ordre :

THESE

présentée

A L'INSTITUT NATIONAL POLYTECHNIQUE DE LORRAINE

pour obtenir

LE DIPLOME DE DOCTEUR DE 3ème CYCLE
EN SPÉCIALITÉ INFORMATIQUE

par

Monique MAZAUD

SYSTÈME D'AIDE A LA PRODUCTION DE TRADUCTEURS

Soutenue publiquement le 27 septembre 1976 devant la Commission composée de :

MM.	C. PAIR	Président
	D. COULON M. GRIFFITHS B. LORHO	Examinateurs

THESE

présentée

A L'INSTITUT NATIONAL POLYTECHNIQUE DE LORRAINE

pour obtenir

LE DIPLOME DE DOCTEUR DE 3ème CYCLE
EN SPÉCIALITÉ INFORMATIQUE

par

Monique MAZAUD

SYSTÈME D'AIDE A LA PRODUCTION DE TRADUCTEURS

Soutenue publiquement le 23 septembre 1976 devant la Commission composée de :

MM.	C. PAIR	Président
	D. COULON M. GRIFFITHS B. LORHO	Examinateurs

Monsieur le Professeur PAIR a bien voulu s'intéresser à mon travail en tant que Conseiller scientifique à l'I.R.I.A. Je le prie d'accepter mes remercìements très sincères pour ses encouragements constants, et l'honneur qu'il me fait de présider ce jury.

Monsieur LORHO Ingénieur de Recherche à l'I.R.I.A. a été un guide très critique pendant la réalisation de ce travail, qu'il trouve ici l'expression de ma gratitude.

Je remercie Monsieur le Professeur GRIFFITHS et Monsieur COULON pour avoir bien voulu faire partie du jury de cette thèse.

Je suis reconnaissante à M.C. DENDIEN-GAUDEL et P. KING de la collaboration qu'ils ont apportée à ce travail par de fructueuses discussions.

Mes remerciements vont enfin à Mesdames PEREZ et SUARD pour la réalisation matérielle de ce mémoire.

TABLE DES MATIERES

INTRODUCTION

L'écriture traditionnelle des traducteurs est à renouveler pour chaque langage source et chaque langage objet.

Afin de réduire le coût considérable de cette opération si on change seulement de langage objet, Orgass et Waite (1) (2) ont proposé un traitement par un macroprocesseur. Comme cette méthode est orientée vers un langage source, l'idée d'un langage de macros universel a été développée par Steel dans UNCOL (3) (4). Cette tentative a été un échec essentiellement pour deux raisons : d'une part, il est pratiquement impossible de décrire tous les langages source, d'autre part, la description de la structure des machines est extrêmement masquée dans le codage des macros.

Afin de réutiliser certaines parties des traducteurs déjà écrits lorsqu'on change le langage source ou le langage objet, l'idée d'une traduction vers un langage intermédiaire a été introduite.

C'est l'approche de Coleman dans le système JANUS, le langage intermédiaire introduit dans ce système opère sur une famille de machines abstraites. Les instructions de ce langage intermédiaire permettent de manipuler soit l'opérande placé dans l'accumulateur, soit des opérandes explicitement nommés par un symbole, soit les opérandes anonymes au sommet de la pile. Malgrè tout cette famille de machines abstraites est plus spécialement orientée vers les machines à registres adressables possédant des registres d'index et de base, et il n'est pas facile de passer du code intermédiaire de JANUS au code d'une machine à pile.

Un tel langage qui se veut universel, ne peut l'être en fait car il devrait recouvrir les caractéristiques combinées de tous les langages source et de tous les langages objet. L'adaptation à un nouveau langage source ou à un nouveau langage objet est assez difficile.

Notre démarche propose un découpage accru du processus de traduction, avec l'introduction de deux étapes intermédiaires. Notre objectif est de rendre compatible la première étape de traduction avec toutes les machines cibles : il faut donc que cette étape ne dépende que des langages sources.

Dans toute la suite de ce texte, nous appellerons ℓi_1 le langage intermédiaire dans lequel est réalisé cette première traduction. Nous montrerons plus loin qu'on peut définir un langage ℓi_1 suffisamment général pour qu'il convienne à la traduction de la classe L_B des langages algorithmiques suivants : FORTRAN, ALGOL 60, ALGOL W, PL 1, PASCAL, ALGOL 68. Nous désirons résoudre durant la première étape des traductions tous les problèmes liés à la compilation, pour effectuer

ensuite un traitement séquentiel : le langage ℓi_1 doit donc être un langage pratiquement "hors contexte". La traduction du langage source en langage ℓi_1 consiste à construire l'arbre syntaxique d'un programme du langage source, décoré par l'ensemble des informations sémantiques associées aux noeuds de l'arbre. La syntaxe de ℓi_1 reflète exactement les propriétés de cet arbre décoré.

Supposons que l'on réalise directement la traduction de ℓ . en assembleur d'une IRIS 80, puis d'une IBM 370. Les mécanismes d'adressage étant très voisins dans ces deux machines, on conçoit que la traduction sera très voisine. Pour regrouper les modules de traduction communs à ces deux machines, on va donc définir une deuxième étape de traductions commune à toutes les machines à registres adressables.

Cette étape fait passer du langage ℓi_1 à un langage ℓi_2 comportant des instructions de manipulation de pseudo-registres en nombre indéfini.

Mais cette étape intermédiaire ne peut être utilisée pour les machines à pile (PDP 11, Burroughs 6700, 7700...) pour lesquelles l'adressage est fondamentalement différent de celui des machines à registres adressables. Il faudrait donc définir deux langages ℓi_2 :

- ℓi_2R langage intermédiaire destiné à l'écriture des traducteurs pour les machines à registres et à mémoire adressables.
- ℓi_2P langage intermédiaire destiné à l'écriture des traducteurs pour les machines à pile.

Dans notre réalisation, nous nous sommes limités à la définition du langage ℓi_2R. La définition du langage ℓi_2P destiné à l'écriture des traducteurs pour les machine à pile, est une extension prévue de ce travail.

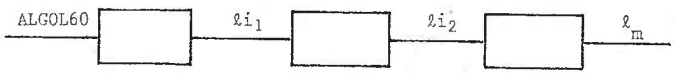
Montrons à l'aide de quelques exemples comment l'introduction des deux langages intermédiaires facilite l'écriture des compilateurs.

Nous prendrons des exemples dans les deux classes de machines :

- la classe des machines à registres adressables notée $\mathcal{C}$.
- la classe des machines à pile notée $\mathcal{C}'$.

a/ Considérons une machine M.

Il existe un compilateur ALGOL 60 et on veut écrire un compilateur PL1 pour cette machine



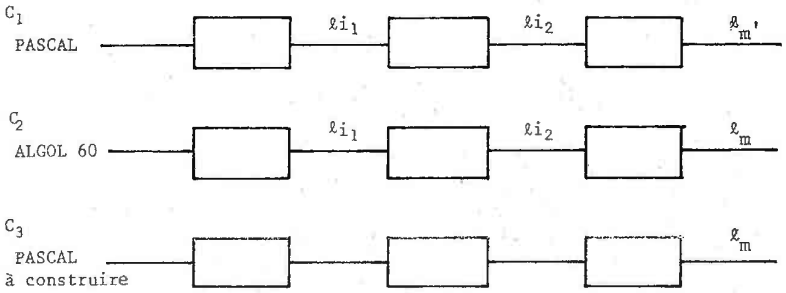
ALGOL 60 et PL1 appartiennent à la classe des langages L_B définie précédemment, par conséquent le premier langage intermédiaire à générer est bien ℓi_1.

Si la deuxième étape de traduction est très modulaire, on peut reprendre dans les compilateurs ALGOL 60, les modules de traduction correspondant aux structures de données communes à ALGOL 60 et PL/1. Pour les structures de

données inhérentes au langage PL1, il faut écrire les modules de traduction. Quant à la troisième étape des traductions, elle est identique puisque la machine cible est la même.

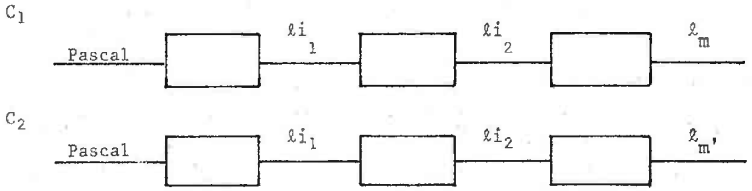
b/ Soit toujours la machine M.

On veut écrire un compilateur PASCAL pour la machine M, on possède un compilateur PASCAL, pour une autre machine M' (de la même classe de machine ou non) et on possède un compilateur ALGOL 60 pour la machine M.



La première partie de la traduction de PASCAL en ℓi_1 existe déjà et peut être reprise telle quelle dans le compilateur C_1. La troisième partie de la traduction existe déjà et peut être reprise telle quelle dans le compilateur C_2 puisque la machine cible est identique. Quant à la deuxième étape de traduction, elle peut être reprise telle quelle dans le compilateur C_1 si la machine M' appartient à la même classe de machine que la machine M ; sinon on reprend dans le compilateur C_2 les modules correspondant aux structures de données communes à PASCAL et à ALGOL 60 (si les choix d'implantation sont identiques).

c/ On veut écrire un compilateur Pascal pour une machine M de la classe $\mathcal{C}$. on possède un compilateur Pascal pour une machine M' de la classe $\mathcal{C}'$.



