du système SIRIS 7 (instructions MØDIFY et INSERT). Ce type de service est très utile pour essayer et tester une modification sous cette forme. Il importe donc, dès qu'une modification devient définitive, de répercuter cette modification sur le texte source.

Finalement, ces deux méthodes s'avèrent complémentaires : une modification est testée en modifiant le programme objet et dès qu'elle s'avère satisfaisante, on la répercute sur le texte source.

2.5.3.2. Modification importante

Une modification devient importante lorsqu'elle affecte plusieurs Xmodules ; en particulier, toute modification du texte d'une Xclasse du compilateur est importante : elle oblige à recompiler les modules qui l'utilisent. Une modification sur un module peut devenir importante si ce module appartient à plusieurs modules directeurs, (certains modules de génération sont utilisés par l'éditeur de liens).

Métasymbol est un traducteur coûteux et il est dommage de tout recompiler si l'on peut s'en passer. De plus, plusieurs personnes travaillent parallèlement sur le compilateur et chacun ne connaît pas toutes les relations Xappelle et Xutilise existant dans l'application, pour déduire toutes les conséquences de la modification d'une unité. Nous avons donc écrit un processeur qui, connaissant une série de modifications faites sur un certain nombre d'unités, détermine quels sont les méta-assemblages et éditions de liens à recommencer, et dans quel ordre. Ce programme fait un travail analogue au module, appelé par l'interprête de langage de commande qui détermine, sur une application Civa, la conséquence d'une modification et provoque toutes les compilations nécessaires ; il a été décrit dans un rapport de D.E.A. [7] , nous nous bornerons à en rappeler les éléments principaux.

2.5.3.2.1. Graphe associé aux unités de nomenclature de l'application compilation

- Unité de nomenclature de l'application compilation

Une unité de nomenclature est caractérisée par :

- son nom,
- son type (Xclasse ou Xmodule),
- son état : une Xclasse est toujours à l'état compressé ; un Xmodule peut être à l'état . compilé (il existe alors sous forme compressée et empilée,
  . édité (il existe aussi sous forme de module de chargement.

Nous appellerons

$\mathcal{S}$ l'ensemble des Xclasses

$\mathcal{M}$ l'ensemble des Xmodules compilés

$\mathcal{M}\mathcal{C}h$ l'ensemble des Xmodules édités

$E = \mathcal{S} \cup \mathcal{M} \cup \mathcal{M}\mathcal{C}h$

- Relations dans E

Les éléments de E sont liés par deux relations :

$\mathcal{U}$

si $s \in \mathcal{S}$ et $x \in E$

alors $x \mathcal{U} s \iff$ x "utilise" s

A

si $m, p \in \mathcal{M} \cup \mathcal{M}\mathcal{C}h$

alors $m \, A \, p \iff$ m "appelle" p

Nous appellerons :

$$\Gamma = \mathcal{U} \vee A \quad \text{et} \quad R = \Gamma^{-1}$$

2.5.3.3. Influence d'une modification

Une modification sur une Xclasse impose un assemblage de tous les Xmodules qui l'utilisent, directement ou indirectement.

Une modification sur un Xmodule (soit directe, soit indirecte) demande que soient réédités tous les modules de chargement contenant ce module.

Si y E est modifié, on peut définir l'ensemble C des modules à assembler

$$C = (\mathcal{U}^{-1})^* (y) \cap (\mathcal{M} \cup \mathcal{M}\mathcal{C}h)$$

et l'ensemble L des modules à éditer

$$L = R^* (y) \cap \mathcal{M}\mathcal{C}h$$

2.5.3.4. Travail effectué par le processeur de mise à jour de l'application

Lorsqu'on veut faire une mise à jour de l'application, il suffit d'écrire une série de cartes de commande précisant :

- la création ou la suppression d'un point,
- la modification d'un point,
- la création ou la suppression d'un arc.

Le travail réalisé est alors une mise à jour d'un fichier contenant la description complète du graphe (E, Γ), une édition éventuelle de ce graphe, l'édition de la liste des éléments à recompiler et à rééditer ; il peut aussi être fait une édition de cartes de commande permettant de réaliser effectivement ces modifications.

Si ce processeur a un intérêt assez faible pendant le démarrage de l'application, il devient très intéressant dès que celle-ci devient importante ; pour l'application compilation complète, on peut estimer la taille du graphe (E, Γ) supérieure à 100 points et 400 arcs, ce qui rend déjà difficile une mise à jour manuelle ; c'est, de plus, un bon moyen pour s'obliger à tenir à jour l'état de l'application.

Remarquons enfin que les modules, écrits pour constituer ce processeur, sont réutilisés pour l'écriture de l'interprête du module de commande.

3
procédures de génération d'instructions

3.1. ROLE DU GENERATEUR

Le compilateur Civa est un compilateur à deux passages : codifieur et générateur.

Le codifieur a plusieurs tâches essentielles :

- assurer le traitement des méta-instructions et remplacer les appels de méta-modules par une suite d'instructions simples ;
- analyser chaque instruction source pour produire une suite de commandes en langage intermédiaire (ou chaîne codée) ; dans cette chaîne, ne figure plus aucun nom, ceux-ci ont été remplacés par un couple (type, emplacement).

Le deuxième passage ou générateur a un rôle très précis : pour chaque commande de la chaîne codée, il génère une suite d'instructions machines ou plus exactement une séquence de module-objet. L'action la plus répétitive du générateur est donc la génération d'une instruction ou d'une commande de module-objet.

Le compilateur que nous présentons ne travaille en fait que sur un sous ensemble du langage Civa. Il devra donc être modifié pour accepter le langage complet. De nombreuses modifications se traduiront par le remplacement d'une instruction générée par une autre (ou par une séquence d'instructions). Il importe donc que cette modification soit la plus aisée possible ; en particulier, modifier une instruction générée ne doit demander que la réécriture d'une instruction du générateur, soit d'une ligne du programme.

Le générateur produit du module-objet : celui-ci, nous l'avons vu au chapitre 1, a une forme assez technologique, sinon rébarbative ; si son organisation logique doit être bien connue de toute personne participant à la réalisation ou la maintenance d'un ordinateur, sa forme précise n'a pas à être connue par tous. Il importe donc que tous les problèmes attachés à la forme du module-objet aient été résolus une fois pour toutes, et qu'il soit possible de demander de "générer l'instruction x" par une directive méta-symbol de même forme.

C'est la raison pour laquelle nous avons défini un ensemble de procédures permettant de générer facilement des instructions. (Notons que ce système de procédure peut s'adapter à n'importe quel compilateur et qu'il se prête facilement à toute modification, étant modulaire).

Exemple :

On veut générer une instruction de branchement à un sous programme (BAL) avec le registre de liaison 6 ; à l'exécution, l'adresse du sous programme sera connue par adressage indirect ; pendant la compilation, l'adresse qui, à l'exécution contiendra l'adresse du sous programme, est définie par un numéro de déclaration qui se trouve à l'adresse BA (A), et par une translation par rapport à cette déclaration. (Cette translation est contenue dans le registre 8 pendant la compilation).

Pour exprimer cela, il suffira d'écrire :

XGINSDEC, BAL, 6 (REG, 8, (AI, 1)), (ADB, BA (A))

XGINSDEC est le nom de la procédure (X Generer INStruction dont la partie adresse est définie par rapport à une DEClaration).

Pour arriver à ce résultat, nous trouverons deux niveaux de programmation intermédiaires :

- des modules de gestion du fichier de sortie,
- une procédure XSØRTBØ qui doit permettre d'écrire facilement des procédures identiques à celle présentée plus haut.

En utilisation normale, on ne se servira en fait que des procédures de génération proprement dites.

3.2. DEFINITION DU FICHIER DE SORTIE BINAIRE DU COMPILATEUR

On peut envisager deux organisations pour le fichier de sortie d'un compilateur :

- Organisation séquentielle

Elle est utilisée lorsque l'on veut associer un seul fichier au résultat d'une compilation. Si elle convient très bien à des applications simples : saisie d'un programme source, compilation, édition de liens, exécution, dans une même session, elle s'avère lourde dans le cas d'une application modulaire car il faut associer un fichier à chaque module ou à chaque classe.

- Organisation partitionnée

Un fichier partitionné est un ensemble de fichiers séquentiels dans lequel chaque sous fichier est repéré par un nom, et se manipule comme un fichier séquentiel. Cette organisation s'applique très bien à Civa : tous les modules-objets sont regroupés dans un seul fichier ; à chaque module ou à chaque classe est associée une partition dont la clé est le nom du module ou de la classe.

Pour permettre aux systèmes de génération d'instructions d'être réutilisés, nous admettrons les deux organisations, d'autant plus que l'organisation séquentielle est plus facile à mettre en oeuvre pour les essais d'un compilateur.

Nous avons donc créé une Xclasse "fichier de sortie du compilateur" ; elle contient les informations relatives à celui-ci, son nom est ZØRGFBØ.

Trois variables de cette classe sont particulièrement importantes pour l'utilisateur :

- XØRGBØ

désigne l'adresse d'un mot qui, à l'exécution, contient un code associé à l'organisation du fichier de sortie, 0 pour un fichier séquentiel, 1 pour un fichier partitionné ;

- XADRKEY

désigne l'adresse d'un mot qui, à l'exécution, contient l'adresse d'octet où l'on trouvera le nom de la partition (dans le compilateur Civa, ce sera le nom du module ou de la classe) ;

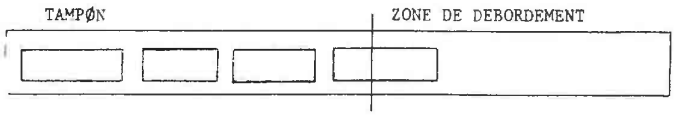
- XLKEYBØ

désigne la longueur de la clé (XLKEYBØ est défini par une directive EQU) ; dans notre application, elle sera rendue identique à la taille maximum d'un nom de module ou de classe. Cette classe contient aussi la déclaration des DCB de sortie.

3.3. MODULES DE GESTION DU FICHIER DE SORTIE

3.3.1. Principe

Le résultat d'une instruction de génération est la production d'une rubrique de chargement ; ces rubriques doivent être regroupées en enregistrements, puis écrits sur le fichier de sortie ; c'est le rôle des modules ZBØI.



La taille d'un enregistrement doit être de 26 mots, la taille courante d'une rubrique de chargement est de quelques octets (6 à 8 en moyenne) ; à chaque demande de génération, la rubrique est ajoutée au contenu de la zone TAMPON ; lorsqu'elle empiète sur la zone de débordement, le contenu de la zone TAMPON est écrit sur le fichier de sortie, la partie utile de la zone de débordement est recopiée dans la zone TAMPON, et on recommence. De plus, sous le système BPM, à chaque écriture sur le fichier de sortie, le mot de contrôle de l'enregistrement est calculé. (Sous SIRIS 7, seul le premier enregistrement contient un octet de contrôle).

3.3.2. Description de la classe XBØ

Cette classe est rédigée de façon particulière : seule la description des tampons est accessible avec une directive SYSTEM, toutes les procédures

sont regroupées dans un programme métasymbol ; le nom de chaque procédure est une définition externe, (ceci de manière à pouvoir l'exploiter en dehors de notre application).

Elle contient deux mots désignant la rubrique à insérer :

(ZBØN) = nombre d'octet de la rubrique,
(ZBØAD) = adresse d'octet de la rubrique.

La procédure de base a pour nom ZBØI : elle insère la rubrique dans la zone tampon et déclenche une écriture s'il y a lieu ; elle ne peut être utilisée pour des chaînes de plus de 100 octets, (taille de la zone de débordement).

La procédure ZBØIG doit être utilisée à la place de ZBØI quand la longueur de la rubrique à insérer (désignée par ZBØN) peut dépasser 100 octets ; elle sera un peu moins rapide que ZBØI.

Le traitement doit être initialisé par l'appel de ZDBØI qui réalise les opérations éventuelles d'ouverture du fichier de sortie et d'ouverture d'une partition ; elle doit être appelée en début de toute constitution de module-objet.

Le traitement doit se terminer par l'appel de ZFBØI qui vide ce qui reste dans la zone tampon et ferme temporairement le fichier de sortie, (pour prévoir une réouverture éventuelle dans le cas d'une compilation "en rafale").

On trouvera un exemple d'utilisation de ces procédures dans la réalisation du module "post-compilateur" utilisé pendant la phase d'essai.

3.3.3. Exemple d'utilisation des modules ZBØI : module post-compilateur utilisé en phase d'essai

Pour permettre d'étudier le fonctionnement du générateur en dehors de l'existence du prééditeur de liens, le post-compilateur génère un module-objet équivalent à :

```
        CSECT   0
        DEF     $nom
$nom    RES     taille de la zone variable
        END
```

XRTVAR, taille zone variable et XR$NØM sont définies dans XRACTAB.

```
*********************************************************************
*                                                                   *
*                    POST-COMPILATEUR     VERSION DU 20 10 72       *
*    CE MODULE GENERE UN MODULE OBJET EQUIVALENT A:  DEF   $NOM     *
*                                                                   *
*********************************************************************
        SYSTEM  XCIVA
        SYSTEM  XRACTAB
        DEF     ZPOSTCO
        REF     ZBOI,ZDBOI,ZFBOI,ZBON,ZBOAD
        XGNS    V
        BAUND   8
T1      RES     1
SORTIE  DATA    X'00000003'     MODELE DE M.O. : DEF
        RES     3                 (ADR. PAIRE)   $NOM
        DATA    X'0A012000'     DEFINITION DE LA VALEUR DE LA DEF
        DATA    X'02000000'     DEFINITION DE LA VALEUR DE LA DEF
DSC     DATA    X'08000000'     DEFINITION D'UNE SECTION DE
*                                 CONTROLE STANDARD
        DATA    X'0E000000'     DEFINITION DE END
        XGNS    P
ZPOSTCO STW.XRL  T1
        BAL.XRL  ZDBOI          OUVERTURE DU FICHIER SORTIE DU MO
        LD.XRTO  XR$NOM         *RANGEMENT DU NOM DU M.O.
        STD.XRTO SORTIE+1       *    DANS LA DEF
        LW.XRTO  XR$NOM+2       *
        STW.XRTO SORTIE+3       *
        LW.XRTO  XRTVAR         *RANGEMENT DE LA TAILLE ZONE VAR.
        AWM.XRTO DSC              DANS LA DEF DE LA SECTION.
        LI.XRTO  BA(SORTIE)
        STW.XRTO ZBOAD          *SORTIE DE  SORTIE
        LI.XRTO  32             *       ET  DSC
        STW.XRTO ZBON           *
        BAL.XRL  ZBOI           *
        BAL.XRL  ZFBOI          FERMETURE DU FICHIER SORTIE DU MO
        B        *T1
        END
```


3.4. PROCEDURE XSØRTBØ

Cette procédure est avant tout une procédure de travail ; elle permet de composer des procédures de génération quelconques. Son rôle est simple : elle insère dans le tampon de XBØ une information repérée par son type et sa localisation. XSØRTBØ permet d'insérer des informations de quatre types : octet, chaîne d'octets, chaîne d'octets précédée d'un octet de comptage, mot ; le type sera précisé dans la zone CF (2) par un code BYTE, CHAIN, TEXTC, WØRD.

```
XSØRTBØ, BYTE          (VAL,5)
```

demande, par exemple, l'insertion d'un octet dont la valeur est 5. La zone argument de la procédure contient la localisation de l'information à insérer.

Nous avons voulu éviter que le programmeur soit obligé de multiplier les transferts pour rendre une demande de génération adaptable à nos procédures ; c'est la raison pour laquelle nous avons détaillé les localisations : il suffit d'indiquer où se trouvent les éléments à rassembler pour que les procédures s'en chargent. Cette localisation se présente sous la forme d'une série de désignations.

3.4.1. Désignation d'octet

3.4.1.1. Désignation simple

Un seul paramètre suffit, en général, pour désigner un octet.

Un octet peut être désigné par :

sa valeur	:	(VAL, valeur)
un registre	:	(REG, numéro de registre)
un mot	:	(ADW, adresse de mot)
un mot avec indirection	:	(ADWI, adresse de mot)
une adresse d'octet	:	(ADB, adresse d'octet)
une adresse d'octet avec indirection	:	(ADBW, adresse de mot)
une adresse d'octet avec une double indirection	:	(ADBWI, adresse de mot)

Dans les désignations REG, ADW, ADWI, l'octet est cadré à gauche dans le mot.

3.4.1.2. Désignation multiple

Un paramètre ne suffit plus à désigner un octet, la désignation a la forme suivante :

(adresse de base { [, translation]* , modification de valeur
, translation [, translation]* [, modification de valeur] })

L'adresse de base est désignée par l'une des désignations ADW, ADWI, ADB, ADBW, ADBWI.

L'adresse effective est obtenue en ajoutant des translations à l'adresse de base ; cette translation peut être :

connue, exprimée en octet	:	(DB, valeur)
contenue dans un mot, exprimée en octet	:	(IB, adresse de mot)
contenue dans un registre, exprimée en octet	:	(IBR, numéro de registre)
contenue dans un mot, exprimée en mot	:	(IW, adresse de mot)

L'octet ainsi obtenu peut être modifié par addition d'une valeur

(AI, valeur)

Exemple :

L'adresse POINTEUR contient l'adresse d'une chaîne d'octets dont le 8ème octet contient un numéro de déclaration (< 64) ; pour générer le début d'une rubrique de chargement par rapport à cette déclaration, il suffit d'écrire :

XSØRTBØ, BYTE (ADBI, PØINTEUR, (DB, 7), (AI, X'80'))

(Rappelons que si 1 est un numéro de déclaration, X'81' est la rubrique de chargement associée).

3.4.2. Désignation d'une chaîne d'octets

La chaîne d'octets se réduit à une chaîne de caractères connue ; on utilisera la spécification TEXT

(TEXT, "chaîne de caractères").

Exemple :

```
XSØRTBØ, CHAIN  (TEXT, "ABC")        produit X'C1 C2 C3'
XSØRTBØ, TEXTC  (TEXT, "AB")         produit X'02 C1 C2'
```

La chaîne n'est pas une chaîne de caractères alphanumériques, elle est alors désignée par son adresse et sa longueur ; sa longueur est précisée dans les mêmes termes qu'une localisation d'octet ; elle doit être inférieure à 256 dès que l'on utilise une spécification autre que VAL, REG, ADW, ADWI. L'adresse d'une chaîne peut être précisée en utilisant les spécifications ADW, ADB, ADWI, ADBW, pour désigner l'adresse de base et DB, IW, IB, IBR, pour les translations éventuelles : une adresse de chaîne est alors désignée par :

(adresse de base [, translation]*)

La chaîne est connue, mais elle n'est pas une chaîne de caractères alphanumériques : elle sera désignée par une spécification VAL

(VAL, zone [, zone]*)

où zone est, soit une valeur considérée comme étant sur 4 octets, soit un couple (valeur, longueur), longueur indiquant le nombre d'octets sur lequel est cadrée la valeur :

(VAL, (1, 1), (2, 1), (3, 1), (4, 1))

désigne la chaîne d'octets 1, 2, 3, 4 ;

(VAL, 1, 2, 3, 4)

désigne une chaîne de mots contenant 1, 2, 3, 4.

3.4.3. Localisation d'un mot

Un mot peut être connu ou contenu dans un registre : on utilisera les désignations VAL et REG ; si le mot est connu par son adresse, il sera simplement désigné par :

([*] adresse [, registre d'index])

3.4.4. Remarques

Le dernier paramètre de la zone argument de la ligne de référence à XSØRTBØ peut être un code : (TEST) ; celui-ci indique qu'un test de débordement de tampon doit être fait en fin de travail, (ceci évite de faire systématiquement un test à chaque sortie d'octet, par exemple).

Dans le cas où l'option de sortie est TEXTC, il est inutile d'indiquer la longueur de la chaîne de caractères, celle-ci étant recherchée par la procédure XSØRTBØ.

Avec les options CHAIN ou TEXTC, le dernier paramètre peut être (LØNG) ; il il indique que la chaîne peut dépasser la taille de la zone de débordement. Le résultat de la méta-procédure sera alors un appel de ZBØIG.

3.4.5. Intérêt de XSØRTBØ

Cette procédure doit permettre d'écrire facilement n'importe quelle procédure de chargement ; elle n'est, en principe, pas destinée au simple utilisateur des procédures de chargement, cela explique la présence de paramè-

tres "dangereux", tels que TEST ou LØNG.

Les séquences produites par la méta-procédure sont optimisées du point de vue temps d'exécution : ceci explique l'abondance de paramètres et à chacun correspond une séquence différente.

Par exemple, les instructions effectivement assemblées par

```
XSØRTBØ, BYTE  (VAL, 11)
```

seront

```
LI, R    11
STB, R   0, XBAD
MTW, 1   XBAD
```

où XBAD est un registre impair pointant dans la zone tampon. (On notera que ces procédures évitent de passer par ZBØI dans les cas où la séquence à générer est plus courte qu'un appel de sous programme).

```
XSØRTBØ, CHAIN      (ADB, BA (A)), (VAL, 40)
```

produira :

```
LI,   XBADA   40
STB,  XBADA   XBAD
LI,   XBADA   BA (A)
MBS,  XBADA   0
```

(XBADA est le registre pair associé à XBAD), et

```
XSØRTBØ, CHAIN      (ADB, BA (A)), (REG, 10), (LØNG)
```

produira :

```
STW, 10       ZBØN
LI, XBADA     BA (A)
STW, XBADA    ZBØAD
BAL, XRL      ZBØIG
```


3.5. PROCEDURES DE GENERATION

3.5.1. Principe d'écriture des procédures de génération

Nous n'avons pas voulu, les problèmes étant souvent assez différents suivant les procédures, donner des spécifications générales autres que les désignations précédentes ; chaque procédure aura donc son mode d'emploi. Il sera très facile, pour une application autre que Civa, d'adapter ces procédures à des cas non prévus, en utilisant XSØRTBØ et ses procédures associées XCHARØCT, XCHARAD, qui permettent de charger, dans un registre, un octet ou une adresse définie par les désignations vues précédemment.

L'ensemble de procédures actuellement définies produit un module-objet réduit à une section de contrôle standard, ce qui est suffisant pour de nombreuses applications ; il résoud tous les problèmes d'emplacement dans une section.

3.5.2. Exemple de réalisation d'une procédure de génération

Les procédures XGENREF et XGENDEF génèrent, à l'exécution, une déclaration de référence externe ou de définition externe. Lorsque le programmeur gère lui-même les numéros de déclarations, il devra simplement écrire :

$$\left\{\begin{matrix}\text{XGENREF}\\ \text{XGENDEF}\end{matrix}\right\} \left[,\text{PRØG}\right] \qquad \text{(désignation d'une adresse de chaîne de caractères).}$$

Le premier octet de la chaîne est un octet de comptage ; (NPRØG précise que la progression des compteurs est à la charge du programmeur).

Exemple :

XGENREF, NPRØG [TEXT, "9INITIAL"]

indique que la référence externe "9INITIAL" doit être jointe au module-objet.

Si le programmeur veut laisser aux procédures le soin de gérer les numéros de déclaration, il écrira alors :

XGENREF / XGENDEF , PRØG, [{ (REG, valeur) / (HALF, adresse, registre) }] (adresse de chaîne)

PRØG indique qu'il faut laisser progresser un compteur de déclaration ; REG et HALF indiquent que la valeur du compteur (qui est alors le numéro de déclaration) doit être rangée dans le registre ou le demi mot spécifié.

Si nous reprenons la définition d'une rubrique de déclaration de DEF ou de REF, le premier octet contient X'03' ou X'05', puis on doit trouver la chaîne de caractères définissant le nom de cette référence. La procédure correspondante est donc écrite comme suit :

```
XGENDEF  CNAME    1
XGENREF  CNAME    2
         PROC
LF       EQU      $
         DA       NAME*1
         XSORTBO,BYTE  (VAL,X'03')
         ELSE
         XSORTBO,BYTE  (VAL,X'05')
         FIN
         XSORTBO,TEXTC   AF
         DA          SCOR(CF(2),NPROG)*0
         MTW,1       XCOMPDEC
         DA          NUM(CF)*3
          DB         SCOR(CF(3,1),REG)
          LW,CF(3,2) XCOMPDEC
        ELSE
         LW,XR1     XCOMPDEC
         STH,XR1    CF(3,2),CF(3,3)
          FIN
        FIN
        FIN
        PEND
```


3.5.3. Procédures utilisables

3.5.3.1. XGENØRG

Cette procédure permet au compilateur de générer une définition d'origine.

3.5.3.2. XGENEND

Cette procédure génère une déclaration de contrôle standard ainsi que les rubriques de fin de module-objet.

3.5.3.3. XGENDATA

Cette procédure permet de générer une suite de constantes non translatables (soit une suite de rubriques chargement absolu).

Exemple :

XGENDATA (ADB, BA (A)), (VAL, 120)

découpe la chaîne implantée en BA (A) en groupes de 16 octets et insère des commandes de chargement pour former les rubriques correspondantes.

3.5.3.4. XGENDDEF

Cette procédure permet de générer une directive de définition de DEF. (Rappelons que dans un programme Métasymbol, "DEF A" déclare A et "A EQU ..." le définit).

Exemple : "déclaration de *module* x" :

si "x" se trouve dans le compilateur rangé à l'adresse BA(NMØD)

```
XGENDEF, PRØG, (REG, 5)  (ADB, BA (NMØD))
XGENDDEF                 (REG,5), (ICI)
```

ICI indique que la définition externe repère l'emplacement courant.

3.5.3.5. XDEMFR

Cette procédure permet de demander un numéro de référence en avant.

3.5.3.6. XGENDFR

Cette procédure permet de générer une définition de référence en avant.

3.5.3.7. XGENWABS

Cette procédure demande la génération d'une rubrique de chargement absolu d'un mot :

```
XGENWABS (VAL, X'2210 0001')
```

permet de générer LI, 1 1.

3.5.3.8. XGENWFR, XGENWDEC

Ces procédures demandent la génération d'une rubrique de chargement d'un

mot dont la partie adresse est définie par rapport à une référence en avant ou par rapport à une définition externe.

3.5.3.9. Procédure de génération d'instruction 10 070

Une instruction 10 070, ou plus exactement une rubrique de chargement contenant une instruction, est le rassemblement de plusieurs informations :

code instruction
registre
registre d'index
marque d'adressage indirect
adresse : numéro de déclaration, translation, résolution.