La première étape de traduction peut être reprise telle quelle dans le compilateur C, il faut choisir dans les modules de traduction de LI_1 et LI_2 que nous proposons ceux qui correspondent aux structures des données de Pascal et écrire la traduction de ℓi_2 en langage machine de la machine cible M.

En résumé, l'écriture d'un compilateur se ramène à trois étapes des traductions :

- une étape de traduction du langage LS en langage ℓi_1, celui-ci dépendant uniquement d'une classe de langage source.
- une étape de traduction de ℓi_1 en ℓi_2 , où ℓi_2 dépend de la structure de la machine (essentiellement de l'organisation de l'unité de traitement).
- une étape de traduction de ℓi_2 en langage machine où on introduit alors les caractéristiques fines de la machine (taille des mots, adressage, nombre des registres disponibles).

Nous allons maintenant établir la comparaison entre l'écriture classique d'un compilateur et l'écriture que nous proposons. Plus précisément nous allons récapituler les fonctions usuelles d'un compilateur et montrer comment elles sont réparties dans notre système.

Nous supposerons que la traduction du langage source en langage ℓi_1 est effectuée à l'aide du système DELTA [6] implanté à l'I.R.I.A., mais ceci n'est pas impératif, l'auteur de compilateur pouvant utiliser les outils dont il dispose.

On peut considérer que le travail d'un compilateur comporte quatre fonctions principales.

a/ La vérification syntaxique du texte source.

La grammaire du langage source à compiler est décrite sous forme de règles BNF et le constructeur syntaxique de DELTA génère l'analyseur syntaxique correspondant à cette grammaire.

Les vérifications syntaxiques correspondant au langage "context free" sont donc réalisées par cet analyseur.

La réalisation des tests contextuels (concernant la validité de l'utilisation des données, des paramètres...) est effectuée à l'aide des attributs sémantiques de KNUTH [7] au cours de la première étape de traduction du langage en langage ℓi_1 .

b/ La création d'une table de symboles où sont rangées les informations concernant les types, adresses des données, toutes informations contextuelles qui servent à effectuer la génération de code machine.

Dans notre système, la fonction de la table des symboles est réalisée par un attribut sémantique. Cette table est remplie lorsqu'on reconnaît dans la syntaxe les instructions des déclarations des données, des étiquettes, des procédures. Les informations concernant les déclarations de ces objets sont transmises (grâce à l'attribut table des symboles) aux instructions y faisant référence et intégrées à la syntaxe du texte ℓi_1 généré.

Dans notre système, une fois la génération du code ℓi_1 effectuée, la table des symboles disparaît car les informations nécessaires à la suite de la traduction ont été intégrées au code ℓi_1 généré.

c/ Les optimisations.

On distingue ordinairement les optimisations indépendantes de la machine et celles qui en dépendent.

Les premières concernent la propagation des constantes, l'élimination des sous-expressions communes redondantes, le déplacement des invariants des boucles. Comme elles mettent en jeu des informations contextuelles, il est naturel de les effectuer par attributs sémantiques, lors de la traduction du langage source en ℓi_1 .

Les secondes dépendent de la machine, elles consistent essentiellement à minimiser les transferts entre l'unité de traitement et la mémoire centrale, donc à adopter une stratégie optimale d'assignation des registres.

Ces optimisations dépendent de la structure fine de la machine, elles doivent donc être traitées lors de la traduction du langage ℓi_2 en langage machine.

Cependant cette dernière phase d'optimisation peut être préparée lors de la traduction de ℓi_1 en ℓi_2 . On peut en particulier rechercher la durée de vie de chaque variable et réaliser, pour les machines à registres adressables, la phase d'allocation des registres (conformément à la terminologie de BEATTY [8]).

d/ La génération de code machine.

Nous avons vu au début de cette introduction que nous ne voulions effectuer la génération de code machine qu'à la fin de la troisième étape de traduction.

Cette génération dépend naturellement des choix d'implantation des données, ces choix ne sont pas inhérents au langage source à compiler (sauf pour ce qui concerne le traitement des précisions des données arithmétiques du PL1) : ils dépendent des objectifs de l'auteur du compilateur, c'est pourquoi nous les introduisons seulement lors de la deuxième étape de traduction de ℓi_1 en ℓi_2 .

Première Partie

L E L A N G A G E L I I

La méthode de définition du langage £i1 .

Notre système doit pouvoir aider à la traduction des langages algorithmiques usuels et de ceux qui en sont dérivés et sont utilisés pour traiter des applications plus spéciales. Nous définirons les caractéristiques du langage £i1 à partir de la comparaison des langages algorithmiques suivants ; FORTRAN, ALGOL 60, ALGOL W, PL1, PASCAL, ALGOL 68. La comparaison portera sur les structures de données, la structure des programmes, les opérations, les structures de contrôle valides dans ces langages.

La suite de ce rapport se compose de l'examen de ces différents points sachant que l'objectif à atteindre après la première étape de traduction est la résolution de tous les problèmes liés à la compilation.

La traduction du langage source en langage £i1 consiste donc à construire l'arbre syntaxique d'un programme du langage source, décoré par l'ensemble des informations sémantiques associées aux noeuds de l'arbre. La syntaxe de £i1 reflétant les propriétés de cet arbre décoré est donc pratiquement "hors contexte". Le langage £i1 est un langage de triplets généralisé, car une adresse dans le code £i1 généré est un numéro de triplet (nous l'écrirons : entier T).

1. LES STRUCTURES DE DONNEES DE LI1

1.1. La méthode de construction.

Pour construire les structures des données de £i1 , deux méthodes viennent à l'esprit. La première consiste à choisir les structures de données du langage le plus riche. Or la comparaison des structures de données des langages source n'est pas aisée ; elle l'est d'autant moins que certains langages autorisent la définition des types.

La seconde méthode consiste à construire une structure de données pour £i1 qui serait l'union des structures de données valides dans les langages usuels. Cette méthode permet évidemment de décrire l'ensemble de ces derniers langages, mais elle manque de souplesse et ne permettrait pas une adaptation à la traduction d'autres langages algorithmiques (A titre d'exemple, la définition de types arithmétiques variant dans une gamme de précisions particulières, ne peut être traitée si on n'a prévu en £i1 qu'un seul type arithmétique réel). Aucune de ces approches intuitives ne permet de résoudre notre problème, seule une solution de paramétrisation des types est acceptable. Pour définir ces paramètres nous allons rappeler comment sont construits les objets des langages algorithmiques. Nous ferons ce rappel en divisant les langages

algorithmiques en trois classes : les langages qui n'autorisent que les types prédéfinis (FORTRAN, ALGOL 60, ALGOL W, PL1), PASCAL qui autorise la définition de types, ALGOL 68 qui introduit l'orthogonalisation.

1.2. Les objets des langages algorithmiques.

Nous distinguerons ici l'ensemble E des langages qui autorisent exclusivement les types prédéfinis (FORTRAN, ALGOL 60, PL1, ALGOL W) ; et les langages PASCAL et ALGOL 68 qui autorisent les définitions de type.

1.2.1. *Les types prédéfinis.*

Les langages de l'ensemble E permettent de manipuler des objets simples possédant un type simple prédéfini : entier, réel, logique en FORTRAN et en ALGOL 60, types artithmétiques, logique, et type chaîne en PL1 et en ALGOL.

Des objets composés peuvent être construits à partir de certains groupements d'objets élémentaires de même type ou de types différents. Dans la terminologie de ces langages, on dit que l'on traite des types complexes : ainsi un tableau d'entiers est un groupement de valeurs entières de type simple entier, le type de ce groupement est le type tableau.

Chaque langage source possède ses règles de construction ou types complexes : le type tableau est valide dans l'ensemble des langages. Le langage PL1 possède en outre le type structure comme type complexe, le langage Algol W le type enregistrement comme type complexe.

Certains langages autorisent l'application de manière répétée de ces constructions par des règles de composition. PL1 permet ainsi de traiter des objets de type tableau de structure ou structure de tableau. En ALGOL 60 ou FORTRAN, il est impossible de répéter la construction tableau ; il est impossible de définir des objets du type tableau de tableau.

1.2.2. *Les définitions de type*

Le langage PASCAL permet de décrire les objets qui gardent une valeur invariable à l'exécution par une déclaration de constante. Le traitement à la compilation peut toujours s'effectuer par la méthode de propagation des constantes. Tous les autres objets sont appelés dans le langage des variables. Ils possèdent un type prédéfini, ou un type défini dans le programme. La définition du type permet ici aussi de créer une hiérarchie conformément à certaines règles : ainsi le type tableau d'ensemble est valide, le type ensemble de tableau ne l'est pas ...

1.2.3. *Les objets d'ALGOL 68.*

La terminologie d'ALGOL 68 diffère de la terminologie usuelle des langages algorithmiques : on dit que chaque objet possède un mode : le mode est une généralisation du type des autres langages algorithmiques. Il caractérise une classe de valeurs sur lesquelles les mêmes opérations et relations sont définies.

Les types simples des autres langages algorithmiques sont appelés modes primitifs en ALGOL 68. Les modes en sont élaborés essentiellement à partir de quatre règles de construction qui sont :

- la construction tableau,
- la construction structure,
- la construction procédure,
- la construction repère.

L'application répétée de ces constructions dans un ordre et un nombre quelconque est autorisée : c'est ce qu'on appelle l'orthogonalisation des modes.

Comme dans tous les langages algorithmiques, un objet est désigné par un identificateur. La déclaration d'un objet peut se faire par deux méthodes : soit la déclaration d'un identificateur suivie d'une description de mode , soit la déclaration d'un identificateur suivie d'un indicateur de mode (nom servant à désigner un mode qui a été déclaré précédemment).

Revenons aux quatre règles des constructions valides en ALGOL 68 pour préciser les constructions procédure et repère.

Les valeurs des objets de mode procédure, peuvent être manipulées comme n'importe quelles autres valeurs, être affectées, incluses dans des structures etc...

Une procédure est un objet quelconque en ALGOL 68, c'est-à-dire qu'il est possible d'effectuer entre autres des affectations entre objets procédure.

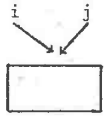
La construction repère permet la construction de "noms". En pratique à l'implantation d'ALGOL 68 un nom est représenté par l'adresse d'un emplacement destiné à contenir la valeur qu'il repère (cette valeur pouvant être un autre nom).

Un nom peut aussi être manipulé comme n'importe quel autre objet. En particulier, il est possible de déclarer que deux identificateurs sont synonymes, car ils désignent les mêmes noms, donc les mêmes adresses.

Exemple : ent j ; .... rep ent i = j

Cette déclaration d'identité comporte deux informations ; d'une part la déclaration d'un objet de mode repère d'entier désigné par l'identificateur i, d'autre part la synonymie des identificateurs i et j.

A l'implantation, on peut représenter cette identité par le schéma suivant



Le langage ALGOL 68 permet de définir les constantes en faisant désigner des valeurs par des identificateurs. Ces valeurs qui sont des notations de constantes ou des valeurs d'expressions sont indépendantes de toute exécution. Des déclarations d'identité permettent de réaliser ces déclarations de constantes.

```
ent c = 3
réel r2 = rac2(i)
```

L'identificateur déclaré désigne la même valeur calculée (cas de r_2), ou écrite (cas de c) aussi longtemps qu'on n'a pas quitté la région où on peut utiliser cet identificateur. Pour les valeurs calculées, il faut réserver à l'implantation une zone de mémoire effective ; les références aux objets constants seront donc traitées exactement comme les références aux objets variables. C'est le travail du compilateur de vérifier que dans le programme les constantes sont utilisées comme telles et non comme des variables.

Pour les valeurs des constantes écrites, on peut effectuer la propagation des constantes. En PASCAL les identificateurs constants désignent uniquement des valeurs écrites et il est toujours possible d'effectuer la propagation des constantes.

En conséquence en ℓi_1 on ne traitera que des variables, puisqu'à la compilation on a pu résoudre le problème des constantes en les traitant soit comme des variables, soit en effectuant leur propagation.

1.3. Les objets de ℓi_1

1.3.1. *Leur méthode de définition.*

Nous venons de voir que tous les langages algorithmiques permettent de manipuler des objets simples ou des objets composés. Le langage ℓi_1 doit donc permettre de manipuler des objets simples ou des objets composés. Pour décrire les structures de données de ℓi_1 nous adopterons la terminologie d'ALGOL 68 : les modes primitifs permettent de décrire les objets simples. L'application des règles de construction sur un ou plusieurs modes primitifs constitue un moyen de décrire des objets plus complexes. L'application répétée de ces constructions constitue une composition.

Les modes primitifs et constructions valides dans le langage ℓi_1 décrit ici sont ceux qui permettent de décrire l'ensemble des langages algorithmiques usuels définis précédemment : FORTRAN, ALGOL 60, ALGOL W, PL1, PASCAL, ALGOL 68. Nous verrons que la construction repère d'ALGOL 68 peut être traitée comme un mode primitif pointeur. Nous décrirons donc les constructions, tableau,

structure, ensemble, procédure. Les compositions de ℓi_1 sont toutes les combinaisons possibles de ces constructions. Cependant il est toujours possible d'introduire d'autres modes primitifs et d'autres constructions pour décrire d'autres langages (par exemple la construction liste pourra être introduite à moins de la ramener à la construction structure comme en ALGOL 68).

Les modes primitifs et les constructions sont donc considérées ici comme des paramètres du langage ℓi_1 .

Revenons à l'ensemble des langages algorithmiques usuels que nous nous proposons de décrire. Etant donné que les structures de données de ℓi_1 se composent de l'union des modes primitifs et des constructions valides dans ces langages, comment peuvent-elles refléter les structures de données d'un langage particulier. Pour répondre à cette question, il faut revenir au schéma fonctionnel de notre système de traduction.

Le système d'aide à l'écriture des traducteurs que nous fournissons se compose de la syntaxe des langages ℓi_1 et ℓi_2 , et des modules de traduction entre ces deux langages. A la charge de l'auteur de compilateur reste l'écriture des actions sémantiques à effectuer pour réaliser la traduction d'un langage source en ℓi_1 .

A titre d'exemple, décrivons le mécanisme d'écriture d'un compilateur ALGOL 60. Seule la construction "tableau" existe dans la syntaxe d'ALGOL 60, dans la règle syntaxique de description d'un tableau on décrira la génération d'une déclaration de tableau en ℓi_1 .

Les autres constructions "structure", "ensemble" etc... n'existent pas en ALGOL 60, on ne pourra donc jamais atteindre leur représentation en ℓi_1 lorsqu'on traduit ALGOL 60. De façon plus générale, seul le sous-ensemble de structures de données de ℓi_1 , correspondant aux structures de données de langage source, pourra être atteint lors de la première traduction.

1.3.2. *La traduction des déclarations des modes en* ℓi_1

Les langages ALGOL 68 et PASCAL offrent la possibilité de représenter un mode (respectivement un type) par un symbole appelé indicateur de mode (respectivement identificateur de type). Afin de simplifier la rédaction, nous limiterons les explications au traitement des déclarations de mode d'ALGOL 68; mais ce qui suit s'applique également au langage PASCAL. Les indicateurs de mode présentent l'intérêt de simplifier l'écriture de certaines déclarations.

En particulier dans le cas où un programmeur veut déclarer en plusieurs endroits du programme des identificateurs de même mode, il lui serait fastidieux de réécrire à chaque fois la définition de ce mode. Mais les déclarations de mode constituent un facteur de la puissance du langage ALGOL 68. En effet les indicateurs de mode qui figurent dans une déclaration de mode peuvent apparaître en partie droite d'une autre déclaration de mode, ou de la même déclaration de mode : le langage autorise sous certaines conditions (voir paragraphe 1.3.3) la récursivité des modes, on obtient ainsi des modes qu'il serait impossible de déclarer autrement. (Notons que la récursivité des modes est interdite en PASCAL).

La simplicité et la clarté de l'écriture au contraire ne sont pas les objectifs fondamentaux du langage ℓi_1 car la syntaxe de ce langage doit être le reflet de la solution apportée au traitement des informations contextuelles. Par conséquent, pour la traduction des déclarations d'identificateurs dont le mode est représenté par un indicateur de mode, nous adopterons la règle suivante : l'expansion de la définition de ce mode m est effectuée dans la déclaration ℓi_1 de l'objet. Si ce mode fait appel à un autre mode m', on fera l'expansion également du mode m'.

La syntaxe du langage ℓi_1 ne comporte donc pas de déclaration de mode ; seules les déclarations d'objets comprenant la description de leur mode appartiennent à la syntaxe du langage ℓi_1 .

1.3.3. *Le traitement des modes récursifs.*

Le seul problème qui se pose réellement pour réaliser l'objectif précédent est le traitement des modes récursifs.

Les restrictions imposées par le langage ALGOL 68 lui-même sur la récursivité des modes, nous permettent de mettre en évidence la solution à ce problème. Deux écritures sont possibles :

- soit l'utilisation du symbole rep devant l'indicateur de mode qui figure en partie droite de la définition d'un mode récursif.
- soit l'utilisation du symbole rep au début de la définition de ce mode.

En ALGOL 68 la déclaration de mode suivante est incorrecte mode a = struct (réel x, a suivant) car après des déclarations d'identité telles que

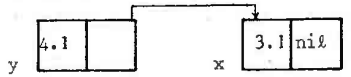
a x = (3.1, fant) ; a y = (4.1, x)

Les valeurs désignées par x et y auraient en mémoire des représentations différentes

Pour éviter cet inconvénient on pourra par exemple déclarer le mode a de la façon suivante :

mode a = struct (réel x, rep a suivant)

Le deuxième champ d'un objet de mode a à l'implantation est un pointeur



On constate que dans ce cas la construction repère d'ALGOL 68 est traduite par un mode primitif pointeur (ou adresse) de ℓi_1 . Ce résultat peut-il être généralisé, c'est ce que nous allons examiner dans le paragraphe suivant en traitant les noms d'ALGOL 68.

1.3.4. *Le traitement de la construction repère d'ALGOL 68.*

Le traitement des synonymes d'ALGOL 68 constitue un autre cas d'application de la construction repère. Soit en ALGOL 68 la déclaration de nom suivante : rep réel z = t [i] . Cette déclaration signifie que z désigne le nom du $i^{ème}$ élément du tableau t. Elle ne constitue pas la déclaration d'un nouvel objet, puisque à l'implantation la place occupée par z et la place occupée par t [i] sont identiques. Remarquons alors que le mode rep réel associé à l'identificateur z sert uniquement à vérifier que les utilisations de z dans les instructions sont valides.