La procédure de génération d'instructions porte trois noms : XGINSABS lorsque l'adresse est absolue, XGINSFR si l'adresse est définie par rapport à une référence en avant, XGINSDEC si elle est définie par rapport à une déclaration.

Un appel de XGINABS est défini comme suit :

```
XGINSABS, code, registre      adressage [, NØTEST]
```

code :

mnémonique d'instruction de 10 070,
désignation d'octet ;

registre :

valeur,
désignation d'octet contenant le numéro de registre ;

adressage = (adresse [,(AI, adressage indirect)] [,(INDX, registre)])

adressage indirect :

valeur : 0 ou 1,

désignation d'octet (contenant 0 ou 1) ;

adresse :

(BYTE, désignation d'octet)

(REG, numéro de registre contenant la partie adresse)

(VAL, valeur)

(WØRD, adresse de mot [, numéro de registre])

Si NØTEST est précisé, il peut ne pas être fait de test débordement tampon ;

XGINSFR et XGINSDEC sont appelés comme suit :

{XGINSFR / XGINSDEC}, code, registre [, resolution]
adressage, declaration [, NØTEST]

déclaration :

(BYTE, désignation d'octet)

(VAL, valeur)

(WØRD, adresse de mot)

résolution :

code BYTE, HALF, WØRD, DBL précisant la résolution d'adresse :

octet, demi mot, mot, double mot (WØRD est implicite).

Exemple d'utilisation

Génération de "LI, 1 3"

XGINSABS, LI, 1 (VAL,3)

Génération de "STW, 1 * A+3" où A est la définition externe de numéro de déclaration 5

XGINSDEC, STW, 1 ((VAL, 3), (AI, 1)), (VAL, 5)

3.6. EXEMPLE D'UTILISATION DES PROCEDURES DE GENERATION

Gestion des sous programmes de librairie appelés par un module Civa.

3.6.1. Méta-procédure XGENCALSSPLIB

De nombreuses instructions se réduisant à l'appel d'un sous programme de librairie, nous avons donc créé, dans le compilateur Civa, une procédure XGENCALSSPLIB dont la séquence d'appel est :

$$\text{XGENCALSSPLIB} \left[, \begin{matrix} \text{DD} \\ \text{ND} \end{matrix} \right] \qquad \text{nom}$$

où "nom" est le nom du sous programme de librairie appelé. L'option DD (déjà déclarée) indique qu'une référence à ce sous programme a déjà été faite ; ND indique qu'aucune référence n'a été faite à ce sous programme ; l'absence de ces options indique que l'on ne sait pas si, dans un module-objet, on a déjà fait référence à ce sous programme.

Les instructions nécessaires à la génération d'un appel de sous programme de librairie étant nombreuses, cette procédure provoque un appel d'un sous programme ZLIBCAL si aucune option n'est précisée, ZLIBCA1 si l'option DD est précisée, et ZLIBCA2 si l'option ND est précisée.

Chaque sous programme de librairie est repéré dans les sous programmes ZLIBCAL, ZLIBCA1 et ZLIBCA2 par un code calculé dans la méta-procédure par une fonction SCØR.

Texte de la procédure XGENCALSSPLIB

```
        REF      I1,I2,I3
        SET      0
        SET      0
        SET      0
*****************************************************************************

                  PROCEDURE              XGENCALSSPLIB

*****************************************************************************
                  CETTE PROCEDURE GENERE UNE SUITE D'INSTRUCTIONS QUI
                      APPELENT UN SSP :ZLIBCAL,ZLIBCA1,ZLIBCA2
        CES SSP GENERENT L APPEL DU SSP DE LIBRAIRIE
                              SEQUENCE D'APPEL  XGENCALSSPLIB,DD    SYMBOLE
                                                              ,ND
                              OU  SYMBOLE EST LE NOM DU SSP DE LIBRAIRIE
                                  DD =ON SAIT QUE LORS DE LA COMPILATION
                                       ON A DEJA GENERE UN APPEL DU DIT  SSP
                                  ND =ON SAIT QUE L'ON A PAS ...
                                  CF(2)=' '  ON NE SAIT  PAS SI L'ON A ..
                  EXEMPLE:    XGENCALLSSPLIB     OITOR
                              XGENCALLSSPLIB,DD OITOR

*****************************************************************************
CA  CNAME
        PROC
        REF       A
        SIGNS     P
        SET       SCOR(AF(1),;
INITIAL,;                                   1
...,;                                       2
...,;                                       3
...)                                        4
        ERROR,3,A=0    'SSP INEXISTANT EN LIBRAIRIE'
        ...XEC1   A=1
        DO        NUM(CF)=1    * APPEL DE ZLIBCAL
        DO        I1=0
        SET       1
        REF       ZLIBCAL
        FIN
        BAL,XRL1  ZLIBCAL
        ELSE
        DO        SCOR(CF(2),DD)        *APPEL ZLIBCA1
        DO        I2=0
        SET       1
        REF       ZLIBCA1
        FIN
        BAL,XRL1  ZLIBCA1
        ELSE
        DO        SCOR(CF(2),ND)        *APPEL ZLIBCA2
        DO        I3=0
        SET       1
        REF       ZLIBCA2
        FIN
        BAL,XRL1  ZLIBCA2
        FIN
        FIN
        FIN
        CLOSE     A
        PEND
        CLOSE     I1,I2,I3
        END
```

On trouvera (page 3.23) le texte de la procédure XGENCALSSPLIB (les méta-variables I1, I2, I3 permettent de repérer le premier appel de l'un des sous programmes ; on peut ainsi utiliser directement XGENCALLSSPLIB dans un module du compilateur sans se préoccuper de déclarer les sous programmes appelés en référence externe, le mode d'emploi de XGENCALLSSPLIB se réduit à l'écriture d'un appel correct).

3.6.2. Tables utilisées par le générateur pour la gestion des sous programmes de librairie

Le nom d'un sous programme de librairie, dans un module-objet produit par le générateur, sera une référence externe ; il importe donc que celle-ci soit déclarée une seule fois. Deux tables seront donc utilisées :

- une table des noms des sous programmes de librairie,
- une table des numéros de déclaration (module-objet) de chacun de ces sous programmes.

La table des noms TAB1SSP est une constante et l'on peut admettre qu'elle figure dans plusieurs segments de recouvrement du générateur ; c'est la raison pour laquelle elle est attachée au module métasymbol contenant ZLIBCAL, ZLIBCA1, ZLIBCA2. La table des numéros de déclaration (TAB2SSP) doit être, dans la racine du générateur, une référence externe ayant toujours le même numéro dans un module-objet ; c'est pourquoi, nous avons fait de TAB2SSP une section fictive et la déclaration de cette section figure dans la racine du générateur. Une procédure SSPLIB permet de déclarer l'existence d'un sous programme de librairie et de créer ses entrées dans les tables correspondantes.

Au niveau du compilateur, l'introduction d'un nouveau sous programme de librairie se réduira à deux opérations : ajouter un nom dans l'appel de la fonction SCØR de XGENCALSSPLIB et dans le module métasymbol contenant ZLIBCAL, un appel de la procédure SSPLIB.

```
         SYSTEM    XGENINST
         DEF       ZLIBCAL,ZLIBCA1,ZLIBCA2
         APEN      NSSP
SSP      SET       0
*****************************************************************************

              PROCEDURE   SSPLIB
  CETTE PROCEDURE GERE LES TABLES TAB1SSP ET TAB2SSP
  TAB1SSP (DSECT) CONTIENT LES NUM. DE REF. DES SSP DE LIBRAIRIE DEJA UTILISES
                          O SINON SUR DES DEMI-MOTS
   TAB2SSP (CSECT) CONTIENT LES NOMS EN TEXTE DES SSP DE LIBRAIRIE
*****************************************************************************
AB1SSP   DSECT     0
AB2SSP   CSECT     0
SPLIB    CNAME
         PROC
         APEN      DEB1,FIN1
         USECT     TAB1SSP
         DATA,2    0
         USECT     TAB2SSP
EB1      SET       $
         TEXTE     AF(1)
IN1      SET       $
         DA        FIN1-DEB1=1
         RES       2
          ELSE
         DA        FIN1-DEB1=2
         RES       1
          FIN
          FIN
SSP      SET       NSSP+1
         CLOSE     DEB1,FIN1
         PEND
         CLOSE     NSSP
         PAGE
         SSPLIB    'INITIAL'                    1


         SSPLIB    'ECRTOR'                     2


         SSPLIB    'ORTOI'                      3


         SSPLIB    'OSTOR'                      4
```

3.6.3. Sous programme ZLIBCA2

Les sous programmes ZLIBCAL, ZLIBCA1, ZLIBCA2 sont trois entrées d'un même module métasymbol. Pour faciliter la lecture, nous présentons ZLIBCA2 comme un sous programme à une seule entrée. Il effectue deux opérations : générer la déclaration d'une référence externe, générer une instruction de branchement à un sous programme (BAL), l'adresse de ce sous programme étant la référence externe préalablement déclarée. (Le code du sous programme, correspondant à son entrée dans les tables, a été transmis par XGENCALSSPLIB dans le registre XRI1).

```
ZLIBCA2   EQU              *
*
*
          LW, XRT1         XRI1
          MI, XRT1         12
          XGENREF, PRØG,   (REG, RX1)
                           (ADW, TAB2SSP, (IBR,XRT1)), (TEST)
          STH, RX1         TAB1SSP, XRI1
          XGINSDEC, BAL,   6       ((VAL, 0)), (REG, RX1)
          B                * XRT1
```

Remarques :

Le registre XRT1 contient en fait le déplacement en octet du nom du sous programme de librairie par rapport au début de la table TAB2SSP.

3.6.4. Utilisation de XGENCALSSPLIB

Traduction de "i=x" où i est entier et x réel ; il faut appeler un sous programme de conversion : 9RTØI ; ce sous programme trouve le nombre réel à convertir dans le registre 8 et donne le résultat converti en entier dans

le registre 9. Au moment où le générateur doit générer cette instruction, nous supposerons que le nombre réel à convertir est défini par un double mot EMETTEUR, le premier mot de EMETTEUR contient le numéro de déclaration associé à x, le second la translation par rapport à cette déclaration ; RECEPTEUR est défini de manière identique, il suffit d'écrire :

```
XGINSDEC, LW, 8          ((WØRD, EMETTEUR+1)),(WØRD,EMETTEUR)
XGENCALSSPLIB            9RTØI
XGINSDEC,STW,9           ((WØRD,RECEPTEUR+1)),(WØRD,RECEPTEUR)
```

4
normes de réalisation du générateur

Avant de décrire l'organisation du générateur, il faut définir de façon précise ce qu'il produit.

Nous venons d'étudier la structure physique des modules-objets ainsi que la manière de générer une rubrique élémentaire de celui-ci. Nous allons maintenant définir l'organisation logique des informations contenues dans un module-objet.

Cette organisation est celle d'un programme écrit en langage machine équivalent au programme source. En fait, chaque instruction source sera traduite par une séquence d'instructions machines ; nous allons donc examiner la manière de réaliser un travail s'exprimant avec des instructions Civa, à l'aide du langage machine 10 070.

4.1. MANIPULATION DES ELEMENTS SIMPLES DANS LES CALCULS ARITHMETIQUES

Une compilation décompose les instructions du langage en opérations élémentaires : branchement ou appel de sous programmes, entrées-sorties et calculs arithmétiques. La traduction de n'importe quelle instruction source se ramène à une succession d'instructions de ces diverses catégories. Quelle que soit la façon d'analyser une expression arithmétique, on se ramène à des calculs élémentaires du type

$$a = b \oplus c$$

où $\oplus$ est un opérateur du type + / * -, abc désignant des variables localisées en mémoire ou dans des registres ; nous ne nous intéresserons ici qu'aux instructions de ce type.

Cette étude est évidemment importante pour la réalisation du traducteur d'expression arithmétique : elle sera aussi directement utilisée à chaque fois que l'on aura besoin de faire un calcul non explicitement prévu par le programmeur ; il suffira de prévoir, dans le compilateur, des méta-procédures décrivant chacune de ces séquences, pour qu'elles puissent être utilisées chaque fois qu'il sera nécessaire de générer un calcul élémentaire.

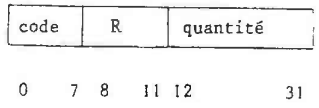
De plus, on ne peut définir, sans connaître exactement l'incidence des calculs arithmétiques, les normes de transmission de paramètres, le traitement des constantes, la traduction des expressions conditionnelles, les vecteurs de renseignements associés aux files, la gestion des registres à l'exécution, les possibilités de contrôle sur les calculs. Lorsqu'on analyse un compilateur, il est bon de commencer par étudier la réalisation des calculs arithmétiques, sans se préoccuper de la manière dont une expression arithmétique source est effectivement traduite.

Les informations contenues dans ce paragraphe n'ont souvent rien d'original, mais leur analyse est cependant assez longue et il nous a paru intéressant d'en faire une étude relativement complète pouvant s'appliquer à n'importe quel type de compilateur (sur IO 070). En particulier, dans l'étude de chaque séquence, nous avons cherché à définir les méthodes optima de réalisation ; dans certains cas, nous avons par exemple préféré une solution du type appel de sous programme de librairie à une séquence intégralement générée. Chaque séquence sera repérée par un code de trois chiffres de manière à ce que l'on puisse, par la suite, s'y reporter.

4.1.1. Adressage sur IO 070

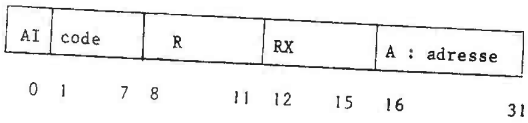
Il y a cinq modes d'adressage sur IO 070 : adressage immédiat, direct, indexé, indirect, indirect indexé et quatre types d'adresses : adresse de mot, de double mot, d'octet, de demi mot.

Une instruction à adressage immédiat a le format suivant :



La quantité est une information sur 20 bits, R indique le registre sur lequel porte l'instruction.

Les autres instructions ont toutes le même format :



Le bit 0 est à 1, s'il y a adressage indirect.

RX désigne un registre d'index ; RX ≡ 0 s'il n'y a pas indexation.

A est l'adresse de l'opérande et toujours une adresse de mot ; toutefois, et selon le code de l'instruction, on peut obtenir par indexation que l'opérande soit à une adresse d'octet de demi mot ou de double mot.

Soient :

AI une fonction telle que AI(A) = A s'il n'y a pas d'adressage indirect

= contenu de A sinon

D (déplacement)

une fonction telle que D(RX) = 0 si RX ≡ 0

= contenu de RX sinon

BA, HA, DA étant trois fonctions faisant passer d'une adresse de mot à l'adresse correspondante d'octet, de demi mot, de mot.

L'adresse de mot effective d'une instruction à adressage par mot est :

AI (A) + D (RX)

L'adresse d'octet effective lorsqu'il y a adressage par octet est :

BA (AI (A)) + D (RX)

De même, les adresses effectives pour les instructions adressage par demi ou double mot seront :

HA (AI (A)) + D (RX)

DA (AI (A)) + D (RX)

Exemple :

Si le contenu de l'adresse 32 est 20,

si le contenu du registre 3 est 2,

l'adresse désignée par l'instruction :

sera :

22 si l'adressage est par mot,

24 si l'adressage est par double mot,

21 si l'adressage est par demi mot,

3e octet de 20 si l'adressage est par octet.

4.1.2. Convention pour l'écriture des instructions

- Symboles

Nous respecterons la syntaxe de Métasymbol et, dans les exemples, nous utiliserons les mêmes symboles que dans les instructions Civa avec les conventions suivantes :

- un symbole en majuscule désigne une adresse : (nous ferons en fait la confusion entre déplacement et adresse dans la plupart des exemples) ;

- un symbole minuscule désigne un nombre ;
- un symbole grec désigne l'adresse d'une mémoire de travail.

- Désignation des registres

Un registre sera représenté par la chaîne de caractères

$$R\,[X]\;\left[\left\{\begin{matrix}I\\P\end{matrix}\right\}\right]\;[D]\;[i]$$

[ ] indique que le symbole est optionnel ; { } qu'il y a choix exclusif entre deux symboles où

X signifie registre d'index,

I signifie registre impair,

P signifie registre pair,

D signifie qu'un couple de registres R, R∪1 sont concernés par l'instruction,

$i \in \mathbb{N}$ sera utilisé pour différencier deux registres, le cas échéant.

Exemple :

RX1, RX2 sont deux registres d'index,

R1 un registre quelconque.

- Notations diverses

$\mathcal{C}_e$ représente l'ensemble des variables entières

$\mathcal{C}_r$ représente l'ensemble des variables réelles

$\mathcal{C}_{dp}$ représente l'ensemble des variables "double précision"

$\mathcal{C}_c$ représente l'ensemble des variables complexes

$\mathcal{C}_{dc}$ représente l'ensemble des variables "double complexe"

$\mathcal{C}_d$ représente l'ensemble des variables décimales

$\mathcal{C}_e$ est l'ensemble des constantes entières représentables.

Nous distinguerons une partition de $\mathcal{C}_e$ en trois sous-ensembles :

$\mathcal{C}_{e4}$ est l'ensemble des constantes représentables sur 4 bits

$(x \in \mathcal{C}_{e4} \Longleftrightarrow x \in \mathcal{C}_e \quad \text{et} \quad -8 \leqslant x \leqslant +7)$

$\mathcal{C}_{e20}$ est l'ensemble des constantes représentables sur 20 bits à l'exclusion de celles représentables sur 4 bits

$$x \in \mathcal{C}_{e20} \Longleftrightarrow \begin{cases} x \in \mathcal{C}_e \\ x \notin \mathcal{C}_{e4} \\ -2^{19} \leqslant x \leqslant 2^{19} - 1 \end{cases}$$

$\mathcal{C}_{e31} = \mathcal{C}_e - \mathcal{C}_{e4} - \mathcal{C}_{e20}$

4.1.3. Opérandes entiers

Les entiers sont rangés sur 1 mot ; les instructions de transfert seront donc de simples LW ou STW, on veillera, pour les autres opérations, à utiliser au maximum les instructions diversifiées existant pour le type entier.

4.1.3.1. Addition

Un cas doit être traité avec le maximum de précaution : les opérations de cumul ("i = i + 1" ou "a = a + expression") ; en effet, ce type d'instruction apparaît principalement dans des bouclages. Les instructions MTW et

AWM ont d'ailleurs été prévues pour cet usage.

Le tableau ci-dessous représente les diverses possibilités de traduction de a=b+c. L'addition étant commutative, nous n'avons pas représenté les configurations se déduisant par commutativité. Nous remarquerons que la reconnaissance des constantes est importante pour un traitement sur entier.

condition			séquence générée	temps moyen exécution	code réf.
$b \in \mathcal{V}_e$ $c \in \mathcal{V}_e$	$a \neq b$ et $a \neq c$		LW,R B AW,R C STW,R A	6µs	1,1,1
	$a \equiv c$ soit $a = b + a$ (identique à $a = a + c$)		LW,R C AWM,R A	4,6µs	1,1,2
$b \in \mathcal{C}_e$ $c \in \mathcal{V}_e$	$a \equiv c$ soit $a = a + b$	$b \in \mathcal{C}_{e4}$	MTW,b A	2,8µs	1,1,3
		$b \in \mathcal{C}_{e20}$	LI,R b AWM,R A	4,1µs	1,1,4
		$b \in \mathcal{C}_{e32}$	séquence identique à 1, 1, 2, B étant l'adresse de la constante		
	$a \neq c$	$b \in \mathcal{C}_{e4} \cup \mathcal{C}_{e20}$	LW,R C AI,R b STW,R A	5,5µs	1,1,5

Dans la séquence 1,1,5, il n'est pas nécessaire de distinguer le cas $b \in \mathcal{C}_{e4}$: "MTW,b R" à la place de "AI,R b".

4.1.3.2. Soustraction

Les cas sont assez semblables à ceux de l'addition.

condition			séquence générée	temps moyen d'exécution	code réf.
b et $c \in \mathcal{V}_e$	$a \neq b$ et $a \neq c$	$a = b - c$	LW,R B SW,R C STW,R A	6µs	1,2,1
	$a \equiv b$	soit $a = a - c$	LCW,R C AWM,R A	4,6µs	1,2,2
$b \in \mathcal{V}_e$ $c \in \mathcal{C}_e$			les séquences sont les mêmes que 1,1,3 ou 1,1,4		
$b \in \mathcal{C}_e$ $c \in \mathcal{V}_e$	$b \in \mathcal{C}_{e20}$	soit $a = cte - c$	LCW,R C AI,R b STW,R A	5,5µs	1,2,3
	$c \in \mathcal{C}_{e32}$		cas identique à 1,2,1		

4.1.3.3. Multiplication

Il faudra tenir compte des particularités de la multiplication sur IO 070 qui peut utiliser deux registres. Les entiers traités en Civa sont toujours sur 1 mot, on opérera donc dans un registre impair.

condition		instructions générées	temps	code gén.
$b \in \mathcal{V}_e$ et $c \in \mathcal{V}_e$		LW,RI B MW,RI C STW,RI A	9,2 µs	1,3,1
$b \in \mathcal{C}_e$	$b \equiv 2^n$ $n \neq 1$ $a = 2^n * c$	LW,R C SLS,R n STW,R A	6,3 + 0,1*n	1,3,2
	$b \equiv 2^n$ $b \in \mathcal{C}_{20} \cup \mathcal{C}_4$	LW,RI C MI,RI b STW,RI A	9,2 µs	1,3,3
	$b \equiv 2$ $a = 2 * a$	LW,R A AWM,R A	4,6 µs	1,3,4

Repérer une constante n'apporte pas un avantage fondamental : cela réalise simplement l'économie d'un mot mémoire ; seul le cas où cette constante est égale à 2^n paraît intéressant.

4.1.3.4. Division

a = b/c

condition	instructions générées	temps	code
$c \in \mathcal{C}_e$, $c = 2^n$	LW,R B SLS,R -n STW,R A	6,3 + 0,2*n	1,4,1
$b \in \mathcal{C}_{e20} \cup \mathcal{C}_{e4}$, $c \in \mathcal{V}_e$	LI,RI b DW,RI C STW,RI A	16,2µs	1,4,2
autres cas : $b \in \mathcal{V}_e \cup \mathcal{C}_{e32}$ $c \in \mathcal{V}_e \cup (\mathcal{C}_e - \{2^n\})$	LW,RI B DW,RI C STW,RI A	16,7µs	1,4,3

4.1.3.5. Opérateur "-" unaire : a=-b

Il existe une instruction permettant de charger le complémentaire d'un nombre entier dans un registre.

a = - b se traduira par :

```
LCW,R     B
STW,R     A        (1,5,1)
```

Une instruction telle que a = -(b+c) se traduira donc par :

```
LCW,R     B
SW,R      S        (1,5,2)
STW,R     A
```

Dans le cas d'une séquence telle que a = - b * c ou a = -b/c, il suffit de remplacer LW par LCW dans toutes les séquences, à partir de 1, 3, 1 jusqu'à 1, 4, 3.

4.1.4. Opérandes réels

4.1.4.1. Simple précision

On utilisera le jeu d'instructions FAS, FSS, FMS, FDS, et ici les instructions seront nettement moins diversifiées que dans le cas des nombres entiers.

- Addition : a = b+c

```
LW, R     B        (2, 1)
FAS,R     C
STW,R     A
```

- Soustraction : a = b - c

```
LW,R      B        (2, 2)
FSS,R     C
STW,R     A
```


- Multiplication :

On aura le choix entre deux séquences suivant les registres utilisés.

```
LW,RI     B        (2, 3, 1)
FMS,RI    C
STW,RI    A

LW,RPD    B        (2, 3, 2)
FMS,RPD   C
STW,RPD   A
```

Dans la séquence (2, 3, 2), le résultat se trouve en double précision, dans RPD et RPD+1. Cette formule peut être intéressante dans le cas où l'on a à traduire des instructions du type a*b + d*e, l'addition, source d'erreur de chute, se faisant en double précision.

- Division : a = b/c

```
LW,R      B        (2, 4)
FDS,R     C
STW,R     A
```

4.1.4.2. Réels double précision

Les formules sont rigoureusement analogues, mais on travaille sur des doubles mots :

- Addition : a = b+c

```
LD,RPD    C        (3, 1)
FAL,RPD   B
STD,RPD   A
```

- Soustraction : a = b-c

```
LD,RPD    B        (3, 2)
FSL,RPD   C
STD,RPD   A
```

- Multiplication : a = b*c

```
LD,RPD      B          (3, 4)
FDL,RPD     C
STD,RPD     A
```

- Division : a = a/c

```
LD,RPD      B          (3, 5)
FDL,RPD     C
STD,RPD     A
```

- Opérateur "-" unaire :

Pour les entiers simple précision, l'instruction LCW convient toujours : un réel négatif étant représenté par son complément à 2, pour les réels double précision, nous utiliserons l'instruction LCD qui effectue le même travail sur un double mot.

Exemple :

a, b, c *double precision* ;

pour traduire "a=-b×c", on déduira de (3, 4)

```
LCD,RPD     B
FML,RPD     C
STD,RPD     A
```

4.1.5. Opérandes complexes

4.1.5.1. Simples complexes

Nous utiliserons les instructions flottantes en nous ramenant à des calculs sur des nombres réels.

Notation : $a=x_a + j\ y_a$

$b=x_b + j\ y_b$

$c=x_c + j\ y_c$

Addition : $a = b + c$

$$\begin{cases} x_a = x_b + x_c \\ y_a = y_b + y_c \end{cases}$$

Deux écritures sont possibles :

```
LW,R        B          (4,1,1)
FAS,R       C
STW,R       A
LW,R        B+1
FAS,R       C+1
STW,R       A+1
```

ou, en tenant compte du fait que les complexes sont rangés dans des doubles mots :

```
LD,RPD      B          (4,1,2)
FAS,RPD     C
FAS,RPD     C+1
STD,RPD     A
```

La forme (4,1,2) est plus rapide et occupe moins de place que (4,1,1) mais celle-ci laissera plus de liberté lors d'une éventuelle optimisation.

Soustraction : nous retrouvons les mêmes écritures

```
LW,R        B          (4,2,1)
FSS,R       C
STW,R       A
LW,R        B+1
FSS,R       C+1
STW,R       A+1
```

```
LD,RPD      B          (4,2,2)
FSS,RPD     C
FSS,RPD     C+1
STD,RPD     A
```

Multiplication :

$$x_a = x_b\, x_c - y_b\, y_c$$

$$y_a = x_c\, y_b + x_b\, y_c$$

La méthode la plus simple consiste à développer en une suite de quatre produits, soit 12 instructions ; une autre solution consiste à appeler un sous programme :

```
LW,RPD      B       (4,3,1)
FMS,RPD     C
LW,RID      B+1
FMS,RID     C+1
FSS,RPD     RID
STW,RPD     A
LW,RPD      B
FMS,RPD     C+1
LW,RID      B+1
FMS,RID     C
FAS,RID     RPD
STW,RID     A+1
```

```
LD,RPD1     B       (4,3,2)
LD,RPD2     C
BAL,RL      MULCPLX
STD,RPD3    A
```

L'ordre des opérations dans 4, 3, 1 est impératif à cause du déroulement de l'instruction FMS. Le sous programme MULCPLX s'écrit de manière relativement simple, mais il prend autant d'instructions que la séquence (4,3,2) ; ou :

```
MULCPLX  LW,RPD3     RPD1
         FMS,RPD3    RPD2
         LW,RID3     RID1
         FMS,RID3    RID2
         FSS,RPD3    RID3
         FMS,RPD1    RID2
         LW,RID3     RID1
         FMS,RID3    RPD2
         FAS,RID3    RPD1
         B           *RL
```


Remarquons que n rencontres de ces cas entraînent l'écriture et l'exécution de n*12 instructions si on adopte la solution de (4,3,1), tandis que si on adopte la solution de (4,3,2), elles demandent l'écriture de 11+4*n instructions, et l'exécution de 16*n instructions (quatre instructions supplémentaires dans la séquence d'appel et un branchement indirect). Un calcul plus précis montre que l'exécution de la séquence (4,3,1) demande en moyenne 48µs (en prenant 7µs comme temps d'exécution d'une multiplication et que l'exécution de la séquence (4,3,2) demande en moyenne 58µs, soit une perte de temps de l'ordre de 20 %). C'est à l'implémenteur de choisir la solution qui lui paraît la meilleure.

Division :

Nous retrouverons le choix entre deux séquences analogues, les calculs à faire étant :

$$\alpha = x_c^2 + y_c^2$$

$$x_a = x_b\, x_c + y_b\, y_c \;/\alpha$$

$$y_a = x_b\, y_c - x_c\, y_b \;/\alpha$$

```
LW,RPD    C
FMS,RPD   RPD
LW,RID    C+1
FMS,RID   RID
FAS,RID   RPD
STW,RID   α
LW,RPD    B
FMS,RPD   C
LW,RID    B+1
FMS,RID   C+1
FAS,RID   RPD
FDS,RID   α
STW,RID   A
LW,RPD    B
FMS,RPD   C+1
LW,RID    C
FMS,RID   B+1
FSS,RPD   RID
FDS,RPD   α
STW,RPD   A+1
```

20 instructions

durée : 100 µs

```
          LD,RPD1     C          (4,4,1)
          LD,RPD2     B
          BAL,RL      ADIVCPLX
          STD,RPD3    A

DIVCPLX   LW,RPD3     RPD1
          FMS,RPD3    RPD3
          LW,RID3     RID1
          FMS,RID3    RID3
          FAS,RID3    RPD3
          STW,RID3    α
          LW,RPD3     RPD2
          FMS,RPD3    RPD1
          LW,RID3     RID2
          FMS,RID3    RID1
          FAS,RPD3    RID3
          FDS,RPD3    α
          LW,RID3     RPD2
          FMS,RID3    RID1
          FMS,RPD1    RID2
          FSS,RID3    RPD1
          FDS,RID3    α
          B           *RL
```

4 instructions + 18

durée : 10 µs + 100 µs = 110 µs

Ici, les durées sont réellement équivalentes à 10 % près et le gain de place tend vers un rapport 5 dès que l'on a plus de 4 appels ; la solution (4,4,2) semble donc s'imposer, sauf si l'on veut une optimisation des registres, la solution sous programme en utilisant 7 et l'autre 2.

4.1.5.2. Double complexe

Ces calculs se rapprochent du cas des simples complexes, mais ici, nous ne disposons plus d'instruction de chargement de doubles mots.

addition

```
LD,RPD    B    (5,1)
FAL,RPD   C
STD,RPD   A
LD,RPD    B+2
FAL,RPD   C+2
STD,RPD   A+2
```

soustraction

```
LD,RPD    B    (5,2)
FSL,RPD   C
STD,RPD   A
LD,RPD    B+2
FAL,RPD   C+2
STD,RPD   A+2
```

On traite la multiplication et la division par deux sous programmes, avec transmission des paramètres par adresse.

multiplication

```
LI,R1     A    (5,3)
LI,R2     B
LI,R3     C
BAL,RL    DCMUL
```

division

```
LI,R1     A    (5,4)
LI,R2     B
LI,R3     C
BAL,R2    DCDIV
```

4.1.5.3. Traitement de l'opérateur "-" unaire

Les règles définies sur les entiers et les réels s'appliquent pour les complexes, il faut cependant remarquer que

```
LCD,RPD   A
```

n'est pas équivalent à :

```
LCW,RPD   A
LCW,RID   A+1
```

en particulier, seule la deuxième forme est à employer dans le cas des complexes simple précision.

a = -b se traduit par :

```
LCW,RPD    B
LCW,RID    B+1
STD,RPD    A
```

si a et b sont simple précision, et par :

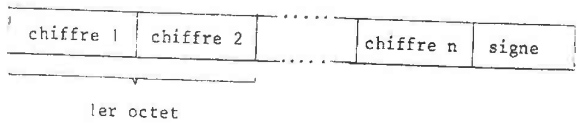
```
LCD,RPD    B
STD,RPD    A
LCD,RPD    B+2
STD,RPD    A+2
```

si a et b sont complexes double précision.

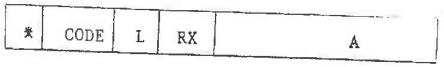
4.1.6. Opérandes décimaux

4.1.6.1. Instructions décimales sur IO 070

Un décimal compressé est code par une suite de chiffres sur 4 bits, terminée par un signe lui aussi sur 4 bits ; on ajoute éventuellement à cette chaîne un chiffre 0 de poids fort pour obtenir une chaîne d'octets :



Un décimal sera repéré par l'adresse de sa chaîne d'octets. Une instruction décimale a le format :



L'opérande est de longueur L et a pour adresse effective BA(AI(A))+ D(RX) ;

on opère sur l'accumulateur décimal (formé des registres 12, 13, 14, 15), le résultat de l'opération se trouve dans l'accumulateur.

Les instructions décimales utilisables sont :

DL	chargement décimal
DST	rangement décimal
DA	addition
DS	soustraction
DM	multiplication
DD	division
DSA	décalage arithmétique décimal

4.1.6.2. Relation entre les déclarations Civa et les instructions machines

Soit :

a *decimal pic* 9(n) V9(m) ;

on désignera par "ℓ" la quantité $\frac{((n+m) \cup 1+1)}{2}$, celle-ci représente, en fait, la longueur de la chaîne d'octets.

WA (A) représente l'adresse, en mots, de ce décimal et A son adresse, en octets.

La quantité : $\alpha = A - 4 * WA(A)$ désignera le déplacement, en octets, de de a par rapport à l'adresse du mot le plus proche.

4.1.6.3. Etude des transferts de nombres décimaux

Position du problème : étant donné les déclarations

a *decimal pic* 9 (n_a) V 9 (m_a)

b *decimal pic* 9 (n_b) V 9 (m_b)

Il s'agit de traduire a=b ou a=-b. (Nous désignerons par d_a et d_b, ℓ_a et ℓ_b les quantités d et ℓ associées à a et b).

a) a et b ont le même facteur de cadrage ($n_b = n_c$ et $m_a = m_b$) : on se ramène à un simple transfert de chaîne d'octets, et il suffit de générer :

```
LW,RI     α          (6,0,1)          (6,0,1)
MBS,RI    BA(B) - BA(A)
```

où α est défini par :

```
α  GEN,8, 24     ℓ , BA (A)
```

b) a et b n'ont pas le même facteur de cadrage : il faut alors utiliser l'accumulateur décimal et on est amené à générer la suite d'instructions :

```
LI,RX     d_b              *          (6,0,2)
DL, ℓ_b   WA(B), RX
DSA       m_a - m_b        **
LI,RX     d_a              *
DST,ℓ_a   WA(A), RX
```

* cette instruction peut être bien entendu supprimée si le déplacement est nul (ce cas peut être relativement fréquent).

** ce décalage disparaît si $m_a = m_b$.

c) affectation d'une constante à une variable : il existe une simplification possible dans certains cas, en particulier si la constante A possède moins de 7 chiffres (c'est-à-dire si elle tient sur 1 mot) dont voici deux exemples :


```
a decimal 99 V9
...
a = 12.5
```

est équivalent à :

```
LI,15     X'125C'                    (6,0,3)
LI,RX     d_a
DST,ℓ_a   WA(A), RX
```

(X'C' est le code du chiffre +)

```
a decimal  9(6)
  a = 215367
```

```
LW,15                                (6,0,4)
LI,RX     d_a
DST,4     WA(A),RX
```

et α DATA X'0215367C'

Si la constante possède de 8 à 15 chiffres, on peut utiliser une simplification analogue en utilisant les registres 14-15.

d) traitement de l'opérateur "-" unaire : a = -b

X'C' et X'D' représentent les signes + ou - (X'E' et X'F', X'A' et X'B' sont aussi respectivement tolérés, mais toute opération sur décimaux produit X'C' et X'D' comme + ou -). On pourra donc, sans problème, passer au complémentaire en ajoutant 1. (On prendra la précaution de normaliser a X'C' et X'D' lorsqu'on ne fera pas d'opération ultérieure).

Exemple :

```
a decimal 99 V 99 ;  (ℓ_a = 3)
b decimal 99 V 9  ;  (ℓ_b = 2)
```

a = -b est équivalent à :

```
DL,2      WA(B)
AI,15     1                    (6,0,6)
AND,15    M2
DST,2     WA(A)
```

où : M2 DATA X'FFFFFFFD' est un masque permettant la normalisation du signe en transformant éventuellement X'E' en X'C'.

4.1.6.4. Traitement des opérateurs classiques

Pour tous les traitements, nous nous placerons dans les cas les plus généraux, étant bien entendu qu'un décalage ou une indexation nulle ne sera pas traduit. Pour déterminer une séquence d'instructions, nous chercherons toujours à ne perdre aucun chiffre sur le résultat, c'est ce principe qui nous conduira à organiser les décalages.

Convention :

Nous utiliserons, pour toutes les formules, a = b ⊕ c où a est déclaré par $9(n_a)\ V9(m_a)$, b : $9(n_b)\ V9(m_b)$ et c : $9(n_c)\ V9(m_c)$ et ⊕ un opérateur quelconque.

Addition : a = b + c

Si $m_c \geq m_b$, on génère :

```
LI,RX     d_b
DL,ℓ_b    WA(B),RX              (6,1,1)
DSA       m_c - m_b
LI,RX     d_c
DA,ℓ_c    WA(C),RX
DSA       m_a - m_c
LI,RX     d_a
DST,ℓ_a   WA(A),RX
```


Si $m_b > m_a$, on inverse les rôles de b et c dans l'exemple ci-dessus.

Soustraction : a = b - c

Si $m_c \geq m_b$, la séquence est rigoureusement identique à (6,1,1) en remplaçant l'addition par une soustraction ; si $m_b > m_c$, il faudra modifier la séquence en remplaçant b-c par -c+b :

```
LI,RX     d_c                   (6,2,2)
DL,ℓ_c    WA(C),RX
AI,15     1
DSA       m_b-m_c
LI,RX     d_b
DA,ℓ_c    WA(B),RX
DSA       m_a-m_b
LI,RX     d_a
DST,ℓ_a   WA(A),RX
```

Multiplication : a = b × c

Il n'y a pratiquement plus de problème de décalage en considérant que le résultat calculé a m_b+m_c chiffres après la virgule ;
séquence d'instructions :

```
LI,RX     d_b                   (6,3)
DL,ℓ_b    WA(B),RX
LI,RX     d_c
DM,ℓ_c    WA(C),RX
DSA       m_a-m_b-m_c
LI,RX     d_a
DST,ℓ_a   WA(A),RX
```

Division :

Si "dec" est le nombre de positions de décalage du dividende, le nombre de

chiffres après le point décimal du résultat d'une division est :

$$m_b + dec - m_c.$$

Pour que l'opération soit la plus efficace possible, il faut que :

$$m_a = m_b + dec - m_a,$$

soit : $dec = m_a + m_c - m_b$.

Deux cas peuvent se présenter :

$m_a + m_c - m_b \geq 0$; on génère alors la séquence :

```
LI,RX      d_b                          (6,4,1)
DL,ℓ_b     WA(B),RX
DSA        m_a + m_c - m_b
LI,RX      d_c
DD,ℓ_c     WA(C),RX
SLD,12     64                  *
LI,RX      d_a
DST,ℓ_a    WA(A),RX
```

$m_a + m_c - m_b < 0$; seul le résultat sera alors décalé :

```
LI,RX      d_b                          (6,4,2)
DL,ℓ_b     WA(B),RX
LI,RX      d_c
DD,ℓ_c     WA(C),RX
DSA        m_a + m_c - m_b
SLD,12     64                  *
LI,RX      d_a
DST,ℓ_a    WA(A),RX
```

* Cette instruction a été introduite pour remettre à zéro les registres 12 et 13. Il existe une simplification intéressante : la division par une puissance de 10 car il suffit alors de remplacer la division par un décalage ; cette remarque est bien entendu valable pour la multiplication.

4.1.6.5. Remarques sur le temps d'exécution des instructions décimales

Ces instructions, par rapport aux instructions équivalentes sur des opérandes d'autre type, sont relativement lentes : le temps d'une addition décimale est de (19,5 + 0,3 * µs) où n est le nombre de chiffres de l'opérande décimal, un décalage demande 20 µs.

A titre comparatif, les instructions a=b+c et a=b*c demandent :

6µs et 9.2µs si a, b, c sont entiers
8.2µs et 12 µs si a, b, c sont réels
11 µs et 18 µs si a, b, c sont double précision
14 µs et 58 µs si a, b, c sont complexes
50 µs et 110 µs à 120 µs si a, b, c sont décimaux de même facteur de cadrage
90 µs et 110 µs à 120 µs si a, b, c sont décimaux de facteur de cadrage différent.

Il y a donc un rapport voisin de 10 entre les manipulations sur les entiers ou les réels et les calculs avec nombres décimaux, (et ici, nous nous sommes mis dans l'hypothèse arithmétique décimale cablée). Le choix du type d'une variable n'est donc jamais négligeable ; rappelons que les décimaux doivent être utilisés pour les variables sur lesquelles on fait peu de calculs et beaucoup d'éditions et dans les cas particuliers où l'on a vraiment besoin de calculs décimaux, ils doivent être absolument bannis en d'autres circonstances.

4.2. TRAITEMENT DES ELEMENTS DE FILE

Nous n'étudierons dans ce paragraphe que les files dont la taille est connue lors de la compilation (soient les files implantées en zone fixes), celles-ci étant les seules acceptées dans le sous ensemble actuellement défini.

4.2.1. Définitions diverses

Nous nous plaçons dans le cas le plus général, c'est-à-dire celui des files de structures, ces structures pouvant comporter des files. Une telle file a alors pour déclaration :

f_n *file* (ℓ_n) *struc* (...,
f_{n-1} *file*(ℓ_{n-1}) *struc* (...,
...
f_1 *file* (ℓ_1) *struc* (..., x *type simple*,...),...),...) ;

nous appellerons variable indicée généralisée (VIG) un élément dont la forme est :

x *de* $f_1(i_1)$ *de* $f_2(i_2)$ ... *de* $f_n(i_n)$.

Pratiquement, les i_k sont quelconques (en particulier, des constantes, des expressions à valeur entière, ou des variables entières) ; nous supposerons qu'il ne s'agit que de variables entières ; une expression est calculée en amont, sa valeur est stockée dans une variable de travail (qui devient un i_k) ; un indice constant revient à considérer la file comme une

structure à plusieurs champs dont on demande le $i_k^{ème}$ champ. (Nous ne nous intéressons en fait qu'à l'indexation dynamique, l'autre étant résolue à la codification).

Dans la suite du paragraphe, nous désignerons par :

i_k l'indice courant pour la file f_k (soit $f_k(i_k)$)

ℓ_k la dimension de la file f_k

t_k la taille (en mots) de la structure, élément de la file f_k
(t_k est aussi la distance en mots de deux éléments de la file f_{k-1})

A_k l'adresse d'implantation de f_k

A l'adresse d'implantation de la file f_n

d_1 la distance de l'élément x au début de la structure $f_1(i_1)$

d_k la distance du premier élément de f_{k-1} au début de l'élément $f_k(i_k)$
(soit à A_k)

Tous ces éléments, sauf i_k, sont connus à la compilation.

Exemple :

f *file* (5) *struct* (a *complexe*,
g *file* (7) *struct* (x *entier*, y *reel*, z *complexe*),
b *entier*, c *entier*, d *double precision*) ;

à b *de* f(i) sont associés :

$n = 1$, $f_1 = f$, $i_1 = i$, $r_1 = 2+4*7+1+1+2 = 34$,
$d_1 = 2+4*7 = 30$, $b_1 = 5$

à y *de* g(i) *de* f(j) sont associés :

$n \equiv 2$, $f_1 \equiv g$, $f_2 \equiv f$; $i_1 \equiv i$; $i_2 \equiv j$;

$t_1 \equiv 4$(mots) ; $t_2 \equiv 34$; $\ell_1 \equiv 7$, $\ell_2 \equiv 5$

$d_1 \equiv 1$; $d_2 \equiv 2$;

4.2.2. Calcul de l'adresse d'un élément

Nous désignerons un élément simple par x et Ax sera son adresse.

Si n = 1, alors l'adresse d'un élément est :

$$A_x = A_1 + (i_1 - 1) * t_1 + d_1.$$

D'autre part, il existe une relation entre l'adresse d'implantation de $f_k(i_k)$: (A_k), et $f_{k-1}(i_{k-1})$ (A_{k-1}) :

$$A_{k-1} = A_k + d_{k-1} + (i_{k-1} - 1) * t_{k-1}$$

d'où :

$$A_x = A_n + \sum_{k=1}^{k=n} (d_k + (i_k - 1) * t_k)$$

soit :

$$A_x = A + \sum_{k=1}^{k=n} (d_k - t_k) + \sum_{k=1}^{k=n} (t_k * i_k)$$

or la quantité $+ \sum_{k=1}^{k=n} (d_k - t_k)$ est une quantité directement connue à la compilation, que nous noterons d_x et :

$$A_x = A + d_x + \sum_{k=1}^{k=n} (t_k * i_k)$$

4.2.3. Accès à un élément courant indicé de type entier, réel, booléen

L'unité d'adressage sur 10 070 étant le mot, ces trois types étant implantés sur un mot, la formule déterminée ci-dessus va s'appliquer avec plus de facilité, en exprimant toutes les quantités en mot. Dans toutes les cas étudiés en début de chapitre, une expression source du type :

$$X_1 = X_2 \oplus X_3$$

où $\oplus$ est un opérateur arithmétique quelconque, amène à générer une série d'instructions du type :

code, registre X_i

Si X_i est un élément indicé, il suffira de remplacer ceci par :

- instructions calculant $\sum_{k=1}^{k=n} (t_k \times i_k)$, le résultat étant dans un registre d'index,
- code, registre $X_i + d_{x_i}$, registre d'index

Si $X_i = x$ *de* $f_1(i_1)$ *de*... $f_n(i_n)$, la séquence générée sera :

```
LW,RXI    I1
MI,RXI    t1                (*)
LW,RI     I2                (**)
MI,RI     t2
AW,RXI    RI
...
LW,RI     tn
MW,RI     In
AW,RXI    RI                (***)
code, registre X+dx, RXI
```

Dans le cas où la file f_1 est une file d'éléments simples, l'instruction (*) est à supprimer ; dans le cas où il n'y a qu'une seule file, la portion comprise entre (**) et (***)est à supprimer.

Exemple :

```
x file (100) entier ;
g file ( 50) structure (a entier, b reel) ;
f file ( 20) structure (a entier, h file (30)
                        structure (x entier, y reel)) ;
```

L'instruction x(i) = a *de* g(k) + y *de* h(m) *de* f(n) se traduit immédiatement en fonction de ce qui précède et de la séquence (!,!,!).

```
LW,RXI     K
MI,RXI     2
LW,R       G-2,RXI

LW,RXI     M
MI,RXI     2
LW,RI      N
MI,RI      61
AW,RXI     RI
AW,R       H +(1-2) + (!-61),RXI

LW,RXI     I
STW,R      X-!,RXI
```

4.2.4. Accès à un élément de file de type double précision

Sur IO 070, le mode d'adressage utilisé va nous amener à modifier très légèrement ce qui a été vu pour les variables implantées sur 1 mot.

Une variable double précision étant implantée sur une frontière de double mot, nous pouvons rappeler que :

Soit : f_n *file* (...) *struct* (... f_1 *file* (...) *struct* (..., a *double precision*,...)...) ;

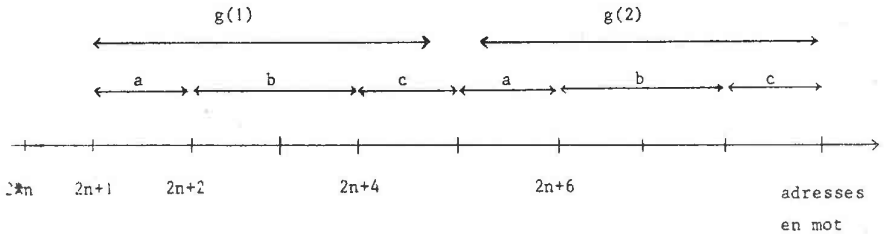
alors : t_i est pair quel que soit i

par contre, rien n'oblige les d_i à être pairs.

Exemple :

```
g file ( ) struc (a entier, b double precision, c entier) ;
```

aura l'implantation mémoire suivante :



ici, t=4 et d_b=1

Nous pouvons appliquer des séquences identiques à celles des entiers, en remplaçant les t_i par $t_i/2$, l'indexation se faisant par double mot, par contre, l'adresse de base étant une adresse de mot, le calcul de d_x reste identique ; x et y étant deux variables double précision, et g étant défini comme ci-dessus.

```
x = y + b de g (i)
```

se traduit par :

```
LD,RPD     Y
LW,RXI     I
MI,RXI     2
FAL,RPD    G-1,RXI
STD,RPD    X
```

4.2.5. Accès à un élément complexe ou double complexe

On applique les mêmes règles que pour les double précision lorsque les complexes sont manipulés globalement (ils sont alors repérés par une adresse de double mot).

On applique, pour les complexes simples, les mêmes règles que pour les réels lorsque leur champ, réel ou imaginaire, est traité individuellement ; toutefois, pour un complexe, la quantité $\Sigma t_k \times i_k$ est calculée une seule fois pour les deux champs.

Exemple :

```
a, b complexe  ; f file () struc(c complexe, d reel , e entier) ;
a = b + c de f(i)
```

se traduit par :

```
LD,RPD     B
LW,RXI     I
MI,RXI     4
FAS,RPD    F-4,RXI
FAS,RID    F-3,RXI
STD,RPD    A
```

ou par :

```
LW,RXI     I
MI,RXI     2
LD,RPD     F-4,RXI
FAS,RPD    B
FAS,RID    B+1
STD,RPD    A
```


4.2.6. Accès à un élément de file de type décimal

Deux cas se présentent :

- Les décimaux d'une file sont implantés à n'importe quelle adresse d'octet (non systématiquement sur une frontière de mot), la solution la plus simple consiste alors à calculer $A+d_x$ en octet, ainsi que les $t_k * i_k$

Exemple :

```
a file (50) decimal  9V99 ; b, c decimal  9V99 ;
b = c + a (I) ;
```

se traduit par :

```
LW,RXI     I          }
MI,RXI     2          }      t1* i
AI,RXI     BA(A)-2    }
DL,2       0,RXI      }
LI,RX      2
DA,2       C,RX                (*)
DST,2      B                   (*)
```

* on suppose que B et C sont implantés chacun sur le même mot, en fait on a B ≡ C

- Les décimaux sont implantés systématiquement à une adresse de mot, la quantité $A+d_x$ peut alors être calculée en mot et figurer directement dans l'instruction.

Exemple :

```
a file (50) struct  (x decimal 9V99, y entier) ;
                    b, c decimal  9V99 ;
b = c + x de a (i)
```

La séquence ne diffère de la précédente que par le chargement de x *de* a (i) :

```
LW,RXI      I
MI,RXI      8
DL,2        A-2,RXI
...
```


4.3. TRAITEMENTS EN MODE INDIRECT

C'est le mode de traitement que nous utiliserons pour la manipulation des paramètres formels et des files dont la taille n'est pas connue à la compilation. Dans ce cas, un élément ou une file n'est connue que par son adresse.

4.3.1. Variables simples non décimales

Le traitement des variables indirectes de type réel, entier, double précision, ne pose aucun problème, toutes les instructions vues précédemment pouvant utiliser l'adressage indirect.

Exemple :

```
procedure proc (x) x entier ; a entier
x = x + a
```

Une sauvegarde des paramètres est faite en tête de procédure et l'adresse X contient l'adresse du paramètre effectif (à l'exécution).

A partir de la séquence (1,1,2), on déduit immédiatement :

```
LW,R        A
AWM,R       *X
```

Pour les complexes, la solution le plus simple consiste à associer deux mots à un complexe paramètre formel : le premier contient l'adresse de la partie réelle, le second celui de la partie imaginaire, ceci pour éviter de refaire ces calculs à chaque utilisation du complexe.

procedure proc (c) c *complexe* ; a, b *complexe* ;

a = b + c

se traduit alors par :

```
LD,RPD     C
FAS,RPD    *B
FAS,RID    *B+1
STD,RPD    A
```

4.3.2. Variables décimales

Deux cas se présentent :

4.3.2.1. Le facteur de cadrage du décimal n'est pas connu à la compilation

Il se pose ici un problème important : les instructions décimales !0 070 contiennent le facteur de cadrage de façon explicite, il va falloir pourtant réaliser des chargements décimaux d'un nombre dont on connaît la taille à l'exécution.