Dans une implantation classique, on associe à z une zone de mémoire contenant son nom puisque ce nom est dynamique.

Ici on peut générer en ℓi_1 le code du calcul de l'adresse de t [i] par une série de triplets. Au cours de la traduction, on mémorise dans les tables le numéro du triplet ayant pour résultat cette adresse, et lorsqu'on rencontre une référence au nom de z dans les instructions, on remplace ce nom par la référence à ce triplet. Nous reprendrons page 30 la traduction précise de cet exemple.

Remarquons que la solution ainsi adoptée permet d'éviter de mémoriser dans le code généré les noms associés aux identificateurs. Une variable de mode rep ent d'ALGOL 68 pourra donc être traitée comme une variable de type entier des autres langages (dans la zone associée à cette variable on trouve directement la valeur).

Cette optimisation peut être étendue aux objets d'ALGOL 68 pour lesquels la construction repère est appliquée en nombre répété.

Soit la déclaration rep ent zz ; zz est de mode rep rep ent. A la traduction en ℓi_1 on pourra déclarer un objet de mode pointeur. Soit l'affectation zz : = z ; zz a pour valeur le nom de z, il suffira de générer en ℓi_1 une affectation sur des adresses.

Si on veut atteindre la valeur pointée, il faut effectuer un certain nombre d'opérations de derepérage. Pour traduire l'affectation suivante d'ALGOL 68 z : = zz + 1, on écrira symboliquement

z : = dereperer (dereperer (zz)) + 1

Le mode primitif de la valeur pointée ent est conservé dans les tables durant la traduction, pour effectuer les contrôles de validité et générer les opérations portant sur la valeur pointée avec l'indication du mode primitif.

Nous reviendrons plus loin sur la traduction exacte de ces exemples.

1.3.5. *Le traitement de la construction procédure des langages algorithmiques.*

Tous les langages algorithmiques autorisent la description d'objets constants de mode procédure. Dans ALGOL 68, on a généralisé ce concept en introduisant les variables de mode procédure.

Une procédure est une construction qui s'applique aux modes des paramètres formels. Est-il nécessaire de conserver la description des modes des paramètres formels ? Si la réponse est non, cela signifie que procédure n'est pas une construction de ℓi_1 .

Les paramètres formels d'une procédure, permettent la communication entre le corps de procédure et l'endroit du programme où elle est activée. Ils font intervenir à l'exécution du corps de procédure , des quantités qui n'y sont pas normalement accessibles, les arguments. A la traduction il faut vérifier que pour chaque appel les types des paramètres effectifs correspondent aux types des arguments. Ceci peut être effectué dans la plupart des cas de façon statique pour les langages que nous considérons, en conservant dans les tables les types des paramètres formels.

Les paramètres formels n'ont en général pas de zone réservée localement à la procédure (sauf dans le cas du passage par valeur d'ALGOL 60 car le rapport ALGOL 60 précise que les paramètres formels déclarés par valeur doivent être traités comme des variables locales à la procédure).

Par conséquent l'information concernant leur type ne sert pas à construire la place qui leur est réservée. Elle est utile évidemment pour traiter les références aux paramètres formels dans le corps de la procédure, dans la suite du processus de traduction. Le type des paramètres formels doit apparaître en ℓi_1 dans les opérations ayant un paramètre formel comme opérande. Ceci est un problème de traduction et non un problème de langage intermédiaire : il suffit de conserver à la traduction le type du paramètre formel dans les tables.

Pour traiter en ℓi_1 , le cas d'un paramètre formel d'ALGOL 60 spécifié par valeur, on effectuera pour celui-ci une déclaration d'un objet local à la procédure.

Ainsi il est inutile de spécifier en ℓi_1 les types des paramètres formels, un objet constant de mode procédure peut être considéré comme un mode primitif de ℓi_1, nous l'appelerons dans la suite du texte mode primitif texte de routine.

En conséquence, comment traduire les variables de mode procédure d'ALGOL 68. La valeur d'une variable de mode procédure est un texte de routine, on peut considérer que le texte de routine est la valeur pointée ; dans ces conditions on peut décrire une variable procédure en ℓi_1 comme un objet de mode primitif adresse.

1.3.6. *Les modes de* LI_1 .

En résumé, on peut dire que le langage ℓi_1 possède des modes primitifs qui sont :

- l'union des modes primitifs des langages algorithmiques,
- les modes primitifs adresse et texte de routine.

Sur ces modes primitifs s'appliquent en ℓi_1 les constructions tableau, structure, ensemble.

Les objets de ℓi_1 sont des variables. Les seules constantes de LI_1 sont les textes de routine qui font l'objet d'une déclaration particulière.

1.4. La désignation des objets en ℓi_1

L'ensemble des langages source que nous désirons pouvoir traduire est constitué par des langages procéduraux à structure de blocs ainsi que FORTRAN.

1.4.1. *Rappel des mécanismes d'adressage classiques des langages procéduraux à structure de blocs.*

Activer une procédure c'est théoriquement insérer le corps de la procédure à la place de l'appel et l'exécuter après avoir fait le remplacement des paramètres par les arguments.

La récursivité des procédures dans les langages à structure de blocs implique que le corps de chaque procédure soit compilé en un sous-programme réentrant. Pour cela il faut créer à chaque appel d'une procédure une zone de travail dont elle se servira. Cette zone de travail contient les données locales à la procédure ainsi que les mémoires du travail qu'elle utilise. L'adresse de début de cette zone s'appelle adresse de base de la procédure. L'adressage final des données doit s'effectuer relativement à l'adresse de base. Il est possible de réserver la place des objets dont la taille est connue à la compilation, pour ceux dont la taille est dynamique, on réserve une zone appelée vecteur de renseignements qui comporte les renseignements utiles pour réaliser l'adressage de ces objets. L'existence de la structure des blocs permet l'utilisation

d'identificateurs locaux et l'utilisation répétée d'un même symbole pour désigner des objets différents. Afin d'attribuer à chaque signification une adresse différente on peut construire une fonction d'adressage à trois ou deux paramètres.

1.4.1.1. *L'adressage classique à trois paramètres.*

Chaque objet est désigné par trois paramètres

- np niveau d'imbrication de la procédure à laquelle appartient l'objet à adresser.
- nb niveau d'imbrication du bloc.
- adr adresse de chaque donnée relative à chaque début de procédure

Cette adresse relative est évidemment fonction de l'encombrement réel des données.

A l'exécution un calcul d'adresse est fait à partir de ces trois paramètres : on repère dans deux piles BLOC et PROC placés en pile dynamique le début d'implantation des variables de chaque bloc et de chaque procédure.

La fonction d'adressage est :

$$f(np,nb,ad) = BLOC(PROC(np)+nb) + ad$$

Cette méthode d'adressage nécessite de mémoriser deux piles et d'effectuer à l'exécution deux indirections. L'encombrement maximal de données est l'encombrement maximal des blocs.

1.4.1.2. *L'adressage classique à deux paramètres.*

Il est possible de concevoir un adressage plus rapide à l'exécution et comportant seulement deux paramètres (np,adr). Pour tenir compte de la structure de blocs l'adresse relative est calculée conformément à l'algorithme suivant :

- à la lecture d'une instruction d'entrée de procédure, on réinitialise la valeur de l'adresse relative à 1,
- à l'entrée d'un bloc, on mémorise la dernière valeur de l'adresse relative avant l'entrée dans ces blocs. Après la lecture de chaque déclaration on incrémente la valeur de l'adresse relative de la taille de la donnée déclarée
- à la sortie d'un bloc, on remet à jour l'adresse relative en lui affectant la valeur qu'elle possédait avant l'entrée dans ce bloc.

Cette méthode ne nécessite qu'une pile de traitement appelée le display procédures, par contre, l'encombrement maximal à réserver est égal au maximum de la somme des encombrements des blocs imbriqués les uns dans les autres.

1.4.2. *La désignation à deux paramètres des objets en* ℓi_1 .

Quelque soit le langage source, un objet est désigné par un identificateur par analogie on peut dire qu'une adresse est la désignation en langage d'assemblage d'un objet.

Nous appellerons "descriptif" la désignation en ℓi_1 d'un objet. Cette désignation doit refléter uniquement les caractéristiques du langage source : on peut donc choisir soit les trois paramètres, soit les deux paramètres. Nous choisirons une désignation à deux paramètres dans le but de prévoir un adressage final plus rapide à l'exécution. Les deux paramètres sont :

- le niveau d'imbrication de procédure np,
- le rang rg : le rang de chaque objet est évalué en affectant à chaque objet une unité de "mémoire abstraite" car la taille réelle des données ne peut être prise en compte ici.

L'algorithme de calcul du niveau d'imbrication de procédure np est :

- la procédure externe porte le niveau de procédure 0 (nous ne traiterons pas ici le problème de la compilation séparée des procédures, nous supposerons donc que tout programme à traduire se compose d'une seule procédure externe).
- quand on lit une instruction de déclaration de procédure incrémenter np de une unité.
- quand on lit une instruction de fin du corps d'une procédure décrémenter np de une unité.

L'algorithme de calcul du rang est identique à l'algorithme de calcul de l'adresse relative dans l'adressage classique à deux paramètres à la différence près qu'on ne peut tenir compte ici de la taille réelle des données. On accroît donc le rang de une unité après la lecture de chaque déclaration de donnée.

A titre d'exemple, montrons quelles sont les valeurs des descriptifs des objets du programme ALGOL 60 suivant :

```
begin integer a,b,c;          descriptif(A) = (0,1) descriptif(Z) = (0,2) descriptif
                                                                          (C) = (0,3)
procedure p(x); integer x;    descriptif(P) = (0,4)
begin real y; integer i;      descriptif(Y) = (1,1) descriptif(I) = (1,2)
procedure g(u); real u;       descriptif(G) = (1,3)
begin real z;                 descriptif(Z) = (2,1)
  -
  -
end;
procedure h(t); integer t;              descriptif (H) = (1,4)
begin integer array tab (1:10,1:10)     descriptif (TAB) = (2,1)
  -
  -
end;
end;
```

Remarquons que les procédures sont traitées comme n'importe quel autre objet donc qu'elles possèdent un descriptif.

2. LA DECLARATION DES DONNEES EN LI 1

2.1. La syntaxe de déclaration

La syntaxe des déclarations de tous les objets en ℓi_1 débute par le code opération ALLC, sauf pour les textes de routines qui comportent la traduction du corps de la procédure. Nous avons déjà indiqué au paragraphe 1.2.3 qu'il n'était pas nécessaire de faire en ℓi_1 la distinction entre constantes et variables car à ce niveau tous les objets sont traités comme des variables. Au paragraphe 1.3.4 on a montré qu'il n'était pas nécessaire de conserver les noms de chaque objet parce qu'on peut ramener dans tous les cas la construction repère à un mode primitif adresse en ℓi_1.

La syntaxe de déclaration des données en ℓi_1 est présentée dans les lignes qui suivent. Conformément aux notations utilisées chez I.B.M., nous plaçons entre crochets les unités syntaxiques dont l'occurrence peut apparaître ou non. Lorsqu'un non terminal apparait entre accolades, l'accolade droite étant suivie d'une astérisque, cela s'interprète comme une suite de ce non terminal : exemple pour {<dim>}* il faudrait écrire en forme normale stricte de Backus <ldim>=<dim>|<ldim><dim> la désignation d'un objet est le descriptif, ou un triplet d'entiers quand il s'agit d'un paramètre formel ou du résultat qui possèdent une zone réservée dans la procédure.

<déclaration> = ALLC <descriptif> <déclareur effectif> |
ALLC (entier,entier,entier) <déclareur effectif>

<texte de routine> = ROUTINE <descriptif> entier ; <ℓtrip> ; END|
ROUTINE * (entier,entier,entier)entier; <ℓtrip> ; END

<déclareur effectif> =
<mode primitif> | <déclareur effectif de tableau> | <déclareur effectif de structure> | <déclareur d'ensemble>

<déclareur effectif de tableau> =
TAB [COMPRESSE] entier ; <dim> ; [{<dim>;}*] <déclareur effectif>

<dim> = DIM E <borne> , <borne>

<borne> = <constante entière> | <ℓdereperage> <descriptif> |* (entier T)|

<ℓdereperage> = <derep> {<derep>}*

<derep> =*

<déclareur effectif de structure> =
STRUCTURE [COMPRESSE] (entier,entier) <champ effectif> ; {<champ effectif>

<champ effectif> = (entier,entier) <déclareur effectif>

<déclareur d'ensemble> = ENS <déclareur effectif>

Remarque : Cette syntaxe est conforme au principe d'orthogonalisation des modes. Remarquons qu'elle n'utilise que des déclareurs effectifs. Rappelons que les déclareurs formels ont été introduits dans la syntaxe d'ALGOL 68 pour

spécifier les modes des paramètres formels. Comme en ℓi_1 on ne déclare pas les paramètres formels (la mémorisation de leur mode a été faite par le traducteur), il n'apparaît pas dans la syntaxe de ℓi_1 des déclareurs formels.

Les modes primitifs du langage ℓi_1 sont le mode primitif adresse et l'union de modes primitifs des langages source. Ces derniers pourront être décrits par l'implanteur au moment de l'écriture de la première traduction.

Dans les paragraphes qui suivent nous décrivons le mode primitif adresse et nous donnerons des exemples de description des modes primitifs des différents langages source.

Puis nous décrivons les différentes constructions de ℓi_1 : construction tableau, construction structure, construction ensemble. Dans ce chapitre consacré à la description des déclarations de données, nous ne décrirons pas les textes de routine. Pour la commodité de l'exposé, il est plus clair de présenter en même temps la déclaration et l'appel des routines ou la traduction de l'appel des routines fait intervenir des affectations ou des identités qui seront décrites au paragraphes 4 et 5. Le chapître 6 sera consacré aux textes de routine.

2.2. Le paramètre de ℓi_1 mode primitif

2.2.1. *Le mode primitif adresse.*

Le mode primitif adresse sert à décrire les pointeurs de PL 1, et la construction repère d'ALGOL 68. Les exemples intéressants d'utilisation de la construction repère font intervenir les autres constructions, c'est pourquoi nous les décrirons plus loin.

2.2.2. *La description des modes primitifs d' ALGOL 60*

ALGOL 60 n'autorise que les modes primitifs arithmétique ou logique

<mode primitif(ALGOL 60)> = E|R|L

On pourra par exemple coder le mode primitif 'integer' par la chaîne E, le mode primitif 'real' par la chaîne R, le mode primitif 'boolean' par la chaîne 'L'.

Exemple

boolean x; ALLC (np_x,rg_x)L
integer u; ALLC (np_u,rg_u)E

2.2.3. *La description des modes primitifs de* PL 1.

PL 1 autorise les modes primitifs arithmétiques, logique, étiquette, pointeur et chaîne.

Les modes primitifs arithmétiques possèdent quatre paramètres, la base, le mode, l'échelle et la précision :

- la base est soit décimale, soit binaire ;
- le mode est soit réel, soit complexe ;
- l'échelle est soit la virgule fixe, soit la virgule flottante ;
- la précision indique le nombre de chiffres que la donnée arithmétique peut comporter.
  - *en virgule fixe la précision spécifie la position de la virgule par rapport au chiffre le plus à droite du nombre
  - *en virgule flottante la précision est le nombre minimal de chiffres significatifs (exposant non compris) qui doit être conservé.

Exemple

DCL A FIXED DECIMAL (5,4)

est la déclaration d'un nombre décimal ne comportant pas plus de 5 chiffres dont 4 à droite de la virgule

DCL B FIXED DECIMAL (6)

est la déclaration d'un nombre entier ne comportant pas plus de 6 chiffres.

En LI_1 il faut évidemment conserver toutes ces informations et décrire complètement le mode primitif arithmétique.

On pourra traduire les déclarations de A et B par :

ALLC (np_a, rg_a) REAL FIX DEC (5,4)

ALLC (np_b, rg_b) REAL FIX DEC (6)

Le mode primitif pointeur de PL 1 sera traduit en ℓi_1 par le mode adresse.
Le mode primitif étiquette de PL 1 sera traduit en ℓi_1 par le mode Et.
On en verra l'application dans le chapitre des structures de contrôle de ℓi_1 .
Le mode primitif chaîne de PL 1 sera également décrit en ℓi_1 par un mode primitif chaîne avec par exemple :

<mode chaine> = <fvar> <nature> <valeur entière>
<fvar> = FIX|VAR chaîne de longueur constante ou
chaîne de longueur variable mais bornée
<nature> = BIT|CH chaîne des bits ou de caractères
<valeur entière> = <constante entière> | entier T

Exemples de traduction :

dcl n character (10) varying (1) ALLC(np_n, rg_n) VAR CH 10 C
dcl id bit (i + j) varying (2) + REAL FIX BIN (15) *(np_i, rg_i), *(np_j, rg
(3) ALLC (np_{id}, rg_{id}) VAR BIT 2 T

Dans le dernier exemple la longueur maximale de la chaîne est dynamique, il s'agit de la valeur de i+j, qui est calculée en ℓi_1 par le triplet 2.

2.2.4. *La description des modes primitifs de PASCAL.*

PASCAL autorise les modes primitifs standards entier, réel, logique, caractère. Une chaîne de caractères en PASCAL n'est pas considérée comme un mode primitif, elle est traitée comme un tableau de caractères.

Nous traiterons les types scalaires définis par le programmeur comme des modes primitifs car les constructions définies plus haut s'appliquent également à eux.

En PASCAL un type scalaire est une suite de valeurs, désigné par un identificateur. Exemple :

type couleur = (blanc, rouge, bleu)
type jour = (lundi, mardi, mercredi, jeudi, vendredi, samedi)

la suite des valeurs représenté par un type scalaire est ordonné.

Lors de la traduction de PASCAL en ℓi_1 , on numérote les types scalaires définis, et on code les valeurs de chaque type par une suite de nombres entiers compris entre 1 et n (n étant le nombre de valeurs discrètes). Afin de simplifier l'écriture du codage des valeurs discrètes, on note [1 ... n] ces valeurs ce qui signifie que toutes les valeurs des entiers successifs compris entre 1 et n sont valides.

type couleur = (blanc, rouge, bleu)
var x : couleur ; (1) ALLC (np_x, rg_x) CODE 1 [1 ...3]
type jour = (lundi, mardi ... dimanche)
var j : jour (2) ALLC (np_j, rg_j) CODE 2 [1 ...7]

2.2.5. *La description des modes primitifs d'ALGOL 68.*

ALGOL 68 autorise les modes primitifs arithmétiques, booléens, binaires et alphanumériques. Les modes primitifs ent et réel peuvent être complétés en spécifiant la précision souhaitée par l'adjonction en nombre répété de court ou long : long ent est un déclareur de mode entier de double longueur. L'implanteur d'ALGOL 68 devra formaliser les précisions en ℓi_1 , puis à la seconde étape de traduction spécifier quel est le plus grand nombre entier admis. Le mode primitif bool permet de spécifier des valeurs logiques, le mode primitif bit des chaînes de bit de longueur bloquée (fixée lors de la deuxième étape de traduction).