Etant donné un décimal D tel que la longueur soit contenue dans L, il existe deux méthodes :

a) construire une instruction

```
LW,RX      D
LW,R       L
SLS,R      20
AWM,R      s+1
LD,0       0,RX
```

à l'exécution, cette dernière instruction deviendra LD,ℓ 0,RX

b) utiliser une instruction EXU

Rappel du fonctionnement de l'instruction EXU : cette instruction permet d'exécuter une instruction se trouvant n'importe où en mémoire, comme si cette dernière instruction était exécutée à l'emplacement de la directive EXU.

Exemple :

```
       MTW,1      A
       EXU        B
       MTW,1      C
       ...
  B    MTW,1      D
       MTW,1      E
```

On exécutera dans l'ordre : "MTW,1 A" puis "MTW,1 D", enfin "MTW,1 C" (et non "MTW,1 E").

Dans le cas d'une instruction BAL, c'est l'adresse de l'instruction qui suit l'instruction EXU qui est sauvegardée dans le registre de liaison :

```
       ...
       EXU        B
 ETI   ...
       ...
 B     BAL,6      SSP
       ...
 SSP   ...
       B          *6
```

L'instruction B *6 provoquera un retour à l'adresse ETI et non à B+1.

Le chargement d'un décimal indirect se traite alors comme suit :

```
       LW,RX      D
       LW,RX1     L
       EXU        TABLE,RX
```

où table est définie par :

```
TABLE  LD,1      0,RX
       LD,2      0,RX
       ...
       LD,15     0,RX
```

On traitera de la même manière les autres instructions décimales.

Exemple :

```
module mod x decimal pic 99V999  y decimal pic 9V9 ;
                egal (x, y) ;
procedure egal (z, t) z, t decimal ;
```

si (Z) = BA(X), (Z+1) = ℓ_x, (Z+3) = m_x

alors : z = t se traduit par :

```
LW,RX     Z+1
LW,RX1    Z
EXU       TABLEDL,RX
LW,R      LT+3
SW,R      Z+3
DSA       *R
LW,RX     T+1
LW,RX1    T
EXU       TABLEDST,RX
```

4.3.3. Manipulation des éléments de files

Seule l'adresse de la file est connue, la formule d'adressage reste encore valable, mais les éléments d_x, A, i_k peuvent être inconnus, le calcul de $\sum_{k=1}^{k=n} (t_d * i_k)$ diffère très peu du précédent si l'on remplace les MI par des MW (il faudra alors prendre la précaution de diviser le résultat par 2 si l'on adresse des doubles mots). On a intérêt à calculer

une fois pour toutes A+d_x

Exemple :

```
x file struct (a entier, b entier) ;
proc (a de x) ;
procedure proc (ℓ) ℓ file entier ;
```

si (L) = X - 1

(LT)= 2

ℓ(I)= 1

se traduit par :

```
LI,R      1
LW,RX     I
MW,RX     LT
STW,R     *L,LT
```

4.3.4. Transmission des paramètres d'une procédure ou d'un module

La transmission des paramètres se fait par nom, seule l'adresse d'un paramètre est effectivement transmise. Le type du paramètre sera aussi communiqué de façon à pouvoir introduire des vérifications.

L'instruction :

```
proc (parametre 1, ..., parametre n) ;
```

sera traduite par :

```
BAL,RL    PRØC
DATA      parametre 1
...
DATA      parametre n
```

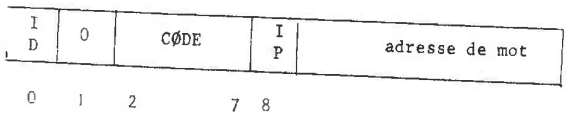
Dans une procédure, à chaque paramètre formel sera associé un ou plusieurs mots permettant de manipuler facilement le paramètre effectif.

Lorsqu'un paramètre effectif est une expression arithmétique, celle-ci est calculée et le résultat est rangé dans une mémoire de travail qui constituera le vrai paramètre effectif.

4.3.4.1. Paramètres compatibles avec Fortran CII

Pour unifier les spécifications, les paramètres effectifs correspondant à des objets manipulés par Fortran auront les spécifications requises par celui-ci.

Un paramètre effectif est représenté sur 1 mot :



ID indique que l'adresse effective du paramètre est contenue dans l'adresse de mot spécifiée ;
IP indique que le paramètre est une constante ;
CØDE identifie le type du paramètre

entier : 1
réel : 2
réel double précision : 4
complexe : 8

double complexe : 16
booléen : 32

A chaque paramètre formel, on associera un mot contenant l'adresse du paramètre effectif pour tous ces types sauf pour les complexes auxquels deux mots seront associés : l'un contient l'adresse de la partie réelle, l'autre, la partie imaginaire.

4.3.4.2. Paramètre effectif de type propre à Civa

- Octet ou caractère : un paramètre effectif est représenté par :

Le mot associé au paramètre formel contiendra l'adresse d'octet du paramètre effectif.

- Décimal : un nombre décimal est caractérisé par :

son adresse,
sa longueur (ℓ),
le nombre de chiffres après le point décimal (m).

Un paramètre décimal sera représenté sur 1 mot :

ou

Le mot spécifié contient alors ℓ, m et l'adresse d'octet avec la même disposition que pour un paramètre simple. On l'utilisera dans la procédure à l'aide de trois mots :

adresse d'octet
ℓ
m

- Paramètre de type file

Dans cette application, nous ne traiterons que les files d'éléments simples ; un paramètre de type file sera représenté sur 2 mots :

mot 1

0	1	0	1	taille actuelle

ou

1	1	0	1	adresse de mot contenant la taille

le 2ème mot représentera l'adresse et le type du premier élément de la file suivant les formats définis plus haut. Pour pouvoir être manipulé commodément dans les calculs, un tel paramètre se présentera sous la forme suivante :

mot 1	taille actuelle	
mot 2	adresse premier élément	
mot 3	adresse premier élément - t	
mot 4	ℓ	décimaux seulement
mot 5	m	

t représente la taille d'un élément, le mot 3 est utilisé pour tous les accès par indexation, le mot 2 pour les traitements séquentiels ou pour les manipulations globales.

4.4. REALISATION DES CONTROLES

Les problèmes à résoudre sont :

- détecter la réalisation d'un événement,
- initialiser et modifier un contrôle,
- libérer suspendre et reprendre un contrôle,
- traduire la séquence de récupération,
- faire des suspensions, libérations et reprises implicites.

4.4.1. Bloc de contrôle de variable

Une variable contrôlable aura plusieurs états et plusieurs informations lui seront attachées pendant l'exécution ; pour chaque variable contrôlable, il sera réservé un bloc de trois mots dit bloc de contrôle de variable (BCV) ; si x est une variable contrôlable, nous désignerons dans ce paragraphe son BCV par BCV(x) ; BCV1, BCV2, BCV3(x) désignent un des mots du BCV.

4.4.1.1. Rôle du BCV1 dans la détection d'un événement

Lorsqu'une variable a été déclarée contrôlable, une action peut être entreprise sur cette variable à chaque affectation, peut varier suivant les instructions de contrôles faites sur elle et peut être nulle si la variable est suspendue ou libérée.

Cette action "variable" peut être caractérisée par une instruction : branchement à un sous programme si une action doit être faite à partir du moment où l'on a fait une instruction *quand* ou instruction vide ; l'instruction vide, la plus rapide est une instruction notée NØP (c'est en fait une instruction LCFI,0 0). Le premier mot du BCV contiendra l'une ou l'autre de ces instructions, il suffira donc de faire suivre chaque affectation à une variable contrôlable, d'une instruction EXU dont l'adresse désignera le BCV1 de la variable affectée.

Exemple :

```
module m ;
        i entier controlable ; j, k, entier ;
        ...
        i = j + k
```

sera traduit par :

```
LW,R    K
AW,R    J
STW,R   I
EXU     BCV1(i)
```

Si i est libéré, on ne fera rien, ou plutôt on exécutera un EXU suivi d'un NØP, soit une perte de temps de 2.5 µs, ce qui est une perte faible sur le temps total de l'exécution, si les variables contrôlables sont en nombre raisonnable.

Si i est contrôlée, on exécutera :

```
BCV1(i)     BAL,RLC     CØNTRØLE(i)
```

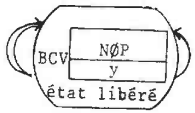

CØNTRØLE(i) sera directement l'adresse de la séquence de récupération si l'événement est systématique, ou un test suivi de la séquence de récupération.

4.4.1.2. Utilisation de BCV2 pour les opérations explicites : *suspendre*, *reprendre*

Les suspensions et les reprises sont des instructions qui se rencontreront le plus souvent dans une séquence de récupération, on a donc intérêt à ce qu'elles soient le plus rapide possible.

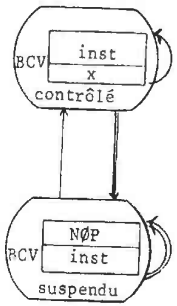
Lorsqu'une variable est suspendue, à chaque affectation de celle-ci, l'action à entreprendre est nulle, BCV1 doit donc contenir NØP ; lorsque l'instruction *reprendre* sera exécutée, il faudra reprendre le contrôle antérieur et BCV2 servira à mémoriser celui-ci.

Si, du graphe des états d'une variable contrôlée, nous extrayons le sous graphe dont les arcs sont réduits aux instructions *suspendre* et *liberer*, nous obtenons :



suspendre ⟹

reprendre ⟶



Pour que l'exécution de *reprendre* soit la plus rapide possible, il faut qu'elle soit indépendante de l'état dans lequel on se trouve afin d'éviter un test : pour passer de l'état suspendu à contrôlé, il suffit de faire un transfert de BCV2 dans BCV1 ; pour que cette opération soit indépendante de l'état, il faut que BCV2 contienne NØP, si l'état est libéré et l'instruction à exécuter, si l'état est contrôlé, ainsi :

x ≡ inst et y ≡ NØP

- pour exécuter *suspendre*, il suffit alors de ranger NØP dans BCV1 ;

- pour exécuter *liberer*, il suffit aussi, quel que soit l'état, de ranger NØP dans BCV1 et BCV2 ;
- pour exécuter *quand*, il faut ranger l'instruction à exécuter dans BCV1 et BCV2.

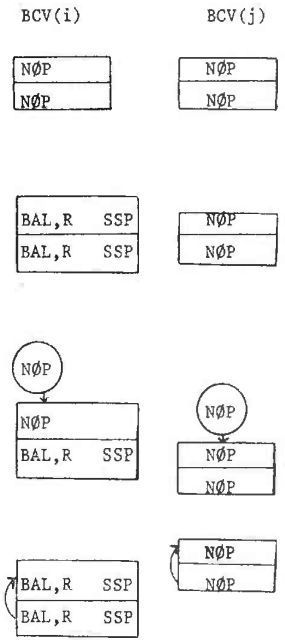
Exemple : évolution des BCV pendant l'exécution

```
module m ;
    i,j entier controlable ;

quand i faire suspendre j ;
    j = 3 ; i = 1 ;
    reprendre j ;
    fqd ;

i = 3 ;

    suspendre i(implicite) ;
    suspendre j ;
    j = 3 ;
    i = 1 ;
    reprendre j ;
    reprendre i (implicite) ;
```



4.4.2. Connexions entre l'événement et sa séquence de récupération

4.4.2.1. Définition d'un accumulateur

A chaque type simple utilisé en Civa, on associe un registre particulier dit accumulateur ; quand, à l'exécution, on manipulera une variable d'un certain type, on se servira en général de son accumulateur comme registre de travail. De même, l'accumulateur sera utilisé pour transmettre des paramètres à un sous programme de librairie ; si 9 est l'accumulateur entier, 8 l'accumulateur réel, "x=i" où i est entier et x réel se traduit par :

```
LW,9     I
BAL,6    9ITØR
STW,9    X
```

Nous utiliserons l'existence de l'accumulateur pour simplifier le traduction des séquences de récupération, et en particulier, dans le cas des événements conditionnels, les tests préparatoires. Pour cela, nous imposerons la présence de la valeur de la variable contrôlable dans l'accumulateur correspondant à son type, avant d'exécuter l'instruction EXU. Cette contrainte est en fait très faible puisque dans la plupart des cas, le résultat d'un calcul se trouve dans l'accumulateur, que la variable affectée soit contrôlable ou non.

4.4.2.2. Evénement systématique

Le problème est relativement simple : quel que soit l'état de la variable contrôlable, l'action de récupération doit être entreprise et elle se

comporte alors comme un sous programme classique ; la seule précaution à prendre est alors de sauvegarder l'adresse de retour dans un mot. En fait, comme nous le verrons, plusieurs autres renseignements doivent être sauvegardés, et l'on définit un bloc de séquence de récupération BSR dont le premier mot contient l'adresse de retour.

Remarquons qu'il peut être intéressant de particulariser trois séquences qui se réduisent à une seule instruction.

- Comptage simple : *quand* i *faire* j=j+1 *fqd* ;

  j=j+1 se traduit par MTW,1 J

c'est cette instruction qui sera placée dans BCV1 et BCV2 lors de l'exécution de l'instruction *quand*.

- Cumul : sur une variable contrôlable entière ;
  quand i *faire* a = a + i *fqd* ;

la variable contrôlable se trouvant dans l'accumulateur, i=x+y se traduit par :

```
LW,R     X
AW,R     Y
STW,R    I
EXU      BCV1(i)   ⟶   AWM,R    A
```

- Affectation : de la valeur de la variable contrôlable à une variable de même type non contrôlable ou suspendue
  quand i *faire* k=i *fqd*

la séquence de récupération se traduit alors par

```
STW,R    K
```

4.4.2.3. Evénement conditionnel

Il est impossible de traduire par une seule instruction un test suivi d'une action, l'instruction contenue dans le BCV sera donc toujours un BAL de manière à sauvegarder le compteur ordinal, donc l'adresse de retour.

Si BAL,RLC SSPTEST

est l'instruction contenue dans BCV1, SSPTEST a alors la forme suivante :

SSPTEST évaluation de la condition
branchement sur condition fausse à l'adresse contenue dans RLC, séquence de récupération proprement dite ;
elle est analogue à celle d'un événement systématique.

Il est alors important d'optimiser l'évaluation, nous nous baserons sur deux faits : la variable contrôlable se trouve dans l'accumulateur au moment où l'on entre dans SSPTEST, dans la plupart des traitements, le code de condition caractérise le signe du résultat, donc de la variable contrôlable à la fin d'une affectation ; il n'est pas modifié par une instruction de rangement ou une instruction BAL. Nous imposerons donc deux règles concernant la détection des contrôles et donc la traduction des affectations :

- la variable contrôlable se trouve dans l'accumulateur et en mémoire,
- le code de condition caractérise le signe de celle-ci.

On peut alors exécuter une instruction EXU BCV1.

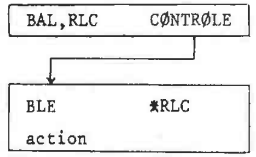
La détection des événements les plus fréquents se traite alors très rapidement.

Exemple :

quand i > 0 *faire* action *fqd* ;

L'instruction "i=3" se traduit par :

```
LI,R     3
STW,R    I
EXU      BCV1(I) ——————→
```



Le temps perdu lorsque "action" n'est pas entreprise est alors de 5,2µs, un événement du type "variable::constante" ou "variable::variable" se traduit de manière aussi rapide :

quand i > t *faire* action *fqd* ;

donnera :

```
CØNTRØLE  CW,R     T
          BLE      *RLC
          action
```

4.4.3. Opérations implicites sur les variables contrôlables

4.4.3.1. Initialisation

Une variable contrôlable est initialement libérée et deviendra contrôlée lorsqu'on exécutera une instruction *quand* ; il faut donc initialiser les BCV et cette initialisation se fait en tête de module pour les variables d'un module, la première fois que l'on utilise une classe pour les variables de celle-ci.

4.4.3.2. Suspensions et reprises implicites

Celles-ci se produisent en début et en fin de séquence de récupération. Lorsque plusieurs événements ont la même séquence de récupération, le BSR associé à celle-ci mémorisera plusieurs informations (en particulier l'adresse du BCV associé à l'événement qui a provoqué la récupération). Une suspension implicite est générée séparément pour chaque BCV ; par contre, en fin de séquence, le BSR repérera le BCV associé à l'événement, cause de la récupération, donc celui qu'il faut reprendre.

4.4.3.3. Libérations implicites

Rappelons que la durée de vie maximum d'un contrôle est celle de l'unité, module ou classe, dans laquelle figure l'instruction *quand*. Ceci ne pose aucun problème pour les variables contrôlables appartenant à l'unité car elles n'existent plus en dehors de cette unité ; il faut cependant, en fin de module, libérer implicitement les variables contrôlables appartenant à une classe utilisée ; le même problème se pose lorsqu'une classe cesse de vivre et que celle-ci déclare des contrôles sur des variables utilisées.

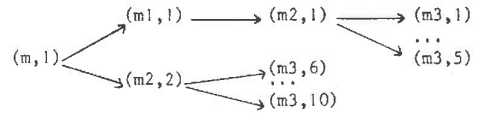
Pour chaque module ou classe, le compilateur générera une table des BCV pour lesquels une libération sera peut-être à faire (TBCVM). Le troisième mot du BCV servira à indiquer quels sont les contrôles à libérer.

Considérons le graphe pour lequel l'ensemble de définitions est constitué par les couples (nom de module, ordre d'appel du module) et la relation "appelle".

Exemple :

```
module m ;          module m1 ;        module m2 ;
      m1 ;                m2 ;         pour chaque i de 1 a 5 pas 1,
      m2 ;          fin module ;       faire m3 fp ;
fin module ;                           fin module ;
```

pour cette application, ce graphe est :



ce graphe est une arborescence ; à chaque module, pendant l'exécution, on associe son niveau "instantané" dans cette arborescence. Lorsqu'on exécute une instruction *quand*, le niveau du module dans lequel on se trouve est rangé dans BCV3 ; lorsqu'on exécute une instruction *liberer*,

on range 0 dans BCV3 ; en fin de module, on examine la table TBCVM et tous les BCV tels que "BCV3 = niveau du module" sont libérés.

Le calcul du niveau du module est simple : une variable NIV de la classe système est initialisée à 0, dans le module de commande ; à chaque entrée dans un module, on exécute :

```
MTW,1    NIV
```

et à chaque sortie :

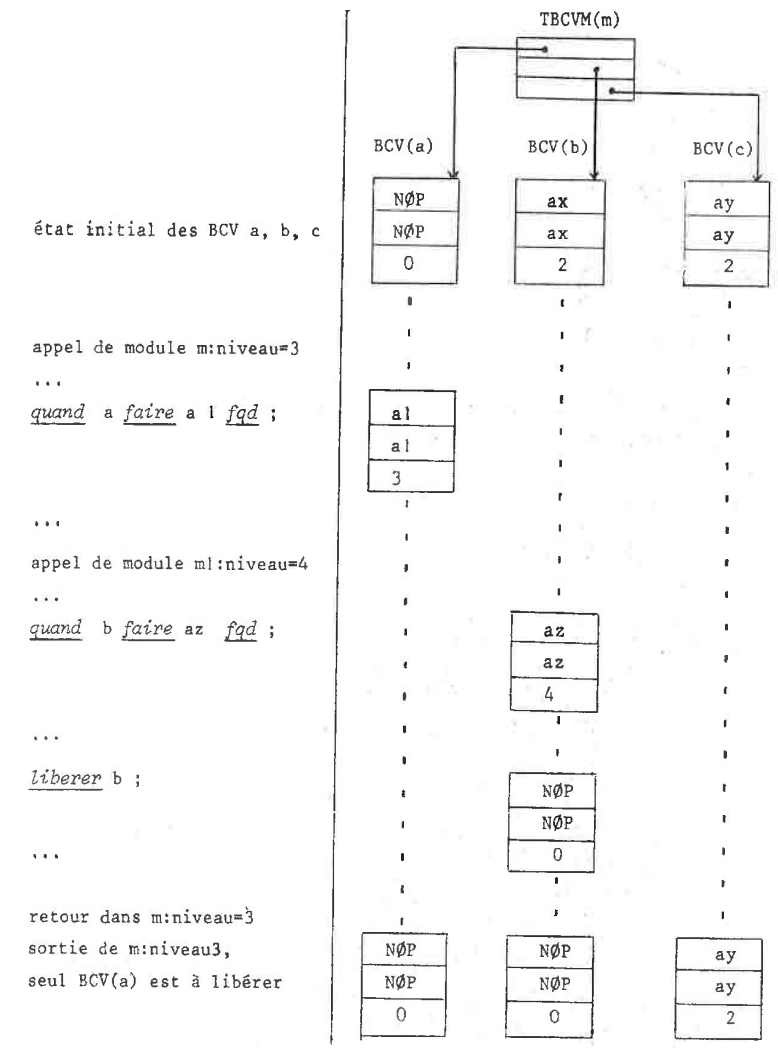
```
MTW,-1   NIV
```

Exemple :

```
module m ;
        utilise c1 ;
        ...
        quand a faire a1 fqd ;
        ...
        quand b faire a2 fqd ;
        ...
        quand c faire a3 fqd ;
        ...
        fin module ;

classe c1 ;
        a, b, c entier controlable ;
        ...
        fin classe ;
```


La figure ci-dessous illustre l'évolution des BCV au cours d'une exécution possible :



4.4.3.4. Présence d'une instruction *sortir* dans une séquence de récupération

Soit :

```
module m ;
        quand anomalie faire sortir fqd ;
        m1 ;
        m2 ;
fin module ;
```

Lors de l'exécution de m1 ou m2, une anomalie peut se produire : ceci implique une sortie du module m. Il importe alors de libérer les contrôles contenus dans m, mais aussi dans m1 ou m2, et éventuellement, dans les modules appelés par m1 ou m2. Pour cela, on tiendra à jour une pile des TBCVM, tout module contenant une TBCVM empile, en début de traitement, l'adresse de celle-ci et la retire lorsqu'on en sort.

La table TBCVM aura en fait l'allure suivante :

mot 1	niveau de module
mot 2	nombre de BCV : n
mot 3	...
...	
mot 2+n	adresse BCV

Pour exécuter *sortir*, il suffit d'explorer, depuis le sommet de la pile, tous les TBCVM, telles que le contenu du mot 1 soit supérieur ou égal au niveau du module dans lequel on trouve l'instruction *sortir* et de libérer les BCV qui doivent l'être.

4.4.4. Remarques

L'introduction des contrôles amène une seule perturbation pour celui qui ne les utilise pas : la tenue à jour du niveau des modules, soit une perte de 4,5 µs par appel de module ; de plus, on ne fera les opérations de tenue à jour des TBCVM que pour les modules dans lesquels le programmeur aura écrit une instruction *quand* portant sur une variable extérieure au module. Ce service est donc relativement économique malgré la possibilité d'opérations implicites.

Les contrôles détectés par le système se traitent de façon analogue.

5
organisation du générateur

Le générateur est l'un des principaux modules de l'application compilation Civa. Son analyse est faite en utilisant le langage d'analyse Civa ; il est donc très proche d'un programme Civa, toutefois, certaines instructions en particulier l'appel des procédures de génération sont en langage libre, l'adaptation en étant très simple ; de même, nous avons introduit certains types (octet) qui ne font pas partie des types de base Civa mais sont manipulés facilement par Métasymbol.

Nous ne donnerons pas ici une analyse complète du générateur, nous nous limiterons à certains modules qui nous ont paru significatifs, et qui montrent bien l'articulation existant entre les classes et les modules de l'application. Signalons que l'analyse du traducteur de table de décision fait l'objet d'un autre rapport de thèse [3].