Le mode primitif car permet de déclarer une valeur de caractères, le mode primitif cat une suite de caractères de longueur bloquée ; le mode primitif chaîne une suite de caractères de longueur quelconque. La traduction des déclarations d'objets élémentaires d'ALGOL 68 pourra se faire par exemple de la façon suivante :

ent x,	ALLC (np_x, rg_x) E
long long reel a ;	ALLC (np_a, rg_a) L 2R
variables réelles de triple longueur	
bit face ;	ALLC (np_{face}, rg_{face}) BIT
bool flip	ALLC (np_{flip}, rg_{flip}) BOOL
ent i = 3 ;	
ent k ;	(1) ALLC (np_k, rg_k) E
ent ℓ ;	(2) ALLC (np_ℓ, rg_ℓ) E
ent j = k+ℓ;	(3) + E *(np_k, rg_k), *(np_ℓ, rg_ℓ)
	(4) ALLC (np_j, rg_j) E
ℓ : j+i ;	(5) : = E 3T, (np_j, rg_j)
	(n) + E *(np_j, rg_j), 3C
	(n+1) : = E nT, (np_ℓ, rg_ℓ)

La déclaration de la constante i ne donne lieu à aucune génération de triplet, on propagera seulement la valeur de la constante dans la suite du texte (en particulier dans la traduction de l'affectation ℓ : = j+i). La valeur de la constante j est calculée, il faut réserver une mémoire pour j, on génère un triplet de code opération ALLC.

2.2.6. *Les conversions.*

Les expressions arithmétiques des langages source peuvent comporter des opérandes de modes arithmétiques différents. Les règles de conversions à appliquer à certains de ces opérandes sont le plus souvent implicites. Cependant en ALGOL 68, il n'existe pas de modification qui convertirait automatiquement une valeur complexe en une valeur réelle ou une valeur réelle en valeur entière. En effet il n'existe pas en général de réel équivalent à un complexe, ni d'entier équivalent à un réel. Le choix de la valeur équivalente est laissé aux soins du programmeur par l'utilisation d'opérateurs monadiques de conversions : les opérateurs entier et arrd permettent de passer d'une valeur réelle à une valeur entière.

En PL I les modifications sont faites automatiquement par le compilateur quand les conversions sont implicites dans le programme source. Mais le programmeur a également le droit de choisir la modification arithmétique qui lui convient en utilisant les fonctions incorporées TRUNC, CEIL, FLOOR pour choisir la valeur entière équivalente à une valeur réelle donnée.

Remarquons que la modification entier d'ALGOL 68 est identique à la fonction incorporée FLOOR de PL I.

Les opérateurs de conversion de ℓi_1 doivent rassembler ces différentes modifications.

En PL I la définition d'un mode arithmétique comporte la description du mode (REAL, COMPLEX), de la base (BINARY ou DECIMAL) de l'échelle (FIXED ou FLOAT), de la précision. Les deux opérandes d'une opération arithmétique peuvent différer en base, mode, précision et échelle. Lorsqu'ils diffèrent une conversion a lieu suivant des règles très précises. Rappelons en les grandes lignes :

a/Si les bases des deux opérandes sont différentes le facteur décimal est converti en binaire.

b/Si les modes des deux facteurs sont différents l'opérande de mode réel est converti en mode complexe.

c/Si les échelles des deux opérandes sont différentes, l'opérande en virgule fixe est converti en virgule flottante. Il existe une exception à cette règle dans le cas d'une élévation à une puissance.

d/Les règles de calcul de la précision du résultat varient suivant l'opérateur concerné. A titre d'exemple rappelons que pour une addition ou une soustraction, la précision (p,q) du résultat se calcule de la façon suivante

$p = 1 + \max(p_1-q_1, p_2-q_2) + \max(q_1,q_2)$

$q = \max(q_1,q_2)$

où p représente le nombre total de chiffres du résultat,
q le facteur d'échelle du résultat.

Les règles de conversion de la base, du mode, de l'échelle, de la précision peuvent être appliquées statiquement par le traducteur. Par exemple si i et j ont été déclarés :

dcℓ i fixed decimal (9,4)
dcℓ j fixed decimal (8,5)

et que l'on trouve dans une expression arithmétique l'opération i + j. Le traducteur peut calculer la précision du résultat à l'aide des règles précédentes, ce qui donne (11,5). Il va donc pouvoir générer les conversions correspondantes des valeurs de i et j. On pourra écrire par exemple en ℓi_1 .

(n)	CV REAL FIX DEC ((9,4),(11,5)) *(np_i, rg_i)
(n+1)	CV REAL FIX DEC ((8,5),(11,5)) *(np_j, rg_j)
(n+2)	+ REAL FIX DEC (11,5) n T, n+1 T

On vérifie que les conversions sont paramétrées ici par les caractéristiques du mode arithmétique.

En ALGOL 60 les modes arithmétiques ne comportent que le mode entier ou le mode réel. Les opérations de conversion peuvent être soit conversion d'une valeur entière en une valeur réelle, soit conversion d'une valeur réelle en une valeur entière.

La paramétrisation de l'opérateur ℓi_1 des conversions est donc plus simple, on pourra écrire par exemple :

CVER qui signifie conversion d'une valeur entière en une valeur réelle

CVRE qui signifie conversion d'une valeur réelle en une valeur entière

Soit en ALGOL 60 l'affectation s : = a - t/3. Supposons que toutes les variables qui interviennent dans cette affectation aient été déclarées comme entiers. L'occurrence de l'opérateur de division / implique que le résultat de la division soit un réel, le deuxième opérande de la soustraction est un nombre réel, il faut donc convertir la valeur du premier opérande en valeur réelle. L'expression évaluée a-t/3 a ainsi un mode réel, comme la valeur à laquelle on l'affecte est de mode entier, il faut faire une conversion de réel en entier avant de traduire l'affectation.

(1) CVER *(np_t, rg_t)
(2) CVER 3C
(3) / R 1 T, 2 T
(4) CVER *(np_a, rg_a)
(5) - R 4 T, 3 T
(6) CVRE 5 T
(7) : = E 6 T, (np_s, rg_s)

2.3. Description de la construction tableau

Rappelons la syntaxe du déclareur effectif

<déclareur effectif de tableau> =

TAB [COMPRESSE] entier <dimension> ; { <dimension>}* <déclareur effectif>

<dimension> = DIME <borne> , <borne>

<borne> = <constante entière> |*(entier,entier) | entier T | Id

TAB est le code ℓi_1 de la construction tableau

COMPRESSE est un attribut indiquant que les éléments du tableau seront compressés en mémoire à l'implantation (c'est l'attribut PACKED de PL 1 ou PASCAL).

entier est le nombre de dimension du tableau.

En FORTRAN et en PASCAL une borne ne peut être que constante, par conséquent dans la traduction en ℓi_1 , on ne pourra atteindre que l'alternative

<constante entière>

PASCAL

Var m : array [0...10] of integer
(1) ALLC (np_m, rg_m) TAB 1
(2) DIM E 0C, 10 C
(3) E

Var mm : array [1...10] of array [3...7] of integer
(1) ALLC (mp_{mn}, rg_{mn}) TAB 1
(2) DIM E 1 C, 10 C
(3) TAB 1
(4) DIM E 3C, 7C
(5) E

Var t : packed array [0...10] of char est la déclaration d'une chaîne de caractère de longueur 11
(1) ALLC (np_t, rg_t) TAB COMPRESSE 1
(2) DIM E 0C, 10 C
(3) CH

En PL 1 une borne peut être une constante, une variable, ou le résultat d'une expression (dans ce cas on évalue d'abord l'expression, la description de la borne comporte alors la référence au triplet résultat de l'expression).

PL 1

dcℓ u (1: i,1:j) char (k-ℓ) varying ;
(1) - REAL FIX BIN(15) *(np_k,rg_k),*(np_1,rg_1)
(2) ALLC (np_u, rg_u) TAB 2
(3) DIM E 1 c, *(np_i, rg_i)
(4) DIM E 1 C, *(np_j, rg_j)
(5) VAR CH 1 T

En ALGOL 60 une borne peut être une constante entière, une variable, une expression arithmétique simple ou une expression conditionnelle. Dans ce dernier cas suivant le résultat du test cette borne peut prendre deux valeurs (et davantage dans le cas de IF imbriqués).

Puisque nous nous imposons de générer la traduction de l'expression de test avant la description du tableau, il faut mémoriser la valeur de cette borne dans une mémoire temporaire.