5.1. INTERFACE DU GENERATEUR AVEC LE SYSTEME DE COMPILATION CIVA

5.1.1. *module* compilateur ;

¢ ce module accepte une suite de textes sources, modules ou classes pour produire des modules objets ¢
utilise ractab
¢ informations locales au compilateur permettant la communication entre les principaux modules appelés par le compilateur ¢ ;
utilise interface compilateur module de commande ;
pour chaque module ou classe à compiler *faire* ;
initialisation compilateur ;
codifieur ; ¢ ce module est décrit dans [4] ¢ ;
générateur ;
post compilateur ; *fp*
fin module ;

5.1.2. Classe de l'application compilation Civa

5.1.2.1. Application compilation Civa

taille identificateur *entier* = 10;
identificateur *type file* (max=taille identificateur) *caractere* ;
¢ description d'un enregistrement du fichier descriptif des classes et des modules ¢ ;
matricule *type demi mot*
¢ à chaque unité de nomenclature est attribué, lors de sa création, un matricule qui servira à la repérer ¢ ;
element de nomenclature *type*,
nature *structure* (matclé *matricule*)
type *code sur 4 bits* (module ¢ 0001 ¢,
classe ¢ 0010 ¢,
meta-module ¢ 0100 ¢,
classe de commande ¢ 1000 ¢),

etat *code sur 4 bits* (source,
compile,
edite)),
renseignements divers *structure*(
nombre de procedures declarees *entier*,
taille *structure* (
zone constante *entier*,
zone variable *entier*),
init classe *booleen*
¢ si l'élément de nomenclature est une classe,
elle doit être initialisée si init = *vrai* ¢
modules de librairie appeles *file* (128) *bit*,
liste classes utilisees *file* ( ) *matricule*,
liste modules appeles *file* ( ) *matricule*) ;
fin classe ;

5.1.2.2. *classe* interface compilateur module de commande ;

liste d'entree *file* ( ) *identificateur* ;
module ou classe a compiler *de* liste d'entree ;
mode *code* (mise au point, normal) ;
niveau de severite *entier* < 16
¢ le niveau de sévérité est 0 lorsqu'il n'y a aucune erreur, de l'ordre de 3 pour des erreurs mineures apparemment récupérées, 7 lorsque l'exécution est hasardeuse mais possible, 12 lorsqu'a priori il paraît inutile de passer à l'exécution ¢ ;

5.1.2.3. *$module* remplissage

¢ ce méta-module sera utilisé pour le calcul d'éléments de structure ; soit

a *struct* (x *entier*, z *reel*, t *entier*) ;

on écrira :

$remplissage (a, (t,18), (z,z1+z2), (x,3)) ;

à la place de :

t *de* a = 18 ;
z *de* a = z1 + z2 ;
x *de* a = 3 ;

¢ nous rappelons que *af* est une méta-liste permettant de repérer les paramètres utilisés dans la référence à un méta-module ¢

$pour chaque $x *de af* (2:) *$faire* ;
$x (1) *de af* (1) $ = x (2) ;
$fp ;
$fin module ;

5.1.3. *classe* descriptif de l'application ;

¢ cette classe décrit l'organisation du fichier descriptif des classes et des modules, elle contient aussi les procédures d'accès à ce fichier ¢

fichier descriptif des classes et des modules
ensemble ( ) *element de nomenclature*, *cle* matcle ;
dictionnaire *file* () *structure* (nom *identificateur*,
cle *matricule*) ;
¢ les éléments de dictionnaire sont classés par ordre alphabétique suivant le champ nom ; un élément du dictionnaire permet d'associer à un identificateur son matricule ¢
rang dictionnaire *file* ( ) *entier* ;
¢ cette file permet d'associer à chaque matricule le rang de l'élément associé dans dictionnaire ¢
cle donnee *matricule* ; nom cherche *identificateur*

5.1.3.1. *procedure* recherche nom ;
indice *entier* ;
indice = rang dictionnaire (cle donnee) ;
nom cherche = nom *de* dictionnaire (indice) ;
fin proc ;
nom donne *identificateur* ; cle cherchee *matricule* ;
nom absent *booleen controlable* ;

5.1.3.2. *procedure* recherche cle ;

¢ recherche dichotomique dans dictionnaire ¢
min, mil, max *entier* ;
max = *taille actuelle* (dictionnaire) ;
min = 1 ; mil = (max + min)/2 ;
condition

max - min ≤ 2	F		V	
nom *de* dictionnaire (mil) < nom cherché	V	F	-	
nom *de* dictionnaire (max) = nom cherché	-	-	V	F
nom *de* dictionnaire (min) = nom cherché	-	-	-	V
clé cherchée = clé *de* dictionnaire (min)			1	
clé cherchée = clé *de* dictionnaire (max)				1
max = mil		1		
min = mil	1			
mil = (max + min)/2	2	2		
retable	3	3		

sinon nom absent = *vrai* ; *anomalie* = *vrai fsi* ;
fin proc
fin classe

5.1.3.3. Exemple d'utilisation du fichier dictionnaire

"éditer la liste des modules appelés par le module "m""

```
module edition ;
        utilise descriptif de l'application ;
        element element de nomenclature ;
        nom donne = "m" ;
        recherche cle ;
        element = fichier descriptif des classes et des modules
                  (cle cherchee) ;
pour chaque matricule de liste modules appeles faire
        cle donnee = matricule ;
        recherche nom ; ¢imprimer (nom donne) ;
fp ;
fin module ;
```

5.1.4.

```
module initialisation compilateur ;
¢ recherche de l'élément suivant à compiler ¢
        utilise descriptif de l'application, ractab ;
        mat matricule ;
        nom donne = module ou classe a compiler ;
        recherche cle ;
        mat = cle trouvee ;
        enrdescr de objet a compiler =
                fichier descriptif des classes et des modules
                (mat) ;
        taille zone constante = 0 ; taille zone variable = 0 ;
fin module ;
```


5.1.5.

```
module post compilateur ;
        utilise ractab, gestion des modules et procedures
        utilise descriptif de l'application ;
        ¢ ce module a pour rôle principal la composition de
        l'enregistrement, correspondant à l'objet venant d'être
        compilé, dans le fichier descriptif des classes et des
        modules, mais en fait il s'agit du remplissage de "objet
        à compiler" décrit dans ractab ¢
        etat = compile ;
        zone constante = taille zone constante ;
        zone variable  = taille zone variable  ;
        pour chaque nom module de      modules appeles,
                    mat de modules appeles faire ;
                nom donne = nom module ;
                recherche cle ;
                mat = cle trouvee ;
        fp ;
        si type de objet a compiler = classe ;
                alors mise de renseignements complémentaires
                      dans le fichier des tables des symboles
                      des classes ;
                fsi ;
fin module ;
```

5.1.6. *classe* ractab ;

¢ cette classe contient les informations communes aux différents modules du compilateur, nous ne présentons ici que les informations utiles à la compréhension des modules présentés ¢

utilise interface compilateur module de commande ;

objet a compiler *struct* (enr descr *element de nomenclature*
, nom *identificateur*) ;

taille zone constante *entier* ;

¢ contient 0 lorsqu'on entre dans le générateur ¢

taille zone variable *entier* :

¢ à la fin de la codification, elle contient la taille de la zone allouée aux variables déclarées ¢

procedures declarees *file* ( ) *identificateur* ;

classes utilisees *file* ( ) *struct* (nom *identificateur*
, numero dec *entier*
, indic init *booleen* ¢ indique que la classe est à initialiser ¢)

¢ indicateurs divers ¢

nombre de variables controlables declarees *entier* ;

TBCVM necessaire *booleen* ;

¢ il est positionné à *vrai* dès que le codifieur rencontre la définition d'un événement portant sur une variable extérieure à l'objet à compiler (voir paragraphe 4.4.3.) ¢

fin classe ;

5.2. GENERATEUR, ANALYSE DE LA CHAINE CODEE

5.2.1. *module* générateur ;

utilise ractab, gentab, definition chaîne codee ;

initialisation generateur ;

corps ;

terminaison generateur ;

procedure corps ;

selection

code *de* rubrique	
fin chaîne codée	*sortir*
début procédure	traitement début procédure ; *retable*
fin procédure	traitement fin procédure ; *retable*
sortir	traitement sortir ; *retable*

¢ remarquons que l'on ne fait explicitement aucune progression dans chaîne codée ; dans chaque module de traitement, on traite la rubrique en cours et l'on passe à la suivante (ceci permet de traiter, en une seule fois, des suites d'expressions arithmétiques par exemple) ¢

fin proc ;

fin module ;

5.2.2. *classe* definition chaîne codee ;
code chaîne codee *code* (fin chaîne codee,
debut procedure,
fin procedure,
appel module ou procedure,
appel procedure,
expression arithmetique,
controle utilisateur,
si,
alors,
sinon,
fsi,
quand,
finquand,
table de decision,
suspendre,
reprendre,
liberer,
sortir) ;
chaîne codee *file struct*
(code *code chaîne codee*,
commande *file* ( ) *mot*) ;
rubrique *de* chaîne codee ;
¢ chaque commande de la chaîne codée a, en fait, une structure particulière, celle-ci sera décrite dans les classes associées à chaque type d'instruction. Si x est un code chaîne codée, dans le module "traitement x" (ou dans une classe appelée par ce module), on trouvera une déclaration du type :
commande x *struct* (...) ;
et l'instruction :
commande x = commande *de* rubrique *de* chaîne codée ¢
¢ contrôle utilisateur est une commande qui figure à la suite d'une affectation portant sur une variable contrôlable ¢
fin classe ;

5.2.3. *classe* gentab ; *utilise* definition chaîne codee
¢ cette classe est utilisée par tous les modules du générateur ¢
¢ elle contient en particulier toutes les procédures de génération ¢
emplacement *type* (nature *code* (declaration, reference en avant
, absolu)
, resolution *code* (octet, demi-mot, mot,
double mot)
, numero *entier*
¢ numéro de déclaration ou de référence en avant ¢
, translation *entier*
¢ translation suivant résolution par rapport à l'adresse de la déclaration spécifiée ¢) ;
emplacement generalise *type* (localisation *emplacement*,
adressage indirect *booleen*,
registre d'index *entier*) ;
compteur emplacement programme *emplacement* ;
compteur emplacement variable *emplacement* ;
dernier numero de declaration *entier*
¢ repère le dernier numéro de déclaration attribué ¢ ;
indicateur procedure *struct* (etat *code* (actif
¢ on est en train de générer des instructions appartenant à une procédure ¢, inactif)
, repère *entier* ¢ numéro d'ordre de la procédure qu'on est en train de traiter ¢);
adresse retour proc *emplacement*
¢ adresse de retour de la procédure qu'on est train de générer ¢ ;
procedure prog compteur (valeur) valeur *entier* ;
translation *de* compteur emplacement programme =
translation *de* compteur emplacement programme + valeur ;
fin proc ;
procedure prog compteur v (valeur) valeur *entier* ;
translation *de* compteur emplacement variable =
translation *de* compteur emplacement variable + valeur ;
fin proc ;

adresse retour *emplacement* ;
pile generateur *file* ( ) *file* ( ) *mot* ;
element de pile *de* pile generateur ;
¢ cette pile sera utilisée pour le traitement des instructions pour chaque, si, quand, pour lesquelles des imbrications sont possibles ¢
¢ un élément de pile contient le contexte associé à l'une de ces instructions ¢
fin classe ;

5.2.4. *classe* classe systeme utilisable a l'execution ;
¢ cette classe décrit l'organisation de la classe système, telle qu'elle se présente lorsqu'on exécute un module directeur. Pour simplifier les problèmes d'interface avec le système SIRIS 7, lorsque, dans un module de commande, on rencontre un appel de module "m", on demande en fait l'exécution de :
module directeur ;
initialisation de la classe systeme ;
m ;
operations diverses telles que fermeture, sauvegarde de la classe systeme ;
fin module ; ¢
¢ cette classe contient les localisations des variables qui constitueront la classe système ¢
numero declaration constantes systeme=3 ¢ nom : $:SYS ¢ ;
numero declaration variables systeme=4 ¢ nom : $$:SYS ¢ ;
¢ dans tout objet à compiler, il faudra déclarer les références externes $:SYS et $$:SYS et leur associer les numéros de déclaration 3 et 4 ; ceci est naturel car la section de contrôle associée à un module aura pour numéro de déclaration 0, la définition externe nom du module 1, la référence externe $ nom du module 2 ¢
¢ abréviation des définitions précédentes ¢
ndcs *comme* numero declaration constante systeme ;
ndvs *comme* numero declaration variable systeme ;

niveau module *entier*
¢ translation par rapport à $$:SYS de l'adresse de la variable contenant le niveau du module en cours (utilisé pour les libérations implicites des contrôles) ¢ ;
pile des TBCVM *entier*
¢ adresse du pointeur de la pile contenant les adresses des TBCVM ¢ ;
NØP *entier*
¢ translation par rapport à $$:SYS d'un mot contenant NØP ¢ ;
¢ liste des BCV moniteur :
les BCV moniteurs ont 4 mots au lieu de 3, le quatrième mot contient l'adresse de la séquence de récupération standard ¢
¢ cette liste est constituée par une méta-machine, (c'est en fait une table construite par Métasymbol) ¢
$ debut table BCV $ = ¢ fixé à l'assemblage ¢ ;
$ index table $ = début table BCV ;
$ *procedure* definition BCV (nom BCV) ;
nom BCV *entier* = $ index table ;
$ index table $ = index table + 4 ;
$ *fin proc*
$ definition BCV (BCV entree module) ;
$ definition BCV (BCV sortie module) ;
...
anomalie *entier* = $ index table ;
$ definition BCV (BCV erad) ;
...
erreur simple *entier* = $ index table ;
$ definition BCV (BCV ihlecr) ;
...
erreur catastrophique *entier* = $ index table ;

5.2.5. *module* initialisation generateur ;
¢ ce module réalise toutes les initialisations nécessaires ; pratiquement, chaque traitement a une incidence dans ce module, ceci explique la disparité des actions entreprises.

utilise ractab, gentab ; suite *emplacement* ;
initialisation module objet ;
reservation en zone variable ;
si type *de* objet a compiler = classe
alors calcul init classe *fsi* ;
decision ;

type *de* objet a compiler =	module		classe	
init classe	-		V	F
mode = mise au point	V	F		
initialisation des BCV	9	7	4	
premier rubrique *de* chaîne codee	1	1	7	3
recuperation des parametres du module	5	5		
sauvegarde adresse retour	2	2	3	
reservation des pointeurs procedures			1	1
recherche initialisation classe	8	6		
generer EXU translation:BCV entree module, declaration: 4	7			
transmission nom procedure	6			
si TBCVM necessaire *alors* empil TBCVM *fsi*	4	4	5	
generer MTW,1 translation:niveau module, declaration: 4	3	3		
generation des constantes	12	12	2	2
demande de reference en avant:suite	10	10		
generer B emplacement : suite	11	11		
definition de reference en avant : suite	13	13		
generer BAL,RL adresse retour			6	

¢ Le module terminaison générateur a une structure rigoureusement analogue ¢

fin module ;

5.2.6. Initialisation des classes à l'exécution

Certaines classes, celles qui contiennent par exemple des déclarations de variables contrôlables doivent être initialisées à l'exécution ; la préédition de liens interdit les initialisations statiques, celles-ci devront donc être dynamiques ; pour réaliser cela, il faut connaître le moment où l'on utilise une classe pour la première fois. Dans ce but, à toute classe demandant une initialisation, on associera un mot qui existera pour chaque classe, pendant toute l'exécution d'un module directeur ; ce mot sera repéré par une définition externe : $¢ nom de la classe. Il contient -1 tant que la classe n'est pas utilisée, on y ajoute +1 à chaque entrée dans un module qui l'utilise et -1 à chaque sortie. Il est ainsi facile pour un module de savoir si on commence à utiliser une classe : il suffit que le compteur associé à la classe passe par 0.

Deux modules du compilateur génèrent les instructions nécessaires à cette réalisation, l'un se préoccupe de ce qu'il faut faire en tête du module, l'autre en fin de module.

```
5.2.6.1. module recherche initialisation classes ;
            utilise gentab, ractab ;
            entier i ; i=dernier numero de declaration +1 ;
            pour chaque x de classes utilisees
                    tel que indic init de x faire
                    generation de REF   ##, nom de x ;
                    generation de MTW,1 declaration:i,
                            translation : 0 ;
                    generation de BCR,3 $+2 ;
                    generation de BAL,RL adressage indirect,
                            translation : 0 ;
                               declaration : numero de x ;
                            ¢ le premier mot d'une classe contient
                            l'adresse de la procédure d'initialisa-
                            tion ¢ ;
                    i = i + 1 ;
         fp ;
      fin module ;
```

```
5.2.6.2. module recherche terminaison classes ;
      ¢ on considère que, en général, toute classe à initialiser com-
      porte une procédure de fermeture ; un seul indicateur suffit
      donc et le traitement ressemble au traitement précédent ¢ ;
            utilise gentab, ractab ;
            entier i ; i=numero dec de dernier de classes utilisees+1 ;
            pour chaque x de classes utilisees
                        tel que indic init de  x faire ;
                    generation de MTW,-1  declaration:i,
                            translation : 0 ;
                    generation de BCS,2   $ + 2 ;
                    generation de BAL,RL  adressage indirect,
                            translation : 1 ;
                               declaration : numero de x ;
         fp ;
      fin module ;
```



```
5.2.7.  module initialisation module objet ;
            utilise gentab ;
            utilise ractab ;
      ¢ ce module déclare les définitions et références externes asso-
      ciées au module à compiler et aux classes qu'il utilise, il dé-
      finit la définition externe associée au module à compiler ¢
            initialisation procedures de generation
            ¢ appelle ZDBØI et initialise certains compteurs ¢ ;
            generation de DEF   nom de objet a compiler ;
            generation de REF   "$", nom de objet a compiler ;
            generation de REF   "$ :SYS"
                                ¢ nom de la partie constante de la
                                classe système ¢ ;
            generation de REF   "$$:SYS"
                                ¢ nom de la partie variable de la
                                classe système ¢ ;
            ¢ ces 4 défininitions précédentes auront pour numéro de
            déclaration 1, 2, 3, 4, quel que soit le module ou la
            classe à compiler ¢ ;
            i entier ;
            pour chaque x  de classes utilisees, i de 5 pas 2 faire ;
            generation de REF   nom de x ;
            generation de REF   "$", nom de x ;
            numero de x = i ;
            fp ;
            generation de ØRG  0 ;
            generation de definition de DEF de numero 1 ;
            ¢ définit la définition externe nom de objet à compiler ¢ ;
      fin module ;
```

5.3. ORGANISATION DES MODULES GERANT LES CONTROLES

Quatre instructions principales gèrent les contrôles *quand*, *suspendre*, *liberer*, *reprendre*. A chacune de ces instructions correspondent des commandes dans la chaîne codée et donc des modules de traitement. Il faut néanmoins remarquer que les contrôles prennent une part importante en début et en fin de module (initialisation et libération implicites). Les instructions *liberer*, *suspendre*, *reprendre* sont des instructions extrêmement simples à compiler, nous ne les décrirons pas ici, (elles ressemblent au module initialisation des BCV).

Les seules instructions posant quelques problèmes sont *quand*, *fin quand*, et *sortir*.

5.3.1. *classe* controle ;
utilise gentab, ractab ;
variable controlable *type* (localisation *code* (
systeme
, local
, classe utilisee).
repere *entier sur 3 octets*,
adresse BCV *redefinit* repere *struct* (
, numero dec *octet*
, translation *entier*)) ;
¢ le type *variable controlable* permet de repérer un BCV, on utilisera le champ repère pour un BCV système ou appartenant au module ou à la classe en cours de traitement, dans le cas d'un BCV système, "repère" désigne la translation du BCV par rapport au début de la classe système, il

est alors facile de savoir à quel type d'événement on a affaire ; dans le cas des classes utilisées, adresse BCV permet d'adresser directement le BCV ¢ ;
BCV base *emplacement* ;
adresse TBCVM *emplacement* ;
table TBCVM *file* ( ) *emplacement*
¢ sert à constituer la table TBCVM qui sera générée en fin de module ¢ ;
¢ variables de liaison entre traitement, quand, fin quand : cet ensemble d'informations appartient en fait à la pile du générateur ; en effet, les *quand* pouvant être imbriqués, il est nécessaire de constituer une pile des informations locales à chacun d'eux ¢
suite *emplacement*
debut sequence recuperation *emplacement*
commande quand *file* ( ) evenement *struct*
(nature *code* (systematique,
conditionnel),
, b c v *variable controlable*
, condition *commande expression arithmetique*) ;
fin classe ;

5.3.2. *module* traitement quand ;
utilise controle ;
¢ on commence par générer un saut de manière à sauter la séquence de récupération ¢
demande reference en avant : suite ;
generer B emplacement : suite ;
debut sequence recuperation = compteur emplacement programme :
commande quand = rubrique *de* chaîne codee ;
nouveau rubrique *de* chaîne codee ;
recherche recuperation simple ; ¢ ce module examine la séquence de récupération pour savoir si l'on ne se trouve pas dans l'un des cas étudié au paragraphe 4.4.2.2. ; dans ce cas, il rend le contrôle après avoir traité la commande fin quand ¢ ;
fin module ;

5.3.3. *module* traitement fin quand ;

utilise controle, gentab ;

¢ ce module a 3 fonctions principales : il termine la séquence de récupération, pour chaque événément, il détermine les instructions qui lui sont propres et il génère les instructions de garniture de BCV ¢

¢ terminer la séquence de récupération ¢

adresse BSR *emplacement* ;

demander une zone de travail, mettre l'emplacement de cette zone dans adresse BSR ;

¢ appel du sous programme de librairie qui, à la fin de l'exécution d'une séquence de récupération, fait les restaurations éventuelles, reprend le contrôle implicitement suspendu en tête de séquence de récupération, et redonne le contrôle à l'instruction qui suit celle qui a provoqué la récupération ¢

generer LI,7 emplacement : adresse BSR ;

generer appel du sous programme de librairie 9FINSQR ;

pour chaque evenement *faire* ;

initialisation sequence de recuperation ;

¢ suivant la nature de l'événement conditionnel ou systématique et sa localisation système, local, appartenant à une classe, on produit une séquence d'initialisation qui fait éventuellement un test et remplit le BSR ¢ ;

si localisation ≠ local *alors* mise a jour table TBCVM *fsi*;

fp ;

definition de reference en avant : suite ;

pour chaque evenement *faire* ;

activation BCV ;

fp ;

fin module ;

5.3.4. *module* initialisation des BCV ;

¢ ce module doit être appelé en tête d'un module ou dans la procédure d'initialisation d'une classe ¢ ;

utilise gentab ;

utilise ractab ;

utilise classe systeme utilisable a l'execution ;

utilise controle ;

n *entier* ; n = nombre de variables controlables declarees ;

si n = 0 *alors sortir fsi* ;

si n = 1 *alors* ;

generer LI,8 0 ;

generer LW,9 declaration:ndcs, translation:NØP ;

generer STW,9 emplacement:BCV base ;

generer STW,9 emplacement:BCV base + 1 mot ;

generer STW,8 emplacement:BCV base + 2 mots ;

sinon ;

generer LI,7 n ;

generer LI,8 emplacement:BCV base ;

generer l'appel du sous programme de librairie 9INITBCV

fsi ;

fin module ;

5.3.5. *module* empil TBCVM ;

¢ ce module est appelé par initialisation générateur lorsqu'une TBCVM doit être déclarée ¢

¢ réservation TBCVM, celle-ci est considérée comme une référence en avant ¢

demande reference en avant : adresse TBCVM ;

generer LI,8 emplacement : adresse TBCVM ;

generer PSW,8 translation : pile des TBCVM

declaration : 4 ;

generer LW,8 translation : niveau module, declaration:4;

generer STW,8 emplacement : adresse TBCVM ;

fin module ;

5.4. TRAITEMENT DES PROCEDURES

Dans un module ou une classe, on distinguera deux sortes de procédures. Les procédures déclarées dans les classes utilisées sont connues par le codifieur qui peut communiquer dans la chaîne codée tous les renseignements concernant celles-ci ; par contre, le codifieur est incapable de distinguer un appel de procédure locale à l'objet compilé, ou un appel de module : en effet, les procédures peuvent être déclarées n'importe où. On trouvera donc deux commandes dans la chaîne codée : appel de procédure, appel de module ou procédure. Le codifieur constitue toutefois une liste des procédures déclarées. On peut donc, pendant la génération, savoir s'il s'agit d'un appel de module ou de procédure.

Le traitement des procédures présente une seconde particularité lorsqu'elles sont déclarées dans une classe : on peut remarquer que, lorsqu'on modifie un module, on fait en général des changements d'instructions, rarement des apports ou des modifications sur les déclarations de variable. La modification d'une classe est coûteuse car elle oblige à recompiler les modules qui l'utilisent ; c'est pourquoi nous avons décidé, pour rendre les recompilations moins fréquentes, de placer, en tête de classe, des pointeurs de procédures. Les procédures d'une classe sont alors adressées indirectement par les modules qui l'utilisent. Ainsi, la modification du texte d'une procédure n'implique aucune compilation secondaire.

5.4.1. *classe* gestion des modules et procedures ;
utilise expression arithmetique, gentab ;
repere procedure *file* (*taille actuelle*
(procedure declaree)) ;
struct (etat *code* (deja defini,
non defini)
adresse *emplacement*) ;
modules appeles *file* ( ) *struct* (nom *identificateur*,
numero dec *entier*) ;
commande appel module ou procedure
structure (nom appele *identificateur*,
, partie parametre
commande expression arithmetique) ;
liste parametre *file* ( )
struct (parametre effectif *mot*
, numero dec *entier*) ;

5.4.2. *module* traitement appel module ou procedure ;
utilise gentab, gestion des modules et procedures ;
commande appel module ou procedure =
commande *de* chaîne codee ;
calcul parametre
¢ ce module qui utilise les services du traducteur d'expression arithmétique remplit liste parametre ¢
trouve *booleen* ; trouve = *faux* ; i *entier* ; i = 0
pour chaque nom *de* procedures declarees,
tant que non trouve *faire*
i=i+1 *si* nom = nom appele *alors* trouve = *vrai fsi*
si trouve *alors*
si etat=non defini *alors* demande reference en avant :
adresse *de* repere procedure (i)
¢ cette référence sera satisfaite lorsqu'on rencontrera la commande procedure associee ¢
fsi ;

```
        generer BAL,RL emplacement:adresse de repere procedure(i)
        sinon appel module fsi
        pour chaque element de liste parametre faire
        generer :parametre effectif de element avec le numero de
             declaration : numero dec de element
        fp
procedure appel module
        pour chaque module de modules appeles
             tant que non trouve faire
        i=i+1 si nom de module=nom appele alors trouve=vrai
        fp
        si trouve alors
             generer BAL,RL translation : 0
                  declaration:numero dec de modules appeles(i)
             sinon
             generer REF  nom appele ;
        taille actuelle(modules appeles) =
                  taille actuelle (modules appeles) + 1;
        nom de dernier (modules appeles) = nom appele ;
        numero dec de dernier (modules appeles) =
                  dernier numero de declaration ;
             generer BAL,RL translation : 0
                  declaration:dernier numero de declaration ;
fsi ;
fin procedure  ;
fin module  ;
```


5.5. TRADUCTION DES EXPRESSIONS ARITHMETIQUES

La traduction des expressions arithmétiques se fait en deux temps : analyse et génération. Le module d'analyse prend ses informations dans la chaîne codée et produit une suite de commandes ; ces commandes sont en fait des instructions destinées à une machine à 3 adresses et analysées par un module qui, pour chaque instruction 3 adresses, génère une suite d'instructions IO 070.

Cette manière de procéder amène beaucoup de souplesse et permet de séparer les problèmes d'analyse et de génération.

5.5.1.
```
module traitement expression arithmetique ;
        utilise expression arithmetique, ractab, gentab,
             definition chaîne codee ;
        si mode = mise au point ;
        alors traduction expression arithmetique simple ;
        sinon traduction expression arithmetique optimisee fsi ;
fin module  ;
```

5.5.1.1.
```
module traduction d'expression arithmetique simple ;
        utilise expression arithmetique ;
        commande expression arithmetique =
             commande de chaîne codee ;
        analyse commande expression arithmetique ;
        generation simple ;
        nouveau rubrique de chaîne codee ;
        si code de rubrique=expression arithmetique
             alors recommencer fsi ;
```

```
        si code de rubrique=contrôle utilisateur alors cont ut ;
        ¢ ce module génère une  instruction EXU   BCV ¢
        sinon sortir fsi ;
        nouveau rubrique de chaîne codee ;
        si code de rubrique=expression arithmetique
            alors recommencer ;
            sinon sortir fsi ;
fin module ;
```

5.5.1.2.

```
module traducteur   expression arithmetique optimisee ;
        utilise expression arithmetique ;
        analyse suite de commande ;
        optimisation ;
        generation e a ;
        si code de rubrique = contrôle utilisateur
            alors cont ut sinon sortir fsi
        nouveau rubrique de chaîne codee ;
        si code de rubrique = expression arithmetique
            alors recommencer fsi ;
    procedure analyse suite de commande ;
        analyse expression arithmetique ;
        nouveau rubrique de chaîne codee ;
        si code de rubrique = expression arithmetique
            alors recommencer fsi ;
    fin procedure  ;
    fin module  ;
```


5.5.2.

```
classe expression arithmetique ;
        utilise ractab, gentab, definition chaîne codee ;
        commande expression arithmetique file ( )
                        struct (octet 0 octet
                                , octet 123  3 octets) ;
        operateur chaîne codee type (champ octet = FF,
                                , code operateur code(+,-,/...)
                                , priorite octet) ;
        operande chaîne codee type (nature code (constante,
                                memoire de travail,
                                variable directe,
                                variable indirecte),
                                type code (caractere,
                                , booleen
                                , entier
                                , reel
                                , reel double precision
                                , complexe
                                , double complexe
                                , decimal)
                                numero dec octet,
                                translation demi mot) ;
        ¢ dans le cas où nature=constante, type=entier ou carac-
        tère décimal, la valeur peut se trouver dans translation ¢
        operateur booleen ; operande booleen ;
        procedure test operateur operande ;
        si octet 0 = FF
            alors operateur = vrai ; operande = faux ;
            sinon operande  = vrai ; operateur = faux fsi ;
        fin proc ;
        operande 3a type (code operande 3a code (emplacement,
                                constante,
                                registre,
                                travail)
```

```
                    loc emplacement generalise
                    valeur redefinit loc entier,
                    registre redefinit loc entier)
                    numero redefinit loc entier)) ;
instruction 3 adresses type (code instruction
                    struct (nom code (+entier, +reel ...
                                , affectation mot ...),
                            accrcpt entier ≤ 16
                            ¢ accumulateur utilise pour le
                            recepteur ¢
                    accemt g entier ≤ 16,
                    accemt d entier ≤ 16,)
                    , operandes struct (recepteur
                                (recepteur operande 3a
                                , emetteur gauche operande 3a
                                , emetteur droit operande 3a))
    base mt ea      emplacement
    ¢ adresse de la première mémoire de travail utilisable ¢
    taille sup mt ea entier
    ¢ taille de la zone occupée par les mémoires de travail ¢
    pile operateur file ( ) operateur chaîne codee ;
    pile operande  file ( ) operande chaîne codee ;
    operateur sommet de pile operateur ;
    chaîne 3 adresses file ( ) instruction 3 adresses
    procedure init chaîne 3a ;
        taille actuelle (chaîne 3 adresses) = 0
    fin proc ;
fin classe ;
```


5.5.3. *module* analyse commande expression arithmetique

Ce module analyse "commande expression arithmétique" : il utilise un algorithme classique d'analyse d'expression arithmétique parenthésée avec priorité des opérateurs, utilisant deux piles (pile opérateur et opérande) ; il est décrit dans [8].

Chaque fois qu'un opérateur doit être appliqué, il y a production d'une ou plusieurs instructions à 3 adresses. Lorsqu'une expression mixte est rencontrée, on applique des règles de priorité sur les types des opérandes pour demander des conversions (une conversion est aussi une instruction particulière du code à 3 adresses).

Dans cette phase, il n'y a aucune attribution précise de mémoire de travail, à chaque requête de mémoire de travail, on attribue simplement un numéro d'ordre que nous désignerons par α ; celui-ci étiquette, en fait, un sommet non terminal de l'arborescence associée à l'expression arithmétique.

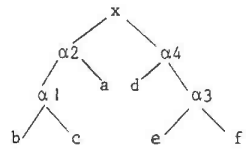
Exemple :

Etant donné l'expression arithmétique

```
x = (b + c)/a - d * (e + f) ;
```

ce module produit

```
b + c        α1
α1/a         α2
e + f        α3
d * α3       α4
α4 -α2        x
```



Remarque :

Dans les instructions à 3 adresses, le code de l'instruction indique le type des opérandes ; par exemple, il existe les codes +entier, +réel, +complexe...

5.5.4. Module génération simple ;

A chaque code d'instruction à 3 adresses est associée une procédure qui produit une suite d'instructions IO 070 telles qu'elles ont été décrites dans le chapitre 4. Le rôle de génération simple est d'attribuer un emplacement, registre ou mémoire de travail, à chaque αi et d'appeler les modules de génération d'instructions IO 070. Il procède de manière très simple, mais rapide : avant d'attribuer une valeur à un αi récepteur, on examine la commande suivante : si αi est réutilisé à gauche lorsque l'opération n'est pas commutative, ou si αi est réutilisé à gauche ou à droite dans le cas contraire, on donne à αi la valeur du registre accumulateur associé au type de l'instruction.

Nous pouvons donner de ce module la description suivante :

5.5.4.1. *module* generateur simple ;
utilise expression arithmetique ;
init mt ;
temporaire *file* (*taille actuelle* (chaîne 3 adresses)-1)
operande 3a ;
¢ cette file repérera les αi : du point de vue de l'implémentation proprement dite, elle est inutile et il suffit de considérer tous les récepteurs de la chaîne 3 adresses sauf le dernier pour obtenir cette file ¢ ;

si taille actuelle (chaîne 3 adresses) ≠ 1
alors
pour chaque inst 2 *de* chaîne 3 adresses (2:)
¢ instruction à examiner ¢
, inst 1 *de* chaîne 3 adresses (1:)
¢ instruction à compléter ¢
, i *de* 1 *pas* 1 *faire*
donner une valeur à αi en examinant inst 2 ;
remplacer les αi par leur emplacement dans inst 1
fp ; *fsi*
generation 3 adresses
¢ ce module génère des instructions IO 070 à partir de chaque commande de la chaîne 3 adresses ¢ ;

5.5.4.2. *procedure* remplacer les αi par leur emplacement dans inst 1 ;
t *file* (3) *operande 3a* *redefinit* operandes *de* inst 1 ;
pour chaque op *de* t, *telque* code oprde 3a=travail *faire*
op = temporaire (numero *de* op) ;
si rang (op) > 1 ¢ emetteur ¢
et code oprde 3a *de* op = emplacement
alors liberer mt *fsi*
fin procedure

5.5.4.3. *procedure* donner une valeur a αi en examinant inst 2 ;

decision ;

¢ emetteur gauche=αi ¢ code oprde 3a *de* emetteur gauche *de* inst 1 = travail *et* numero *de* emetteur gauche *de* inst 1 = i	F	V	F	
¢ emetteur droit =αi ¢ code oprde 3a *de* emetteur droit *de* inst 2 = travail *et* numero *de* emetteur droit *de* inst 2 = i	F	F	V	
operation instruction 1 commutative			V	F
demande mt	1			1
temporaire (i) = mt	2			2
temporaire (i) = registre, accemt g *de* inst 2		1	2	
echanger emetteur gauche emetteur droit			1	

5.5.4.4. ¢ gestion des mémoires de travail ¢

¢ les mémoires de travail sont gérées en pile, la manipulation des αi dans l'analyse imposant que le dernier demandé soit le premier utilisé ¢

mt *operande 3a*

taille zone mt *entier* ; increment *entier* ;

procedure init mt

¢ remplissage (mt, (code oprde 3a, emplacement)
, (loc, base mt ea))

taille zone mt = 0 ;

increment = 0 ;

procedure demande mt

translation *de* loc *de* mt = translation *de* loc *de* mt
+ increment

calcul increment ;

¢ cette procédure examine le code instruction pour déterminer la longueur de l'incrément suivant, soit la taille de αi ¢

taille zone mt = taille zone mt + increment

si taille zone mt > taille sup mt ea

alors taille sup mt ea = taille zone mt *fsi*

fin procedure ;

procedure liberer mt ;

calcul decrement ;

translation *de* loc *de* mt =

translation *de* loc *de* mt - decrement ;

fin proc ;

fin module ;

5.5.4.5. Exemple

Si nous reprenons l'expression analysée précédemment, (tous les éléments sont supposés entiers)

x = (b + c)/a - d * (e + f)

b + c ⟶ α1

α1 /a ⟶ α2 ⟹ α1 = accumulateur entier

e + f ⟶ α3 ⟹ α2 = mémoire de travail T1

d * α3 ⟶ α4 ⟹ α3 = accumulateur entier et instruction inversée

α2 - α4 ⟶ x ⟹ α4 = mémoire de travail T2

la séquence effectivement générée sera :

```
LW,9    B
AW,9    C
DW,9    A
STW,9   T1
LW,9    E
AW,9    F
MW,9    D
STW,9   T2
LW,9    T1
SW,9    T2
STW,9   X
```

la séquence optimale sera :

```
LW,11   B
AW,11   C
DW,11   A
LCW,9   E
SW,9    F
MW,9    D
AW,9    11
STW,9   X
```

Le temps d'exécution de la séquence optimale est de 28.9 µs ; la séquence effectivement générée dure 35.3 µs, la séquence générée sans aucune simplification aurait été de 43.5 µs. Cet algorithme est donc assez loin de l'algorithme optimal, mais il apporte quand même un gain appréciable pour un prix de compilation très faible : il est donc très acceptable en mode mise au point.

Remarquons que si les opérandes sont réels, les temps comparés sont 43, 49, 57µs, si l'écart absolu est le même, l'écart relatif diminue nettement.

5.5.5. Module optimisation

Ce module n'est pas défini dans la version actuelle du compilateur Civa. Il est en cours d'analyse mais nous pouvons cependant en donner les idées directrices.

L'opération la plus importante est la suppression des calculs redondants. C'est la raison pour laquelle, en mode normal, on analyse plusieurs expressions arithmétiques avant d'optimiser, les redondances étant fréquentes d'une instruction source à la suivante. Les éléments α introduits dans l'analyse permettent de repérer facilement un calcul, et la plus grosse partie du travail sera obtenu en égalant les α repérant un calcul identique.

Exemple :

```
a = b + x * (y + t)
c = d + x * (y + t)
y + t ⟶ α1
x * α1 ⟶ α2
b + α2 ⟶ a
y + t ⟶ α3
x * α3 ⟶ α4
d + α4 ⟶ c
```

optimisation ⟶

```
y + t ⟶ α1
x * α1 ⟶ α2
b + α2 ⟶ a
α3 ≡ α1
x * α3 ⟶ α4 devient
x * α1 ⟶ α4
⟹ x4 ≡ α2
d + α2 ⟶ c
```

La seule précaution à prendre durant cette étape est de contrôler la validité des simplifications.

Exemple :

```
a = b + x  * (y + t)
x = a + b
c = d + x  * (y + t)
```

```
y + t ⟶ α1
x *α1 ⟶ α2
b +α2 ⟶ a
a + b ⟶ x
y + t ⟶ α3
x *α3 ⟶ α4
d +α4 ⟶ c
```

Ici, la seule simplification possible est $\alpha 3 \equiv \alpha 1$. Ces simplifications étant faites, il sera nécessaire de modifier l'ordre des calculs afin d'optimiser le nombre de mémoires de travail ; pour cela, deux remarques s'imposent :

- un αi utilisé deux fois comme opérande gauche d'une instruction doit nécessairement être stocké dans une mémoire de travail ; en effet, lors de l'exécution de la première instruction, le registre qui le contient change de valeur.

Exemple :

a + b ⟶ αi
αi/ x ⟶ y
...
αi/ t ⟶ z

```
LW,9     A
AW,9     B
STW,9    αi
DW,9     X (9 est
           modifié)
...
LW,9     αi
DW,9     T
```

- on peut éviter de stocker temporairement un αi qui apparaît comme opérande gauche et droit de deux instructions (et l'un peut ainsi obtenir une solution optimale avec deux registres).

Exemple :

x + y ⟶ xi
αi/b ⟶ αk
c /αi ⟶ αp
⟹

```
LW,9     X
AW,9     Y
LW,!1    C
DW,!1    9
STW,!1   αp
DW,9     B
STW,9    αk
```


Cette réorganisation des calculs doit aboutir à une décomposition en sous calculs, chaque sous calcul a la forme d'un arbre et ne contient que des calculs de variables de même type. Sur chaque sous calcul, on fait une optimisation de registres (si les opérandes sont d'un type simple, entier ou réel), en utilisant les algorithmes de SETHI et ULLMAN par exemple [13].

Il est important de noter que ce module ne fait que des transformations sur chaîne à 3 adresses, il est donc très facilement interchangeable.

6
aide à la mise au point
contrôle de l'execution

6.1. INTRODUCTION

6.1.1. Vie d'une chaîne de traitement au sein d'une application

Au sein d'une application, une chaîne de traitement vit trois étapes fondamentales : analyse, programmation et exploitation. Pendant les deux dernières, nous trouverons plusieurs phases caractéristiques :

- mise au point d'un module ;
- regroupement de plusieurs modules pour former une chaîne ;
- mise en exploitation d'une chaîne de traitement.

Remarquons qu'il ne s'agit pas d'une division en sous étapes mais, au moment de la programmation, ces phases peuvent intervenir à différents moments pour différents modules, chaînes ou sous chaînes.

6.1.1.1. Mise au point d'un module

Dans cette phase, il s'agit essentiellement de vérifier qu'un module répond de façon précise aux spécifications qui lui ont été données. Ce module doit répondre aux spécifications du langage utilisé et le compilateur doit détecter toute erreur ou ambiguité syntaxique rencontrée. Le module doit être compatible avec la machine utilisée qui doit détecter toute impossibilité pendant l'exécution. Mais le plus important est de vérifier que le travail fourni est le travail demandé. Pour cela, il sera nécessaire d'introduire une série de tests par le compilateur pour vérifier la bonne exécution des

instructions Civa (absence de débordement d'indice de file par exemple), par le programmeur pour vérifier l'adéquation du module à ses spécifications ; ceci va alourdir considérablement le texte et la durée du module. Il faudra donc prévoir deux modes d'exécutions : mode mise au point et mode exploitation.

En mode mise au point, toutes les vérifications prévues par le compilateur ou par le programmeur figurent dans le texte généré.

En mode exploitation, l'utilisateur doit définir lui-même les vérifications qui subsisteront. Pour celles introduites par le compilateur, seules resteront celles qui ne "coûtent rien" ; on dira qu'un service ne coûte rien lorsqu'il n'a aucune incidence sur le texte généré par le compilateur pour un programme qui ne l'utilise pas. (La détection des indices hors limite est un service qui coûte, mais la détection de fin de fichiers sous SIRIS 7 ne coûte rien). Une variable booléenne de la classe système, mise au point, indique au compilateur que toutes les aides à la mise au point doivent être insérées. Le programmeur peut lui aussi l'utiliser comme une méta-variable, pour prévoir des instructions qui ne seront compilées que dans ce mode.

Exemple :

¢ *si mise au point* ¢ *alors* action ¢ *fsi* ;

¢ le module action n'est appelé que si le module est compilé en mode mise au point ¢.

Cette phase est en général menée par le programmeur qui a réalisé l'écriture du module. Celui-ci le connaît et pourra facilement analyser un incident et y remédier par une modification du texte original. Mais ceci présente un inconvénient : en effet, il est très difficile d'admettre les erreurs de logique sur des "raisonnements-évidents", et des anomalies risquent ainsi de persister après cette première phase. Une meilleure solution consisterait en deux mises au point : la première, par le programmeur, la seconde, par une autre personne.

6.1.1.2. Regroupement de plusieurs modules

Il s'agit encore de vérifier que la chaîne est viable et répond à ses spécifications. Quelques anomalies (le moins possible) vont apparaître lors de cette deuxième phase ; les problèmes sont analogues à la mise au point d'un module, mais plusieurs phénomènes auront une grande influence.

Le nombre d'informations manipulées est important et l'analyse d'une anomalie est rendue plus difficile.

Toute modification peut créer une nouvelle anomalie : on aura intérêt, si l'on décide de modifier un module, à reprendre la première phase pour celui-ci.

Plusieurs personnes (programmeur, pupitreur, analyste) peuvent être concernées par cette étape, ce qui pose un problème de documentation commune.

L'exécution d'un essai devient coûteuse et deux attitudes sont alors possibles : comme la chaîne de traitement a une longue durée de vie,

on peut donc se permettre de perdre du temps-machine avant la mise en exploitation et tous les contrôles prévus dans la phase "mise au point d'un module" subsisteront ; la chaîne de traitement est éphémère et l'on peut prendre le risque de passer rapidement à la phase exploitation en remédiant aux incidents directement, sans essayer de les prévoir à l'avance ; en fait, une telle chaîne ne passe jamais en exploitation réelle.

6.1.1.3. Exploitation d'une chaîne de traitement

Dans cette phase, l'application est considérée comme parfaitement au point, c'est-à-dire efficace, mais toute anomalie n'est pas forcément exclue ; le traitement des incidents est propre à chaque type d'exploitation. Quatre types d'incidents peuvent se produire.

- Détection d'une erreur de programmation

L'analyse de l'erreur devient plus difficile après la disparition de nombreux contrôles et une anomalie non détectée provoque en général une série d'anomalies invisibles aboutissant à une anomalie visible n'ayant que peu de rapport apparent avec la première. Il est en général hors de question de reprendre tout le traitement en mode mise au point pour analyser l'erreur. On peut appliquer parallèlement deux méthodes : prévoir des points de reprise, repartir en mise au point du dernier point de reprise et prévoir aussi une procédure de récupération qui sauvegarde un grand nombre d'informations de manière à pouvoir, a posteriori, analyser l'erreur.

- Incidents hardware locaux (erreur de lecture sur une bande magnétique, par exemple)

Il faut alors associer à chaque incident une procédure de récupération permettant d'indiquer la conduite à tenir dans chaque cas, ceci doit être prévu dès l'analyse pour être efficace.

- Incident global et imprévisible (erreur système, panne machine)

Aucune séquence de récupération ne peut être prévue en dehors de celles proposées par le système lui-même (sauvegarde des fichiers). Le seul remède contre ce type d'erreur est l'existence de points de reprise prévus par le programmeur à des moments précis figurant dans le texte du programme.

- Incident global prévisible par l'exploitation

L'exploitation se voit dans l'obligation d'arrêter une chaîne de traitement, mais pour pouvoir interrompre la chaîne de manière à la reprendre ensuite, il faut donc qu'elle soit capable de prévoir un point de reprise à la demande de l'opérateur ; autrement dit, il faut qu'il existe une procédure de récupération permettant de créer un point de reprise.

Il faut aussi tenir compte de plusieurs aspects pouvant influer sur l'exploitation d'une chaîne.

- Application évolutive

Nous appellerons application évolutive une application comportant un ensemble important d'informations (fichiers) évoluant au cours du temps. Il faut alors définir des points de reprise au niveau de l'application et non seulement au niveau d'une chaîne de traitement ; le problème n'étant pas

seulement de repartir du milieu d'une chaîne de traitement, mais au début de celle-ci. Les opérations de sauvegarde de fichiers deviennent alors très importantes.

- Communication entre programmeur et utilisateur

Dans de nombreuses installations, le lancement d'une chaîne de traitement amène naturellement la coopération des équipes de programmation et d'exploitation. Toutefois, les packages sont l'exemple type de des applications pour lesquelles les communications sont pratiquement nulles. Leur maintenance est difficile car les utilisateurs de package n'ont que des rapports lointains avec les producteurs. En particulier, si un incident se produit, il faut, a priori, n'attendre aucune aide des utilisateurs : ils peuvent être coopératifs, mais on ne doit pas le supposer lors de l'analyse ! Toute anomalie doit pouvoir automatiquement entraîner le déroulement d'une procédure de récupération permettant de visualiser a posteriori l'incident. Le maximum de renseignements doit être sauvegardé car les personnes qui vont les analyser n'auront pas d'autres contacts avec la source de l'incident.

- Rentabilité d'une chaîne de traitement

Comme les points de reprise ne sont pas gratuits, leur présence et leur fréquence doivent être soigneusement étudiées en fonction de la durée de la chaîne, du temps perdu à chaque sauvegarde et du taux de fiabilité du système utilisé. Par exemple, si t est le temps perdu en point de reprise, T la durée d'une chaîne, t/T ne doit pas être supérieur à la probabilité d'avoir un incident pendant l'intervalle de temps T.

Rappelons que la durée d'un programme dépend en général d'un ou deux modules figurant dans des boucles internes. Ces modules doivent être particulièrement étudiés, mais les autres n'ont pas besoin d'être optimisés du point de vue temps et, en particulier, peuvent très bien rester en mode mise au point.

6.1.2. Problèmes rencontrés pendant la maintenance

Nous pouvons regrouper l'ensemble des problèmes évoqués dans le paragraphe précédent en plusieurs catégories et voir comment Civa peut aider leur résolution. Une remarque préalable s'impose : la maintenance exige la coopération de plusieurs personnes n'ayant pas les mêmes connaissances sur l'application (analyste, programmeur, spécialiste système, pupitreur), et les besoins ne sont pas toujours les mêmes pour chacun. Il importe donc d'adapter les outils proposés à leur utilisation.

6.1.2.1. Modification d'une chaîne ou d'un module d'une chaîne de traitement

C'est le rôle principal de la maintenance : dès que l'on constate qu'un texte ne répond pas aux spécifications ou qu'une spécification change, il faut introduire des modifications au sein d'une application. Pour minimiser l'impact d'une modification, elle se doit d'être locale et ne concerner directement qu'un petit nombre de modules (un seul, si possible !). Si l'on veut faciliter la maintenance, il importe donc de s'appuyer sur le caractère modulaire des descriptions. En particulier, si toute action ou information n'est décrite qu'une seule fois, une modification de celle-ci ne figure

qu'une seule fois : l'interprète du module de commande déduit alors toutes les conséquences sur l'ensemble d'une chaîne grâce à la notion d'état d'une unité.

6.1.2.2. Existence de points de reprise

Ceux-ci peuvent être de deux types.

- Type local

Il est utilisé pour pouvoir analyser une erreur. (Si nous sommes dans la phase regroupement de plusieurs modules, on peut admettre l'exécution de la chaîne en mode exploitation, en reprenant a posteriori, en mode mise au point, un module dans lequel une erreur s'est introduite).

- Type global

En mode exploitation, il permet de reprendre une chaîne accidentellement interrompue.

Ils peuvent être déclarés par le programmeur ou par l'opérateur.
Dans ce cas, l'utilisation d'un système modulaire s'avère particulièrement intéressante car toute entrée dans un module peut être considérée comme une adresse de reprise et l'état de la chaîne est alors connu du programmeur qui peut prévoir les sauvegardes indispensables à la reprise. L'aide supplémentaire que nous pouvons apporter est de faciliter les opérations de sauvegarde et restauration.

6.1.2.3. Accès aux informations nécessaires à l'analyse d'une anomalie

C'est ici que l'aide au programmeur est la plus nécessaire. D'une part, pour que l'information principale, l'erreur proprement dite, soit connue le plus tôt possible (avec le minimum d'erreurs intermédiaires non décelées) et, d'autre part, si l'erreur source ne peut être directement localisée, toute information peut être utile à un utilisateur pour la retrouver. Trois aides sont donc possibles : accès aux informations systèmes, possibilité d'éditer facilement une information sous un format prévu et de créer soi-même des vérifications pour déceler une erreur dès son apparition.

6.1.2.4. Contrôle de l'exécution

Pour indiquer une procédure à effectuer en cas d'incidents hardware, déclencher une sauvegarde sur intervention de l'opérateur, éditer ou sauvegarder les informations adaptées à chaque erreur rencontrée, introduire de nouvelles vérifications prévues ni par la programmation initiale ni par le système, il faut contrôler l'exécution. Pour permettre ce contrôle, nous introduirons la notion d'événement : c'est un changement d'état d'un objet quelconque d'un programme survenant à un instant non prévisible ; il sera caractérisé en général par le changement de valeur d'une expression booléenne. Une instruction permettra d'associer, à un événement, une suite d'instructions à effectuer lors de l'apparition de l'événement.

6.2. ECRITURE DES ENTREES-SORTIES ADAPTEES A LA MISE AU POINT

En mise au point, un problème primordial est la connaissance de nombreuses informations, de manière à obtenir, avec un effort de conception minimum, le maximum de renseignements et à pouvoir faire facilement des sauvegardes et des restaurations. Le programmeur doit faire ces opérations rapidement, sans se préoccuper de définir des fichiers : une action doit, si possible, se traduire par une seule instruction.

Il existe en fait trois opérations d'entrée-sortie de base : éditer, préserver temporairement et récupérer un ensemble d'informations. Il serait possible de définir trois nouvelles instructions correspondant à ces trois opérations mais nous avons préféré les définir comme des méta-modules : $imprimer, $sauvegarder, $restaurer. Ils peuvent être écrits en Civa en utilisant le répertoire d'instructions de base ; compte tenu de la fréquence de leur apparition, on peut envisager d'écrire trois modules en assembleur, les méta-modules se réduisant à un appel de sous-programme.

Remarquons enfin que ces méta-modules ne sont pas à usage exclusif de la maintenance ou de l'aide à la mise au point, mais peuvent être utilisés n'importe où.

6.2.1. Edition simple

6.2.1.1. Définition du méta-module "$imprimer"

Prévoir un format pour éditer une variable est une opération extrêmement fastidieuse, lorsque seule sa valeur nous intéresse et cela simplement lors de la mise au point d'une application. Ce problème n'est pas nouveau et d'ailleurs résolu dans la plupart des langages de programmation (instruction ØUTPUT de Fortran IV étendu, DISPLAY de Cobol...). Nous proposons un méta-module de même rôle dont le seul but est de lister, suivant un format défini par l'implémenteur, une variable quelconque ou même un groupe de variables. Elle s'écrit :

$imprimer (liste de sortie) ;

une liste de sortie est une liste composée de

- constantes de type simple ou chaînes de caractères ;
- identificateur de structure, variable simple, variable indicée ;
- identificateur de file ou de sous file.

Les divers éléments de la liste seront édités sur le fichier de contrôle suivant un format standard défini lors de l'implémentation. Sur IO 070, nous choisirons pour chaque type :

booléen : VRAI ou FAUX

caractère : le caractère ou une "*", si le code du caractère n'appartient pas au code EBCDIC

entier : sa valeur cadrée à droite sur 10 caractères

décimal : sa valeur, le point décimal virtuel étant imprimé

réel ou double précision : sa représentation suivant le format type E ou F de Fortran IV

complexe : il sera représenté par deux réels séparés par ","

file de caractère : la chaîne de caractères (sans séparation entre les caractères)

file d'éléments simple : chaque élément de la file sera édité suivant le format défini ci-dessus et ces éléments seront séparés par des espaces

structure : chaque champ sera édité suivant son type.