Le traitement de ce cas particulier d'ALGOL 60 nous oblige à réintroduire les noms au niveau du langage ℓi_1 (mais ils ne s'identifient jamais aux objets du langage source). Ces noms sont les désignations de mémoires temporaires qui possèdent un type et une zone réservée à la suite des données locales à la procédure, cette zone est libérée après la dernière utilisation de cette mémoire de manoeuvre.

integer array t (1 : if x > 3 then i, else p, 1 : j)

(1) >E * (np_x, rg_x), 3 c
(2) SINON 6 T
(4) = E * (np_i, rg_i), TEMP 1
(5) ALLERA 7 T
(6) = E *(np_p, rg_p), TEMP 1
(7) ALLC (np_t, rg_t) TAB 2
(8) DIM E 1 C, TEMP 1
(9) DIM E 1 C *(np_j, rg_j)

Le premier triplet effectue la comparaison entre la valeur de x, et la valeur 3. Si ces valeurs sont différentes, on va se brancher au triplet 6, où on mémorise la borne supérieure de la première dimension dans TEMP 1 ; sinon on continue en séquence au triplet 4.

Dans tous les cas la déclaration du tableau en ℓi_1 commence au triplet 7.

En ALGOL 68, une borne est une proposition unitaire, en ℓi_1 on génère la traduction de la proposition unitaire avant la déclaration proprement dite du tableau.

L'exemple ci-dessous montre quelle est la traduction de la déclaration d'un rang de nombres entiers.

[10] ent t, (1) ALLC (n_p, rg_t) TAB 1
(2) DIM E 1 C, 10 C
(3) E

L'exemple qui suit montre la traduction de la déclaration d'un tableau de rang de 5 caractères.

[10, 0 : a+b] [5] car x, (1) + E *(np_a, rg_a), *(np_b, rg_b)
(2) ALLC (np_a, rg_a) TAB 2
(3) DIM E 1 C, 10 C
(4) DIM E 0 C, 1 T
(5) TAB 1
(6) DIM E 1 C, 5 C
(7) CH

2.4. La construction structure

Cette construction existe dans les langages PL 1, PASCAL et ALGOL 68.

Rappelons la syntaxe d'un déclareur effectif de structure.

<déclareur effectif de structure> =
STRUCTURE [COMPRESSE] (entier,entier); <champ effectif>, { <champ effectif>}*
<champ effectif> = (entier,entier) [COMPRESSE] <déclareur effectif>

Un des principaux objectifs de la traduction du langage source en langage ℓi_1 est la suppression de la plupart des informations contextuelles ; c'est pourquoi nous mémorisons la fin de la génération de chaque sous-structure.

STRUCTURE (entier,entier) marque le début de la génération d'une sous-structure (explicité par le premier nombre entier), la fin de la génération de la sous-structure est le triplet dont le rang est le deuxième nombre entier. Pour la description de chaque champ on précise le début et la fin de la génération dans le doublet (entier,entier).

Afin de clarifier l'exposé, nous allons faire une remarque concernant les modes des sélections d'un identificateur du mode structuré.

Soit en ALGOL 68 la déclaration d'identificateur suivante :

struct (ent no, reel prix) article ;

l'identificateur article a pour mode rep struct (ent no, reel prix), no de article a pour mode rep ent, mais en ℓi_1 comme en ALGOL 68 la description du champ no aura pour mode entier ; par contre une référence à la selection no de article aura pour mode rep ent

ALGOL 68	ℓi_1
struct (ent no, reel prix) article ;	(1) ALLC ($np_{article}$,$rg_{article}$) STRUCTURE (1,3) (2) (2,2) E
no de article : = 3 ;	(3) (3,3) R – (n) := E 3 C, ($np_{article}$, $rg_{article}$, (2,2))

Exemple 2

struct (ent no, ent nb, [1:7] struct (chaîne cheflieu, ent pop)ar)d ;

(1) ALLC (np_d, rg_d) STRUCTURE (1,8)
(2) (2,2) E ;
(3) (3,3) E ;
(4) (4,7) TAB 1 ;
(5) DIM E 1 C, 7 C ;
(6) STRUCTURE (6,8) ;
(7) (7,7) CHAINE ;
(8) (8,8) E ;

Exemple 3

mode article = struct (ent i, rep article p) :
article dep, dep 1 ;
p of dep : = dep 1 ;

Cet exemple comporte une déclaration de mode structuré d'indicateur article, deux déclarations d'identificateurs dep et dep_1 de mode article. En ℓi_1 on déclare deux données dont le mode est la traduction du mode article

(1) ALLC (np_{dep}, rg_{dep}) STRUCTURE (1,3)
(2) (2,2) E
(3) (3,3) A
(4) ALLC (np_{dep_1}, rg_{dep_1}) STRUCTURE (4,6)
(5) (5,5) E
(6) (6,6) A
(7) := A (np_{dep_1}, rg_{dep_1}), ((np_{dep}, rg_{dep}), (2,2))

le deuxième champ du mode structure est un nom pour traduire la récursivité.

Exemple de traduction d'une structure PL 1

```
DCL 1 Z                              (1) + E * (np_ii, rg_ii), *(np_jj, rg_jj)
    2 U                              (2) ALLC (np_z, rg_z) STRUCTURE (2,11)
      10 A FIXED DECIMAL (6)         (3) STRUCTURE (3,5)
      10 B CHARACTER (II+JJ)         (4) (4,4) REAL FIX DEC (6)
    2 V                              (5) (5,5) FIX CH 1 T
      6 C  FIXED DECIMAL (8)         (6) STRUCTURE (5,10)
      6 D  (1 : x,1 : y) BIN FIXED   (7) (6,6) REAL FIX DEC (8)
                                     (8) (8,10) TAB 2
                                     (9) DIM E 1 C,  *(np_x, rg_x)
                                    (10) DIM E 1 C,  *(np_y, rg_y)
                                    (11) REAL FIX BIN (15)
```

Les déclarations des niveaux disparaissent, elles sont remplacées par le code STRUCTURE suivi par un couple d'entiers indiquant le premier et le dernier triplet généré pour la structure correspondante.

Les feuilles d'une structure PL 1 sont des variables, dans la traduction ℓi_1 on ne décrit que le mode du champ comme en ALGOL 68, mais une sélection a pour mode rep (mode du champ).

Par exemple la sélection Z.A. a bien pour mode rep ent.

Autre exemple de traduction d'un tableau de structures de PL 1

```
DCL 1 METEO (12)                     (1) ALLC (np_meteo, rg_meteo) TAB 1
    2 TEMPERATURE (31)               (2) DIM E 1  C, 12 C
      3 MAXI FIXED DEC (5,3)         (3) (3,13) STRUCTURE (3,13)
      3 MINI FIXED DEC (5,3)         (4) (4,8) TAB 1
    2 PRECIPITATIONS                 (5) DIM E 1 C, 31 C
      3 CUMUL MOIS FIXED DEC (6,2)   (6) (6,8) STRUCTURE (6,8)
      3 JOURNEE (31) FIXED DEC (6,2) (7) (7,7) FIXED DEC (5,3)
                                     (8) (8,8) FIXED DEC (5,3)
                                     (9) (9,13) STRUCTURE (9,13)
                                    (10) (10,10) FIXED DEC (6,2)
                                    (11) (11,13) TAB 1
                                    (12) DIM E 1 C, 31 C
                                    (13) FIXED DEC (6,2)
```

Exemple de traduction d'un enregistrement de PASCAL

```
type date = record mo : (janv,fev,...)   (1) ALLC (mp_x, rg_x) STRUCTURE (1,4)
            day : 1...31                 (2) (2,2) CODE 1 [1...12]
            year : integer               (3) (3,3) CODE 2 [1...31]
            end                          (4) (4,4) E
var x : date
```

2.5. Description de la construction ensemble

<déclareur d'ensemble> = ENS <déclareur effectif >

La syntaxe de ℓi_1 autorise ainsi toutes les compositions comportant la construction ensemble. Ceci est une extension du langage PASCAL qui n'autorise que les ensembles de valeurs de type scalaire ou rang.

```
                PASCAL                          li1
type jour = (lundi,mardi,...dimanche)  (1) ALLC (mp_s, rg_s) ENS
type semaine : set of jour             (2) CODE 1 [1...7]
                                       (3) ALLC (np_travail, rg_travail) ENS
Var s, travail, libre : semaine        (4) CODE 1 [1...7]
    j : jour                           (5) ALLC (np_libre, rg_libre) ENS
                                       (6) CODE 1 [1...7]
                                       (7) ALLC (np_j, rg_j) CODE 1 [1...7]
```

les constantes de modes ensemble pourront être décrites de la façon suivante :

C[ ] est la formalisation en ℓi_1 de l'ensemble vide.

C [1 {1...7}] est l'ensemble des valeurs du type jour, c'est la traduction de l'ensemble [lundi, mardi, mercredi...samedi,dimanche]

C [*(np_j, rg_j)] est l'ensemble [j] construit à partir de la variable j.

2.6. La description de la construction repère d'ALGOL 68

Exemple 1 : traduction de la récursivité des modes.

mode article = struct (ent i, rep article p)	(1) ALLC (np_{dep}, rg_{dep}) STRUCTURE (
article dep, dep 1 ;	(2) (2,2) E
	(3) (3,3) A
p of dep : = dep 1	(4) ALLC ($np_{dep\,1}$,rg_{dep1})STRUCTURE (
	(5) (5,5) E
	(6) (6,6) A
	(7) : = A (np_{dep} rg_{dep}) ((np_{dep}, rg_{dep}),(3,3))

le 2ème champ des objets de mode article est un pointeur. Si on veut chaîner les deux listes, on place l'adresse de dep 1 dans la 2ème feuille de la structure dep.