6.2.1.2. Exemple

```
module m ;
        i entier ;
        a file (5) car ; b file (5) entier ;
        x, y, z reel ;
        pour chaque x de b, i de 5 a 10 pas 1 faire x=i fp
        a = 'abcde' ; i = 15 ;
        x = 2 ↑ 10 ; y = 1/3 ; z = 10 ↑ i ;
        $imprimer ("a", a, "i", i) ;
        $imprimer (a (1 : 3), b (1 : 3)) ;
        $imprimer (x, y, z) ;
fin module ;
```

provoquera l'impression de :

```
A       ABCDE       I       15
ABC     5           6        7
1024.   0.333333    0.1E+11
```


6.2.2. Edition de zones en hexadécimal

6.2.2.1. Position du problème

Imaginons la déclaration ci-dessous

```
x ( 20) entier  ;
y (100) car ;
z (100) reel ;
```

et supposons qu'une variable entière t ait une valeur aberrante, nous nous orientons vers une erreur de débordement : si nous demandons l'impression de t, le renseignement sera peu intéressant, mais sa valeur éditée en hexadécimal pourra souvent nous donner une "piste". Sur 10 070, la valeur X'E3D6E3D6' fera penser à une chaîne de caractères, indice de débordement du tableau y ; X'000425E7' sera vraisemblablement un entier et X'40D50000' sera probablement un réel... Dans de nombreux cas, ce n'est d'ailleurs pas un mot qui nous intéressera mais une zone.

Pour donner quelques libertés au programmeur, la mémoire est définie dans la classe système par un tableau d'entiers appelé *mémoire*. L'adresse d'un élément peut être connue par les méta-fonctions systèmes *$adrinf* et *$adrsup*. *$adrinf* (x) est l'adresse du mot de x dont l'adresse est la plus basse, *$adrsup* (x) celui dont l'adresse est la plus haute ; x peut être un nom de file, de structure, de variable simple, d'un module ou d'une classe et la zone repérée par *$adrinf* et *$adrsup* est alors la zone sans protection de taille fixe associée à la classe ou au module.

Exemple :

```
a, b, c, d, ... x, y, z entier ;
$imprimer (memoire ($adrinf (a) ; $adrsup (z))) ;
```

provoquera l'impression des 26 variables.

6.2.2.2. Module de conversion de binaire en hexadécimal éditable

Ce module a la description suivante.

séquence d'appel :

```
hexadécimal (i, c) ;
```

i est une expression entière, adresse du mot dont le contenu est à éditer ;

c est une file de caractères dans laquelle on rangera 8 caractères, chacun d'eux correspondant à un chiffre hexadécimal.

Il est écrit, pour des raisons de rapidité, en métasymbol, et pourra être directement utilisé pour des éditions simples.

6.2.2.3. Exemples d'utilisation : module de vidage mémoire

Ce module est écrit en Civa et possède deux paramètres : les adresses hautes et basses de la zone à éditer en hexadécimal sur imprimante

```
module dump (inf, sup), inf, sup entier ;
¢ inf et sup sont des expressions entières, adresses hautes et
basses de la zone à lister ¢.
```



```
¢ : ce module édite une zone mémoire en hexadécimal, à raison
de 8 mots par ligne et, en tête de ligne, figure l'adresse
hexadécimale du premier mot ¢.
        ligne structure (adresse file (5) car,
              zone utile file (8) contenu file (8) car ) ;
        n ligne entier ; n ligne = i/8-1 ;
        k entier ; adr entier ;
    pour chaque k de 1 a n ligne pas 1 , adr de i pas 8
        faire editer ligne fp ;
        adr = adr + 8 ;
    pour chaque contenu de zone utile faire contenu=' ' fp ;
        editer ligne ;
    procedure editer ligne ;
        adresse 1 file (8) car ;
        hexadecimal (adr, adresse 1) ;
        adresse = adresse 1 (4 : 8) ;
        pour chaque mot de memoire (adr : adr + 8) ;
                contenu de zone utile faire
            hexadecimal (mot, contenu) fp
            imprimer ligne ;
    fin procedure
fin module
```

Exemple d'utilisation :

```
module m ; a file (3) entier ;
pour chaque x de a faire x = -1 fp ;
dump ($adrinf (a) , $adrsup (a)) ;
fin module ;
```

provoquera l'impression de :

```
04810    FFFFFFFF    FFFFFFFF    FFFFFFFF
```

Un méta-module permettra d'adapter ce module à n'importe quelle circonstance et rendre ainsi son appel beaucoup plus aisé. Ce méta-module s'appellera vider

```
$vider ;
```

demandera l'impression de toute la mémoire utilisateur.

```
$vider (x) ;
```

demandera l'impression de la zone mémoire contenant x, x peut être une variable simple, une structure, une classe.

```
$vider (x, y) ;
```

demandera l'impression de la mémoire à partir de l'adresse la plus basse de x jusqu'à l'adresse la plus haute de y.

```
$module vider ;
$ si num (AF) = 0  $ alors dump (adrmin, adrmax)
¢ dump mémoire complet adrmin et adrmax étant les adresses extrêmes de la mémoire à l'exécution ¢
$ sinon $ si num (AF) $=1 $ alors dump ($adrinf (AF (1)),
                                        $adrsup (AF (1)) ;
$ sinon dump (adrinf (AF (1)), adrsup (AF (2))
        $ fsi  $fsi
$ fin module
```

Exemple d'utilisation :

```
adr1, adr2, adr3 file (8) car ;
hexadecimal ($adrinf (a), adr1) ;
hexadecimal ($adrinf (b), adr2) ;
hexadecimal ($adrinf (i), adr3) ;
$imprimer "adresse de a:", adr1, "adresse de b:", adr2,
          "adresse de i:", adr3 ;
$vider ;
```


provoquera un dump mémoire complet avec l'édition préalable des adresses de a, b, i, rendant plus facile l'exploitation de celui-ci.

Remarque : bien entendu, l'utilisateur pourra définir de la même manière d'autres procédures de dump (par exemple, en éditant simultanément chaque zone sous la forme de chaîne de caractères).

6.2.3. Sauvegarde et restauration

6.2.3.1. Description des méta-modules $sauvegarder, $restaurer

Dès que l'on veut programmer un point de reprise, on a besoin de sauvegarder un certain nombre d'informations pour les récupérer ultérieurement ; la méthode la plus simple pour repérer un certain nombre d'informations est de leur donner un nom. Il est donc bon, au niveau d'une application, de définir deux méta-modules, spécialisés dans ces opérations. On peut définir par exemple :

```
$sauvegarder (nom, liste de sortie) ;
$restaurer   (nom, liste d'entrée) ;
```

nom est une variable ou une constante de type chaîne de caractères (ces caractères doivent être alphanumériques) ; nom repère l'emplacement où sera sauvegardé et d'où sera restauré l'ensemble d'informations défini par la liste d'entrée ou de sortie.

liste de sortie a la même définition que celle utilisée dans $imprimer ; liste d'entrée est analogue à liste de sortie, mais les constantes y sont exclues.

6.2.3.2. Réalisation de ces méta-modules

Les informations désignées dans la liste constitueront des enregistrements d'un fichier (à clé sous un système tel que BPM, séquentiel indexé sous un système tel que SIRIS 7). Ce fichier doit être transparent pour l'utilisateur, en particulier tous les blocages des enregistrements sont réalisés par des modules de service appelés par les méta-modules ; il est unique pour une application.

Le fichier ainsi défini sera permanent à l'application, et des modules de nettoyage seront prévus, ce qui est simple si l'on prend, dès le lancement de l'application, des règles du genre :

- attribuer à chaque programmeur deux lettres : elles sont utilisées comme premières lettres du nom désignant l'emplacement de sauvegarde ; l'une est associée à des informations de caractère relativement permanent, l'autre à des informations à caractère temporaire
- interdire l'utilisation d'autres noms
- supprimer relativement souvent toutes les informations à caractère temporaire.

6.2.3.3. Exemple d'utilisation de $sauvegarder et $restaurer

- Sauvegarde du contenu d'une classe

```
module m ;
        utilise c ;
        ...
        $sauvegarder ("classe c",
                    memoire ($adrinf (c) : $adrsup (c))) ;
        ...
fin module
```

- Restauration d'une classe préalablement sauvegardée

```
module reprise m ;
        utilise c ;
        $restaurer ("classe c",
                    memoire ($adrinf (c) : $adrsup (c))) ;
fin module
```

Pour faciliter ce genre de traitement, il est facile de définir un méta-module tel que $sauveclasse (nom, liste de nom de classe) ou $restaureclasse (nom, liste de nom de classe).

- Utilisation de ces instructions pour définir des variables remanentes

```
module edition bordereau ;
        utilise classe edition ;
        $restaurer ("numero", numero d'ordre)
        ¢ numéro d'ordre est défini dans classe édition ¢ ;
        pour chaque bordereau faire edition fp
        $sauvegarder ("numero", numero d'ordre) ;
fin module
```

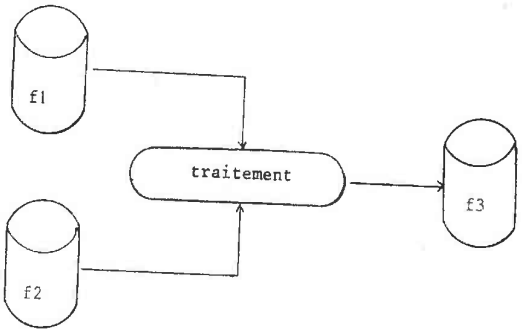
6.2.4. Exemples d'utilisation des entrées-sorties simplifiées

6.2.4.1. Programmation d'un point de reprise

Etant donné un travail décrit par :

```
module traitement ;
        utilise clf1, clf2, clf3 ;
        pour chaque x1 de f1 faire action fpc
fin module
```

action est un module qui crée séquentiellement des enregistrements dans f3 à partir d'exploration séquentielle dans f2 ; les fichiers f1, f2, f3 d'élément courant x1, x2, x3 sont définis dans clf1, clf2, clf3,



Le module de traitement peut être redéfini comme suit :

```
module traitement ; utilise com ;
        utilise clf1, clf2, clf3 ;
        i, j, k entier  i=1 ;
        si reprise alors $restaurer ("traitement", i, j, k) ;
             positionnement (f2, j) ;
             positionnement (f3, k) ;
             fsi
        pour chaque x1 de f1 (i:) faire
             $sauvegarder ("traitement" rang(x1), rang(x2),
                          rang(x3)) ;
             action ; fpc
fin module ;
```

"com" est une classe de commande dans laquelle on trouve la déclaration :

```
reprise booleen ;
```

Trois modules peuvent être définis sur cette application :

a) lancement d'un traitement à partir du début

```
module normal ;
        utilise com ;
        reprise = faux ;
        traitement ;
fin module ;
```

b) lancement avec reprise

```
module reprise ;
        utilise com ;
        reprise = vrai ;
        traitement ;
fin module ;
```

c) analyse en cas d'abandon

```
module analyse ;
        entier rangf1, rangf2, rangf3 ;
        $restaurer ("traitement" rangf1, rangf2, rangf3) ;
        $imprimer (rangf1, rangf2, rangf3) ;
fin module ;
```

Ces trois modules peuvent être regroupés dans une seule chaîne pour former un module de commande, mais le rôle de "analyse" devient plus important car ce module choisit entre une tentative de reprise et un arrêt ; il faut alors sauvegarder plus de renseignements sur la nature de l'erreur, par exemple. Ils peuvent être considérés aussi comme trois modules de commande distincts, chacun étant lancé à l'initiative de l'exploitation. Remarquons, dans ce dernier cas, que la seule manipulation en salle machine est le lancement des divers modules, (aucune saisie d'informations externe n'est nécessaire pour la reprise).

Remarques :

La fiabilité d'un tel dispositif n'est pas absolue car elle suppose qu'en cas d'incident, le fichier de sauvegarde n'a pas été altéré et que l'enregistrement traitement a bien été sauvegardé par le système, ce qui peut être faux dans le cas d'une erreur système. Notons cependant que le module de commande analyse permet de savoir si a priori la reprise est possible ou non.

L'exemple traité ci-dessus est particulièrement simple. Il est facile de l'étendre à des traitements plus complexes en augmentant les sauvegardes

(par exemple, en cas de cumuls, les variables cumulées peuvent être sauvegardées et restaurées très facilement).

Si cet exemple est simple, comment un programmeur débutant ne disposant pas de ce type d'instructions peut-il le traiter ?

- sans reprise, car "cela demande un peu d'imagination" ; cette réaction est malheureusement fréquente. Ce type d'instruction peut donc inciter un programmeur à prévoir des reprises ;
- édition des valeurs de *rang* (x1), *rang* (x2), *rang* (x3) à chaque boucle : ceci a pour effet direct de gaspiller du papier, dans 60 % des cas, et nécessite de prévoir un module de reprise, variable en fonction des valeurs éditées, ce qui est une source supplémentaire d'erreurs ;
- la meilleure solution est de prévoir soi-même un fichier où le contexte est périodiquement rangé.

C'est cette dernière démarche que nous voulons faciliter.

6.2.4.2. Module d'analyse et d'édition associé à une classe

Pour une classe, il peut être intéressant d'écrire un module dont le rôle est de vérifier que les informations contenues dans cette classe sont logiques et répondent à un certain nombre de critères, et d'éditer les renseignements utiles à une analyse plus approfondie. Un tel module pourra être appelé systématiquement en mode mise au point après chaque manipulation sur cette classe. Dans les phases ultérieures, il pourra servir à une vérification a posteriori si un incident est arrivé pendant un traitement : il suffit alors de prévoir des sauvegardes de cette classe, opération bien

moins coûteuse qu'une série de vérifications. Sur une grosse application, de tels modules doivent faire gagner un temps précieux aux programmeurs chargés de la maintenance, mais, pour cela, ces modules doivent être écrits et définis dès la phase d'analyse.

Exemple :

Dans une application de comptabilité, la classe dépense contient quatre files contenant les dépenses prévues et effectives par atelier et par mois. On peut supposer qu'elle est utilisée dans un traitement faisant les cumuls des dépenses en fonction de chaque fournisseur.

```
classe depenses ;
        fgm    prevu file(12) depense globale mensuelle entier ;
        fdapa prevu file(25) depense annuelle par atelier entier;
        ¢ la somme des éléments de fgm doit être égale à celle
        des éléments de fdapa ¢ ;
        fgm    effectif comme fgm    prevu ;
        fdapa effectif comme fdapa prevu ;
        ¢ l'écart entre deux éléments correspondants de
        fgm prévu et fgm effectif doit toujours être inférieur
        à 10 % ;
        test booleen ;
fin classe ;
```


A cette classe, il est facile d'associer un module qui vérifie la cohérence des informations contenues

```
module analyse depenses ;
        utilise depenses ;
s, s1, s2, s3 entier ; test = vrai ; total entier ; total = 0 ;
$somme (fgm prevu, s) ; $somme (fgm effectif, s1) ;
$somme (fdapa prevu, s2) ; $somme (fdapa effectif, s3) ;
$imprimer ("test concordance fgm fdapa") ;
si s=s2 alors $imprimer ("prevu : bon") ;
        sinon $imprimer ("prevu:erreur",s, s2); test=faux fsi ;
si s1=s3 alors $imprimer ("effectif:bon") ;
        sinon $imprimer (effectif:erreur", s1, s3);
                        test = faux fsi ;
si test alors sortir fsi ;
pour chaque x de fgm prevu, y de fgm effectif faire  s=(x-y)/x ;
        si s > 1.1 ou s < 0.9 alors ;
            imprimer ("decalage", rang(x), x, y) ;
                        total = total + 1 fsi ;
si total > 3 alors $vider (depenses) ;
¢ demande dump mémoire hexadécimal ¢
$imprimer (memoire ($adrinf (depenses) ; $adrsup (depenses))) ;
¢ édition de chaque mot suivant le format entier ¢ fsi
fin module ;
```

En phase d'exploitation normale, six sauvegardes du contenu des dépenses ont été faites avec les noms : D1 ... D6 ; le module de commande pourrait être :

```
module calcul depense ;
        utilise depenses ; nom file (2) car ; nom = "d1" ;
        traitement ;
        analyse depense ;
        si test alors arret fsi ;
        pour chaque i de 1 a 6 pas 1 faire ;
            $restaure classe (nom, depenses) ;
            $imprimer ("point de sauvegarde", nom) ;
            analyse depense ; nom (2) = nom (2) + 1 fp ;
fin module ;
```

On a ainsi réalisé un module faisant une trace de la classe dépense. Lorsque tout est normal, la seule perte est de six sauvegardes peu importantes. En cas d'incident, seul ce qui est significatif est imprimé.

6.3. CONTROLE DE L'EXECUTION

Nous entendons par contrôle de l'exécution la possibilité, donnée au programmeur, d'intervenir automatiquement et facilement lorsqu'un événement survient lors de l'exécution d'un module.

6.3.1. Notion d'événement

La réalisation d'un événement est caractérisé par le changement d'état d'un objet survenant à un instant non défini et souvent non prévisible. De façon plus précise, nous pouvons dire que la réalisation d'un événement peut être :

- certaine ou simplement probable : la rencontre d'une fin de fichier est certaine mais une erreur de programmation est probable ;

- unique ou multiple : la réalisation de l'état compteur nul est unique mais l'apparition d'erreurs d'entrée-sortie peut être multiple.

Les événements peuvent être classés en deux catégories.

- Nous appelons événements indépendants de la volonté du programmeur ceux pour lesquels le moment de réalisation de l'événement ne dépend pas a priori de la volonté de celui-ci : entrent dans cette catégorie tous les événements externes, les événements relatifs aux entrées-sorties (erreur d'entrée-sortie, fin de fichier) et toutes les erreurs de programmation détectées par hardware, par le système d'exploitation et par les

contrôles introduits par le système de compilation ; il est logique de faire intervenir les erreurs dans cette catégorie, car, si elles résultent effectivement d'une action du programmeur, elles ne sont pas voulues. Nous pouvons dire aussi que ce sont des événements dont la réalisation n'est pas liée directement à une instruction d'un programme.

- Les autres changements d'états sont le résultat de l'exécution d'instructions. A première vue, parler d'événement lorsque se réalise un de ces changements d'état peut sembler un abus de langage. Pourtant, au moment de l'analyse d'une classe, la variation d'une variable de celle-ci survient à un moment non défini ; de même lorsqu'un module en appelle un autre, pour le module appelant, la modification d'une variable appartenant à une classe commune se produit, dans le module appelé, à un moment non défini, et sa réalisation peut constituer un événement.

6.3.2. Contrôle d'un événement

Trois problèmes se posent : définir un événement, lui associer une action spécifique, détecter sa réalisation et entreprendre l'action correspondante.

- Définir un événement

c'est définir un état, sa réalisation étant l'événement proprement dit. Pour définir un état, la solution la plus simple est d'utiliser une expression booléenne. Dans le cas d'événements dépendant du programmeur, celui-ci emploiera une expression booléenne normale. Dans le cas d'événements externes au programmeur, on leur associera une variable booléenne ; on caractérise alors cet événement en donnant à la variable booléenne la valeur *vrai*.

- Associer une action spécifique à un événement

est une opération simple au niveau de l'analyse ; il suffit de définir un événement en précisant quelle action devra être entreprise lors de sa réalisation. On peut donc, en programmation, définir une instruction permettant de réaliser cette association, (instruction ØN de PL1 par exemple). Elle peut être statique, l'association d'un événement à un procédure étant valable pour tout un module. Ceci présente de nombreux inconvénients : lorsque la réalisation est multiple, l'action à entreprendre peut être totalement différente suivant l'ordre des réalisations ; lorsque la réalisation est unique, l'action à entreprendre peut varier suivant le moment où elle apparaît.
Elle peut être dynamique : l'association est alors une instruction qui déclare un événement et lui associe une action. C'est cette solution, généralement adoptée pour les événements externes, que nous avons choisie en l'étendant à tout événement.

- Détection de la réalisation d'un événement

Dans le cas où le langage de programmation ne permet pas d'associer une action à un événement, la détection est à la charge du programmeur qui se servira des instructions de test classiques. Cette solution présente un inconvénient : elle oblige à répéter les mêmes instructions de test à chaque fois que la réalisation d'un événement est possible. Il en résulte un texte lourd à écrire, à exécuter et difficile à modifier. Le seul problème est alors de permettre au programmeur d'effectuer l'action prévue ; il se résoud en général par un appel de sous programme. Il est cependant

des cas où il est dangereux d'opérer ainsi, en particulier pour des événements survenant à des instants aléatoires car ils supposent alors que tous les sous programmes de librairie sont réentrants.

La détection de la réalisation d'événement définie par le programmeur est relativement simple : il suffit de générer des instructions de test, dans le cas d'une association statique, ou des appels à un sous programme de test, pour des associations dynamiques, à la suite de chaque instruction modifiant une variable d'un programme. Cette solution s'avère catastrophique du point de vue rendement si l'on n'impose pas de limitation. Le programmeur devra donc préciser les variables pour lesquelles la réalisation d'un état peut constituer un événement. Même avec ces restrictions cet outil reste lourd ; c'est pourquoi, il est souhaitable qu'il soit utilisé avec beaucoup de discernement en mode d'exploitation. En revanche, il peut être très intéressant en phase mise au point, ou lors de l'analyse.

6.3.3. Instruction de contrôle en Civa

6.3.3.1. Variable contrôlable

Une variable sera dite contrôlable si un état de cette variable constitue la définition d'un événement. Pour indiquer qu'une variable possède cette propriété, on fera suivre sa déclaration du symbole de base *controlable*. Lorsqu'une variable est déclarée contrôlable, elle est susceptible d'être contrôlée dans tout module qui l'utilise mais ne le sera effectivement que lorsqu'une instruction "contrôle" portant sur cette variable aura été exécutée.

Exemple :

```
i entier controlable ;
```

Dans la version actuelle, seules peuvent être contrôlables les variables simples (hormis les complexes), les éléments simples d'une structure simple, les éléments courants de file déclarés par une déclaration du type x *de* f. De nombreuses variables réservées sont contrôlables : ce sont celles liées aux événements indépendants du programmeur.

6.3.3.2. Définition d'un événement Civa

Nous distinguerons deux types d'événement en Civa :

- l'événement conditionnel : il est associé à un état d'une variable contrôlable

<événement conditionnel> ::= <identificateur de variable contrôlable>
[<comparateur> <expression arithmétique>]

Cet événement est réalisé lorsqu'une affectation ou une modification ayant été faite sur la variable contrôlable, l'expression booléenne globale prend la valeur *vrai* ; un événement conditionnel peut donc être réduit à une variable booléenne. Remarquons que dans le cas d'une expression de comparaison, celle-ci n'est testée que lorsque la variable de droite est seule modifiée. (Les variables contrôlables de la partie gauche ne sont pas prises en compte).

- l'événement systématique : il est associé à tout changement d'état d'une variable contrôlable

<identificateur de variable contrôlable> [*affecte*]

Cet événement est réalisé dès qu'une affectation ou une modification a été faite sur la variable contrôlable, (le symbole *affecte* n'est obligatoire que pour une variable booléenne).

6.3.3.3. Initialisation d'un contrôle

Initialiser un contrôle, c'est associer à un événement une suite d'instructions à effectuer quand l'événement se produit. Cette initialisation qui constitue une déclaration dynamique d'événement s'écrit :

quand < evenement > *faire* < sequence de recuperation > *fqd* ;

A l'exécution, à partir de la rencontre d'une instruction "contrôle", la variable contrôlable associée à l'événement sera dite contrôlée ; à chaque affectation de la variable, on exécutera systématiquement la séquence de récupération dans le cas d'un événement systématique et seulement si la condition booléenne prend la valeur *vrai* dans le cas d'un événement conditionnel.

Remarque :

L'expression booléenne, dans le cas d'un événement conditionnel, est dissymétrique, en particulier la réalisation de i<j et j>i donne deux événements distincts.

6.3.3.4. Simplification d'écriture

Plusieurs événements ayant la même séquence de récupération peuvent figurer

dans une même instruction *quand* ; elle sera néanmoins équivalente à plusieurs contrôles distincts.

Le symbole *affecte* peut être omis dans le cas de variable non booléenne.

La *syntaxe* de l'instruction contrôle devient alors :

<controle> ::= *faire* <liste d'evenement> *alors* <liste d'instruction> *fqd*

<liste d'evenement> ::= <evenement> [, <evenement>]*

<liste d'instructions> ::= <instruction> [; <instruction>]*

<evenement> ::= <identificateur>

| <identificateur> *affecte*

| <identificateur> <comparateur > <expression arithmétique >

<comparateur >::= < | > | = | ≠ | < | > |

Exemple :

```
module m ;
        i, j entier controlable ; i = j = 0 ;
        quand i faire imprimer "i = ", i fin ;
        quand j > 0 faire imprimer "j = ", j fin ;
        j = 1 ;
        j = 0 ;
        i = 1 ;
        i = j ;
        J = i ;
        i = j = 1 ;
        i = j = -1 ;
```

```
J = 1
I = 1
I = 0
J = 1
I = 1
I = -1
```

6.3.3.5. Libération, modification, suspension d'un contrôle

L'instruction permettant de libérer un contrôle s'écrit :

liberer <suite d'identificateur> ;

<suite d'identificateur> ::= <identificateur> [<identificateur>]*

Un identificateur figurant dans une instruction *liberer* repère une variable contrôlée. Cette instruction annule l'effet de tout contrôle antérieur. (Nous verrons qu'il y a toutefois quelques exceptions pour certaines variables réservées).

Pour modifier un contrôle déjà initialisé par une instruction *quand*, il suffit d'en écrire une deuxième qui annule l'effet de la première.

La manipulation des contrôles nécessitant certaines précautions, il nous a paru utile de joindre deux instructions, non fondamentalement indispensables, permettant de suspendre provisoirement un contrôle et de le reprendre. Ces instructions s'écrivent :

suspendre <suite d'identificateur> ;
reprendre <suite d'identificateur> ;

Il est automatiquement inséré une instruction de suspension et de reprise en tête et en queue d'une séquence de récupération pour la variable associée au contrôle uniquement.

quand i, j *faire* action *fqd* ;

est équivalent à :

quand i *faire* *suspendre* i ; action ; *reprendre* i *fqd* ;
quand j *faire* *suspendre* j ; action ; *reprendre* j *fqd* ;

6.3.3.6. Durée de définition d'un contrôle

La durée de vie maximum d'un contrôle est l'unité, module ou classe dans laquelle il se trouve, et non la durée de vie de la variable contrôlable. Ceci est très naturel, un contrôle étant l'association d'une variable à un certain nombre d'instructions, et il faut que les deux existent pour que le contrôle soit défini. Notons cependant qu'un contrôle initialisé dans un module, peut être valable dans tout module appelé : ceci est logique car les variables et constantes d'un module existent depuis l'appel d'un module jusqu'à ce qu'on en sorte effectivement.

```
module m ;                    module m1 ;
        utilise c ;                   utilise c ;
        m1 ;                          quand i alors action fqd;
        ...                           ...
        i = i + 1 ;           fin module
        ...
fin module


classe c
        i entier controlable ;
          ...
fin classe
```

Le contrôle initialisé dans m1 ne peut avoir une durée de vie supérieure à m1, en particulier il ne porte plus sur l'instruction i=i+1 du module m. Si l'on veut que le contrôle lancé dans m1 soit valable ensuite dans m, il faut le définir dans c en réécrivant m1 et c comme suit :

```
module m1 ;                classe c ;
        utilise c ;                procedure controle i ;
        controle i ;               quand i alors action fqd ;
        ...                fin classe
fin module
```

En règle générale, on a intérêt à définir les contrôles dans les unités où sont définies les variables contrôlées.

6.3.3.7. Etat d'une variable contrôlable

a) une variable contrôlable peut avoir trois états :

- libéré : si elle n'a jamais été contrôlée ou si elle a été libérée par une instruction *liberer*
- contrôlé : à la suite d'une instruction *quand*
- suspendue : à la suite d'une instruction *suspendre*

L'introduction de suspension implicite nous amène à préciser les règles suivantes :

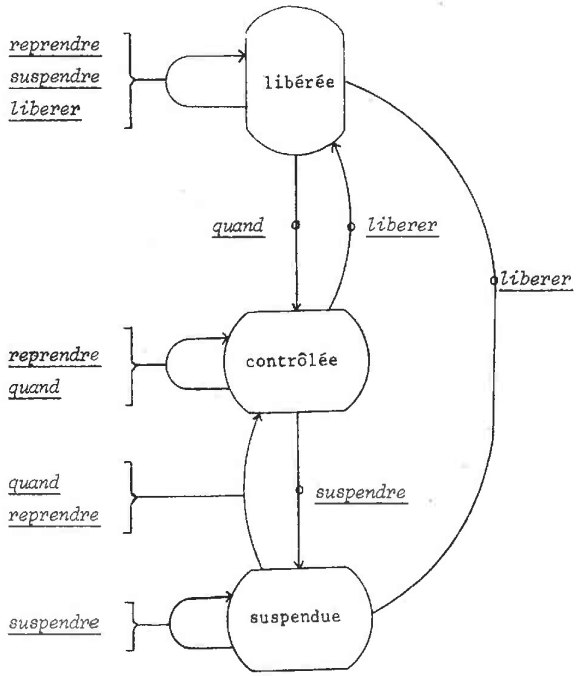
b) une instruction *suspendre* sur une variable libérée ou suspendue n'a aucune action ;

une instruction *liberer* sur une variable suspendue la rend libérée ;

une instruction *quand* sur une variable suspendue la rend à nouveau contrôlée ;

une instruction *reprendre* sur une variable libérée ou contrôlée n'a aucune action.

Ceci peut se traduire par le graphe des états suivant :



GRAPHE DES ETATS D'UNE VARIABLE CONTROLABLE

6.3.4. Exemples d'utilisation des instructions de contrôle

6.3.4.1. Utilisation de *liberer*

```
module n ;
        i entier controlable ;
        quand i = 0 faire j = 1 ; liberer i ; fqd ;
        i = 1 ;
        i = 0 ;
        imprimer j ;
        ¢ provoque l'impression de J = 1 ¢
        j = 0 ;
        i = 0 ;
        imprimer j ;
        ¢ provoque l'impression de J = 0 ; la variable i ayant
        été libérée lors de la récupération précédente ¢
fin module ;
```

6.3.4.2. Suspension implicite

```
i entier controlable ;
quand i > 10 faire i = 15 fqd ;
```

est une expression correcte car, lors de l'exécution de i = 15, la variable i est suspendue et il n'y a aucun risque de "bouclage".

6.3.4.3. Application des suspensions pour l'équivalence de deux variables implantées en des endroits différents par asservissements réciproques

```
i entier controlable ;
j entier controlable ;
quand i faire suspendre j ; j = i ; reprendre j ; fqd ;
quand j faire suspendre i ; i = j ; reprendre i ; fqd ;
```

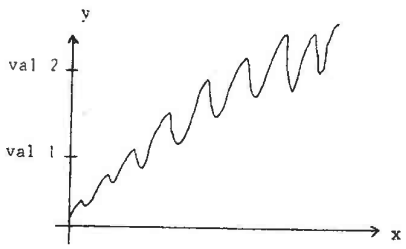
On peut remarquer que l'utilisation de *reprendre* et *suspendre* permet, dans des cas simples, d'optimiser les séquences de récupération. Le compilateur génère les affectations j=i et i=j sans contrôle, c'est-à-dire que la libération effective est inutile ; ceci n'aurait pas été possible si l'utilisateur l'avait écrit sans disposer de ces instructions.

```
procedure controle 1 ;
        quand i faire liberer j ; j = i ; controle 2 ; fqd ;
fin proc ;
procedure controle 2 ;
        quand j faire liberer i ; i = j ; controle 1 ; fqd ;
```

Notons que les appels croisés entre contrôle 1 et contrôle 2 ne sont qu'apparents, car contrôle 1 n'appelle pas contrôle 2 mais précise qu'ultérieurement, lorsqu'on sera sorti de contrôle 1, on devra éventuellement appeler contrôle 1.

6.3.4.4. Modification dynamique d'un contrôle

On veut tabuler une fonction y= f(x) croissante en moyenne avec les contraintes suivantes :



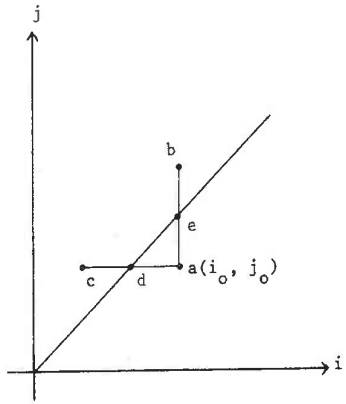
x a un pas de 1 tant que l'on n'a pas rencontré de y > val1

x prend un pas de 10 ensuite, et l'on s'arrête dès que n valeurs de y sont supérieures à val2.

Cette application peut s'écrire :

```
module m ;
        x reel ; y reel controlable ;
        pas, val1, val2 reel ; i, n entier ;
            $ calcul (val1, val2, n) ;
            ¢ ce méta-module donne des valeurs à val1, val2, n ¢
        quand y > val1 faire pas = 10 ; i = 0 ;
            quand y > val2 faire i = i + 1 ;
            si i = n alors arret fsi fqd ;
        fqd ;
        x = 0 ; pas = 1 ; boucle ;
procedure boucle ; x = x + pas ;
        y = f(x) ;
        imprimer y ; recommencer ; finproc
fin module ;
```


6.3.4.5. Réalisation de contraintes



Etant donné 2 variables i et j et une contrainte telle que i > j ; à partir d'un point a(i_o, j_o) la contrainte peut être rompue si j devient trop grand (tentative de rejoindre le point point b) ou i trop petit (tentative d'atteindre c). Très souvent, le traitement d'anomalie est "borner la variable qui tente de déborder" : dans le cas où i devient trop petit, on fera i=j et dans l'autre j=1.

Il est extrêmement simple de programmer un tel problème.

```
i, j entier controlable ;
        quand  i < j faire i = j  fqd ;
        quand  j > i faire j = i  fqd ;
```

Cet exemple illustre le caractère dissymétrique des expressions de relation utilisées dans une instruction *quand*.

6.3.5. Remarque sur l'utilisation de l'instruction *sortir*

Rappelons que l'instruction *sortir* permet de sortir de la procédure ou du module à l'intérieur duquel elle se trouve.

```
module x ;
        ...
        proc ;
        ...
        sortir ;
        ¢ provoquera la sortie du module x ¢
procedure proc ;
        ...
        sortir ;
        ¢ provoquera la sortie de proc et le retour dans x ¢
fin proc ;
fin module ;
```

Il en est de même dans l'instruction *quand* , dans laquelle la séquence de récupération n'est pas considérée comme une procédure.

```
module x ;
        ...
        quand y faire sortir fq ;
        ¢ provoquera une sortie du module x ¢
        quand z faire si t = 1 alors sortir ;
        sinon simprimer (t) fsi fqd ;
        ¢ si t = 1, on sortira du module x, sinon on retournera
        en séquence après avoir imprimé t ¢
        ...
fin module ;
```


6.4. SERVICES D'AIDE A LA MISE AU POINT FOURNIS PAR CIVA

L'aide à la mise au point apportée par Civa inclut bien sûr l'utilisation des contrôles fournis par le 10 070, et les services classiques existant sur un compilateur (contrôle des débordements d'indice, mode trace...). En fait, aucun nouvel outil Civa, aucune nouvelle instruction ne sera introduite, seules seront introduites des variables réservées et contrôlables ; une instruction de contrôle standard sera initialisée en tête du module de commande, mais l'utilisateur pourra modifier ces initialisations, les variables réservées se manipulant comme n'importe quelle variable utilisateur. (Il y aura cependant quelques restrictions pour des raisons de sécurité).

6.4.1. Localisation d'un événement par édition

Dans de nombreux cas, l'action d'associer à un événement est une simple édition de la localisation de l'événement dans le programme. Pour cela, le système Civa met plusieurs variables à la disposition de l'utilisateur mais elles ne sont pas contrôlables.

adrevt entier ; ¢ cette variable contient l'adresse de l'instruction qui a provoqué, ou pendant l'exécution de laquelle s'est produit l'é l'événement ¢

nom module file (max = 12) *car* ; ¢ cette file contient le nom du module en cours ; cette variable n'est positionnée que si le module a été compilé en mode mise au point ¢

nom procedure file (max = 12) *car* ; ¢ cette file contient le nom de la procédure en cours. Sa taille est nulle si l'on n'est pas effectivement dans une procédure ; cette variable n'est positionnée qu'en mode mise au point ¢

instruction struct (emplacement *code*(module, classe),nom *file* (max = 12) *car* ¢ nom de la procédure pendant laquelle s'est produit l'événement ¢, numéro *entier* ; ¢ numéro de ligne de l'événement ¢.

Pour faciliter les éditions dans les séquences de récupération, un module a été écrit une fois pour toutes :

```
module impression standard (message) ;
¢ ce module peut être redéfini par l'utilisateur ¢
        message file car ;
        adresse file (8) car ;
        hexadecimal (adrevt, adresse) ;
        $imprimer ('****', message, 'a l'adresse', adresse) ;
        si mise au point alors ;
        $imprimer ('****', message, emplacement de instruction ,
                           "procedure", nom de instruction,
                           "ligne", numero de instruction ;
        fsi ;
fin module ;
```

Exemple d'utilisation :

```
quand x > 100 faire impression standard ("x trop grand") ;
sortir fqd ;
```

Si l'événement se réalise, il y aura impression de

```
**** X TROP GRAND A L ADRESSE x x x x x   et en mise au point
**** X TROP GRAND MODULE x
       PROCEDURE proc LIGNE x x x
```


6.4.2. *mode trace*

Une trace n'est possible qu'en mode mise au point. Nous envisagerons deux niveaux :

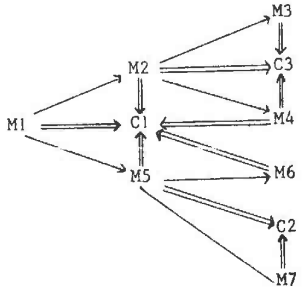
niveau 1

Seront considérés comme points de trace : l'entrée et la sortie d'un module et d'une procédure ; quatre variables contrôlées peuvent représenter ces événements :

entree module
entree procedure
sortie module
sortie procedure

L'utilisateur peut alors facilement doser ses éditions ; considérons un exemple simple d'une utilisation rationnelle.

Soit un ensemble de modules et classes reliés par les relations utilise (⟹) et appelle (⟶) comme l'indique le graphe ci-dessous :



En tête de M1 figure :

```
quand entree module faire $imprimer (nom module);
        $sauveclasse  (c1) ;
        $imprimer (x, y, z) ;
        ¢ où x y z sont des variables de c1 ¢
        fqd ;
quand sortie module faire $imprimer ("sortie module") ;
                          $imprimer x, y, z fqd ;
```

En tête de M2 et M3 figurent deux contrôles semblables sur entrée procédure et sortie procédure (sans sauvegarde).

En cas d'erreur d'analyse difficile sur M2, M3, M4, il est possible de repartir de M2 en introduisant des contrôles supplémentaires, (de même pour M5, M6, M7)

niveau 2

Toute instruction est considérée comme point de trace. Ce mode se révèle extrêmement lourd, c'est la raison pour laquelle, même en mode mise au point, nous ne l'introduirons pas systématiquement. Il faudra que l'utilisateur fasse la déclaration

```
fin instruction controlable
```

pour que cette trace soit possible. Nous l'avons introduite car elle peut se révéler parfois intéressante pour la mise au point du compilateur, en écrivant par exemple

```
quand fin instruction alors dump (0, 16) fqd ;
```

pour éditer les registres à la fin de chaque instruction.

6.4.3. Contrôles des anomalies et erreurs rencontrées pendant l'exécution

Elles sont classées en trois catégories.

6.4.3.1. Anomalie

Si une condition pouvant être anormale a été détectée, il est néanmoins possible de continuer l'exécution. Les contrôles sur anomalie seront initialisés par le système, mais l'utilisateur peut les modifier ou les libérer.

Pour l'ensemble des anomalies, une variable contrôlable permet de repérer la réalisation d'une anomalie. Pour chaque anomalie, il existe, de plus, une variable contrôlable qui porte son nom. En fait, si *anom* est le nom de l'anomalie, le système positionne seulement la variable contrôlable *anom*. La déclaration et l'initialisation des contrôles se présentent comme suit, dans la classe système :

```
anomalie booleen controlable ;
anom 1   booleen controlable ;
...
anom x   booleen controlable ;
quand anom 1 faire ;
        impression standard ("anomalie anom i") ;
        anomalie = vrai fqd ;
...
```

Si l'on a exécuté l'instruction

```
liberer anom 1 ;
```

la variable *anomalie* ne sera pas positionnée s'il y a réalisation de anom i.

Il existe un méta-module $init anomalie dont l'appel est

$init anomalie (nom de la variable caractérisant l'anomalie)

il produit une instruction *quand* associée de façon standard à l'anomalie ou

$init anomalie (toutes)

il produit une instruction *quand* standard sur toutes les anomalies.

En début de module de commande, le système exécute $init anomalie (toutes).

La variable *anomalie* peut être commodément utilisée par le programmeur, pour définir un traitement commun en cas d'anomalie ;

quand anomalie faire arret fqd ; ou :

quand anomalie faire sortir fqd ;

...

6.4.3.2. Erreur simple

Une erreur est une anomalie qui met sérieusement la vie du programme en danger ; en particulier, l'action à entreprendre par le système peut être indéfinie. Ces événements auront un comportement particulier : ils ne peuvent être ni suspendus ni libérés. Si l'on rencontre une instruction *liberer* ou *suspendre*, le système reprend aussitôt le contrôle standard. (Cependant, l'utilisateur récupère normalement une erreur simple car les erreurs simples sont contrôlables). Le traitement est analogue à celui des anomalies (présence de la variable *erreur simple* ; toutefois, *erreur simple* est contrôlable, mais la séquence de récupération associée ne doit pas comporter d'appel de module ou de procédure. Les initialisations standard se font à l'aide du méta-module $init erreur. L'initialisation d'une anomalie se présente ainsi :

```
quand xnom faire ;
        impression standard ("erreur xnom") ;
        erreur simple = vrai ; arret fqd ;
```

(Remarquons que l'action finale est *arret*).

6.4.3.3. Erreur catastrophique

Une erreur est considérée comme catastrophique quand la reprise est impossible. Le programmeur peut toutefois contrôler ces événements à l'aide de la variable *erreur*. Ce contrôle ne peut comporter que des instructions $imprimer ou $sauvegarder, et, dans tous les cas, le système rend le contrôle au module de commande ou au moniteur, si l'erreur se produit au niveau du module de commande.

Sont considérées comme erreurs catastrophiques :

- la rencontre d'une erreur simple dans la séquence de récupération d'une erreur simple ;
- le dépassement du temps imparti à une chaîne de traitement.

6.4.4. Exemples

6.4.4.1. Contrôle de validité des calculs arithmétiques

Il existe trois modes de calculs (entier, réel, décimal) auxquels seront associés trois indicateurs de débordement. Ces indicateurs sont des variables réservées booléennes contrôlables, de nom *dbent* (débordement entier), *dbflt* (débordement flottant) et *erad* (erreur arithmétique décimale). Les événements *dbent*, *dbflt* et *erad* sont considérés comme anomalies.
Sur IO 070, *dbflt* est toujours significatif en mode mise au point, comme en mode exécution. Par contraste, *dbent* et *erad* ne sont significatifs que s'il se produit effectivement un déroutement hardware. Il se produit un déroutement virgule fixe en cas de débordement sur les seules instructions

addition, soustraction et division. Un déroutement erreur décimale ne se produit que sur les conditions décimal incorrect ou débordement de l'accumulateur.

Exemple : tabulation de sin(x)/x

```
module tabulation ;
    f file (200) reel ; y de f controlable ; ...
    quand dbflt faire quand y faire y=1. ; liberer y fqd fqd ;
    pour chaque y de f, x de -1 pas 0.01 faire y=sin(x)/x fqd ;
    $init anomalie (dbflt) ;
    ...
```

6.4.4.2. Contrôle des opérations sur les files

6.4.4.2.1. Contrôle de validité des indices

Ce contrôle n'est possible automatiquement qu'en mode *mise au point*. Dans ce cas, pour chaque variable indicée, le compilateur génère des instructions permettant de vérifier que l'indice reste correct. Deux événements ont été distingués :

ihlecr (indice hors limite en écriture)
ihllect (indice hors limite en lecture).

Le premier est considéré comme une erreur, le second comme une anomalie. Pour ce dernier, le calcul sera faux, mais on peut, en mise au point, laisser se dérouler le programme : ou la conséquence de l'anomalie est locale et l'on a eu raison de continuer, ou elle est importante et l'on ne tardera pas à tomber en erreur. En revanche, dans le cas d'un débordement en écriture, on peut "écraser" n'importe quoi, et il peut être dangereux de continuer.

Exemple : utilisation classique de *ihlecr*, *ihllec*

```
module m1 ;
        utilise c ;
        ¢ c contient la déclaration indic booleen ¢
        pour chaque x1 de f1, x2 de f2 faire ;
        m2 ;
        si non indic alors m3 ;
        fp ;
fin module ;
```

m2 pourrait commencer par :

```
module m2 ;
        utilise c ;
        quand ihlecr, ihllec faire ;
        $imprimer ("erreur dans m2") ;
        $imprimer ("enregistrement :", rang (x1)) ;
        indic = vrai ; sortir ;
        fqd ;
        indic = faux ;
        ...
```

6.4.4.2.2. Autres contrôles des opérations sur les files

On peut utiliser les fonctions *premier*, *dernier*, *sorti* pour définir des événements. A chaque déclaration d'un élément de file X, la réalisation de *sorti* (X) est considérée comme un cas d'erreur simple. *sorti* est une fonction particulièrement intéressante pour traiter les fins de file.

Exemple : interclassement de deux files

```
module interclassement ;
¢ ce module réalise l'interclassement de deux files f1 et f2 dans
une file f ¢
        utilise c ;
        x1 de f1 ; x2 de f2 ; x3 de f ;
        quand sorti(x1) faire f(rang(x3):) = f(rang(x2):) ;
            sortir fqd ;
        quand sorti(x2) faire f(rang(x3):) = f(rang(x1):) ;
            sortir fqd ;
        premier x1 ; premier x2 ; premier x3 ;
        condition ;
```

x1 > x2	V	F
x3 = x1	1	
x3 = x2		1
nouveau x1	2	
nouveau x2		2
nouveau x3	3	3
retable	4	4

```
fin module ;
```

6.4.4.5. Evénement particulier, intervention opérateur, interruption compteur temps

Ces événements peuvent arriver à n'importe quel moment, et Civa n'autorisant pas les procédures récursives et les réentrantes, ils sont à manipuler avec précaution.

- Intervention opérateur

Il existe une variable booléenne *itoperateur* qui est associée à l'interrupteur pupitre. Elle n'est pas implicitement contrôlable et doit être déclarée contrôlable de la même manière que la variable *fin instruction* ; en effet, dans ce cas, pour éviter des incidents, le compilateur génère des instructions qui vérifient que la réalisation de l'événement s'est produite. Ceci entraîne un léger ralentissement de l'exécution.

- Interruption compteur temps

Quatre éléments permettent d'avoir accès à la variable temps.

- la variable réelle non contrôlable *temps restant* : elle contient le nombre de secondes restantes allouées au Job.

- une procédure *init compteur* (t) où t est une expression entière exprimée en seconde ; elle initialise un compteur temps à la valeur t.

- une variable réelle non contrôlable *temps compteur* : elle contient le nombre de secondes restantes avant le positionnement à zéro du compteur temps.

- une variable booléenne pouvant être déclarée contrôlable *itcompteur*. (Elle se comporte comme *itoperateur*). Cette variable prend la valeur *vrai* lorsque le compteur temps, initialisé par une instruction *init compteur*, est à l'état zéro.

Exemple d'emploi : contrôle de la convergence rapide d'une variable "c"

```
module resolution ;
        reel c, c1 ; itcompteur controlable ;
        init compteur (1) ;
        ...
        c1 = c ;
        quand itcompteur faire si c1-c<0.01 alors sortir fsi ;
        c1 = c ; init compteur (1.) fsi fqd ;
        ...
fin module ;
```

conclusion

Ce que nous venons de présenter n'est, en fait, qu'une étape dans la réalisation d'un vaste projet. Un travail important reste à faire pour que le compilateur accepte tous les éléments du langage Civa. Deux points nous paraissent particulièrement intéressants à développer :

Le module de commande, avec la structure que nous lui avons donnée, s'intègre très bien dans le cadre d'une utilisation conversationnelle ; en effet, celui-ci est interprété et le même interprète peut s'adapter, moyennant peu de frais, à un système de time sharing par exemple, les instructions du module de commande étant entrées directement sur console. Il a alors l'avantage d'utiliser les instructions du langage et ne demande aucun effort d'adaptation de la part de l'utilisateur.

En allant plus loin dans cette voie, on peut obtenir un outil de mise au point très efficace en introduisant une exécution conversationnelle des modules ; nous entendons par ceci la possibilité d'interrompre le déroulement d'un module, l'interroger, modifier l'état de ses variables, le reprendre. Ceci ne demande aucune modification de la définition du langage, les instructions *quand* étant bien adaptées à ce genre de traitement. Il faut, cependant, reconnaître, contrairement au premier point, que ce dispositif est d'un prix relativement élevé tant au point de vue conception qu'exécution ; il ne serait donc utilisable que pour des mises au point très localisées (d'une procédure ou d'un module, par exemple).

bibliographie

Articles concernant directement le Projet Civa :

[1] DERNIAME J.C.
Le Projet Civa
Thèse d'état, Nancy, 1973 (à paraître)

[2] DENDIEN J.
Gestion statique de mémoire dans un système de programmation modulaire
Thèse docteur ingénieur, Nancy, 1973

[3] AUBRY B.
Traduction des tables de décision
Thèse 3ème cycle, Nancy, 1973

[4] PERROT D.
Contribution à la compilation dans le Projet Civa
Thèse 3ème cycle, Nancy, 1973 (à paraître)

[5] CHABRIER J.J.
Acquisition, édition de fichiers
Thèse 3ème cycle, Nancy, 1973 (à paraître)

[6] BENAMGHAR L.
Mouvements d'objets dans le Projet Civa
Thèse docteur ingénieur, Nancy, 1973 (à paraître)

[7] CRIDLIG A.
Conséquences d'une modification d'un texte source dans une application Civa ou Métasymbol
D.E.A., Nancy, 1972

[8] HUMBERT
Traduction des expressions arithmétiques dans le Projet Civa
D.E.A., Nancy, 1972

Principales références techniques

[9] C.I.I.

Ordinateur CII 10 070
Manuel de présentation (C 900 950 H/FR)

[10] C.I.I.

Métasymbol sous SIRIS 7/SIRIS 8
(CII, réf. 3753 E/FR)

[11] C.I.I.

Normes de programmation SIRIS 7
(CII, réf. 3851 E/FR)

Pour la conception du compilateur, nous nous sommes inspirés de :

[12] PAIR C.

Cours de compilation
Ecole d'Eté d'Informatique de Neuchâtel (septembre 1972)

[13] LOUIT G.

Optimisation des expressions arithmétiques
Extensions des algorithmes de R. SETHI et J.D. ULLMAN
Congrès de l'A.F.C.E.T., Grenoble, 1972)

En ce qui concerne la manière d'écrire un compilateur, on trouvera les références à LP70 et LMU dans

[14] C.I.I.

Langage de programmation LP70
(C.I.I., réf. 4069 E/FR), février 1972

[15] NOLIN L.

Formalisation des notions de machines et de programmes
Gauthiers-Villars, Paris, 1969

et des articles proposant des méthodes analogues à la nôtre, dans

[16] PELTIER M.

Macro-instruction OS 360 pour l'écriture de compilateur
Séminaire I.M.A.G., 21 juin 1968

[17] ASSABGUI M.

Application du langage Métasymbol à la construction d'un compilateur Algol 60 pour CAE 10070
Séminaire I.M.A.G., octobre 1967

Deux langages proposent une instruction ØN dont la structure ressemble à notre instruction *quand*

[18] VEILLON et CAGNAT

Cours de programmation en langage PL1

[19] RADING et ROGOWAY P.

Highlights of a new programming language
Programming systems and languages, Saul Rosen
Mc Graw-Hill Book Company, 1967

NOM DE L'ETUDIANT : DUCLOY Jacques

Nature de la thèse : Docteur Ingénieur

Vu, Approuvé

et permis d'imprimer

NANCY, le 27 Février 1973

Le Président du Conseil de l'Université de NANCY I

J.R. HELLUY