Exemple 2 : la traduction de la synonymie

rep ent z = t [i]	(1) ORIG (np_t, rg_t) TAB 1
rep ent fred = t [j]	(2) IND E * (np_i, rg_i)
	(3) E
z : = fred + 3	(4) ORIG (np_t, rg_t) TAB 1
	(5) IND E* (np_j, rg_j)
rep ent zz ;	(6) E
	(7) + E 3 C, *(6T)
	(8) : = E 7T, 3T
zz : = z,	(9) ALLC (np_{ZZ}, rg_{ZZ}) A
	(10) : = A 6T, (np_{ZZ}, rg_{ZZ})

Lorsque le traducteur lit l'identité rep ent z = t [i], il place dans sa table des symboles l'identificateur z, son mode rep ent et son adresse (qui est l'adre calculée de la variable indicée t [i], c'est-à-dire le numéro de triplet (3). Ensuite il appelle la routine de génération du code ℓi_1 de l'adresse d'une variable indicée : la séquence de triplets (1) à (3) est générée.
Le même processus est appliqué à l'identificateur fred et la séquence de triplets (4) à (6) est générée.
Lorsque le traducteur rencontre l'affectation z : = fred + 3, il doit aller rechercher en table des symboles le nom de z qui est le triplet 3 T et la valeur de fred qui est le contenu de l'adresse 6 T.

Exemple 3. Autre exemple

mode m = [1 : 10] rep ent	(1) ALLC (np_x, rg_x) TAB 1
m x ;	(2) DIM E 1 C, 10 C
ent i ;	(3) A
x [2] :=i ;	(4) ALLC (np_i, rg_i) E
/	(5) ORIG (np_x, rg_x) TAB 1
rep ent (x [2]) : = 100 ;	(6) INDE 2 C
	(7) A
	(8) : = A (np_i, rg_i), 5 T
	.
	.
	.
	(n) : = E 100 C, *(7T)

x a pour mode rep [1:10] rep ent c'est-à-dire que x est un tableau de pointeurs.
L'affectation x [2]: = i signifie que l'on attribue à x 2 l'adresse de i.
Le triplet (8) traduit ce chaînage en ℓi_1.
L'affectation rep ent (x [2]): = 100 ; signifie qu'on attribue la valeur 100 à la mémoire pointée par x [2]. L'adresse de cette cellule en ℓi_1 est * (7T)

3. LES REFERENCES AUX OBJETS

On trouve des références aux objets dans les instructions, instructions qu'on peut en première approximation diviser en instruction de contrôle et instructions d'affectations.

Dans une instruction d'affectation l'objet à affecter et la valeur qu'on affecte ne sont pas traitées de façon symétrique. Nous considérerons l'objet à affecter comme un contenant, en $\&i_1$ nous en prendrons donc l'adresse. Nous considérerons la valeur qu'on affecte comme un contenu (en $\&i_1$ nous effectuerons donc le nombre d'opérations de derepérage nécessaires pour obtenir la valeur).

En résumé, une référence à un objet en $\&i_1$ sera soit la référence à l'adresse de cet objet, soit la référence à la valeur de cet objet.

3.1. Les références aux adresses d'objets.

La référence à l'adresse d'un objet peut être une référence à l'adresse d'un objet pris dans sa totalité (c'est le descriptif, ou le triplet d'un paramètre formel, voir le paragraphe 6.2.3 "désignation de paramètres formels"), ou la référence à l'adresse d'une partie de cet objet (tranche d'un tableau, variable indicée, feuille d'une structure, sous structure). Les références aux noms de procédure qui sont les appels de procédure ne seront pas traitées ici mai dans le chapitre concernant les textes de routines.

<référence adresse> = <reference adresse objet> | <reference adresse element>
<référence adresse objet> = <descriptif> | (entier,entier,entier)
<référence adresse élément> = <adresse variable indicée> | <adresse tranche> |
<adresse sous structure> | <adresse feuille>
<adresse variable indicée> =
ORIG <origine> TAB entier ; <indice> { <indice>; }* <déclareur effectif>
<origine> = <descriptif> | entier T
<indice> = IND E <valeur entière>
<valeur entière> = <constante entière> | *<descriptif>|**<descriptif> | entier T
<adresse tranche> =
TRANCHE entier ORIG <origine> TAB entier ; <rang> { <rang>; }*<déclareur effectif
<rang> = <gamme> | <indice>
<gamme> = IND E <domaine>
<domaine> = ? | <valeur entière>,<valeur entière>
<adresse sous structure> = (<descriptif> , (entier,entier))

3.1.1. *Référence à l'adresse d'un objet global.*

L'adresse d'un objet en $\&i_1$ est le descriptif ou le triplet pour un paramètre formel.

3.1.2. *Référence à une variable indicée ou une tranche.*

Pour obtenir l'adresse d'une variable indicée, il faut décrire sa fonction d'accès :

- l'adresse de l'origine du tableau est décrite par le triplet ORIG <origine> TAB entier.
  Comme le tableau peut être aussi une tranche la désignation de ce tableau se fait soit par le descriptif, soit par l'adresse de la tranche.
- la valeur de chaque indice est décrite par un triplet de syntaxe IND E <valeur entière>

Exemple

integer array u [1 : n, 1 : m]

(1) ALLC (np_u, rg_u) TAB 2
(2) E
.
.
u [i,j]
(n) ORIG (np_u, rg_u) TAB 2
(n+1) IND E * (np_i, rg_i)
(n+2) IND E * (np_j, rg_j)
(n+3) E

L'exemple qui suit montre comment on décrit la référence au nom x [1,3][2] élément du tableau de tableau de caractères dont la déclaration [10,0 : a + b][5] car x ; a été traduite dans la description de la construction tableau au paragraphe 2.3

x [1,3][2]

(1) ORIG (np_x, rg_x) TAB 2
(2) IND E 1 C
(3) IND E 3 C
(4) TAB 1
(5) IND E 2 C
(7) CH

En ALGOL 68, il est possible de sélectionner une tranche d'un tableau c'est-à-dire un de ces sous tableaux. Pour prendre une tranche d'un tableau on fixe les valeurs de certains indices, les autres restent indéterminés (c'est-à-dire variant entre les bornes inférieures et supérieures) soit variant dans un domaine de valeurs plus restreint.

Exemple

[10,10] réel x ;

x [2,] est une tranche.

On peut faire désigner cette tranche par un nouvel identificateur :
rep[ ] réel y = x [2,] . Cette déclaration est en ALGOL 68 une synonymie entre les noms de y et le nom de la tranche x [2,] donc il ne faut pas effectuer des nouvelles allocations pour les valeurs de y. Mais la fonction d'accès des éléments de la tranche n'est pas identique à la fonction d'accès des éléments du tableau car la tranche possède une nouvelle adresse origine et les distances entre les éléments de la tranche ne sont pas forcément identiques aux distances entre les éléments du tableau (suivant que le tableau est implanté ligne par ligne ou colonne par colonne).

Il faut donc construire un nouveau vecteur de renseignements pour la tranche sans allouer de mémoire pour les valeurs de la tranche. La déclaration d'une tranche ne peut se traduire en $\&i_1$ par la déclaration d'un tableau

ALLC (np,r) TAB n

on utilisera un code différent le code TRANCHE. Comme on ne mémorise pas à l'implantation le nom de la tranche on n'effectue pas en $\&i_1$ des désignations de cette tranche par

TRANCHE (np_y, rg_y) TAB 1

mais on précise de quel tableau cette tranche est un sous tableau.

	(1) ALLC (np_x, rg_x) TAB 2
[10,10] réel x,	(2) DIM E 1 C, 10 C
	(3) DIM E 1 C, 10 C
	(4) R
rep[ ] réel y = x [2,]	(5) TRANCHE 1 ORIG (np_x, rg_x) TAB 2
	(6) IND E 2 C
	(7) IND E ?
	(8) R
rep[ ] réel yy = x [2,3 :7]	(9) TRANCHE 2 ORIG (np_x, rg_x) TAB 2
	(10) IND E 2 C
	(11) IND E 3 C, 7 C
	(12) R
yy [4] : = 0 ;	(14) ORIG 12 T TAB 1
	(15) IND E 4 C
	(16) R
	(17) := R 0 C, 16 T

Comme il est possible de prendre plusieurs tranches d'un même tableau, on les numérote (il faudra réserver un vecteur de renseignements différent pour chacun). Le triplet de fin de description de chaque tranche a pour résultat l'adresse de cette tranche (le triplet 12 a pour résultat l'adresse de yy), donc pour décrire l'adresse de la variable indicée yy [4] on fait référence au triplet 12 pour décrire la fonction d'accès.

3.1.3. *Adresse d'une feuille ou d'une sous structure.*

Une sous structure ou une feuille d'une structure peut se caractériser à l'implantation par un déplacement par rapport à l'adresse de l'origine de la structure majeure. En $\&i_1$ on peut décrire le nom d'une feuille ou d'une sous structure par deux renseignements :

- le nom de la structure majeure qui est un descriptif,
- le doublet caractérisant les triplets de début et de fin de génération de la déclaration de la feuille ou de la sous structure.

La syntaxe est donc identique pour une sous structure ou une feuille d'arbre.

<adresse sous structure> = (<descriptif>, (entier,entier))

Exemple

DCL	1 Z	(1) ALLC (np_Z, rg_Z) STRUCTURE (1,7)
	2 U	(2) STRUCTURE (2,4)
	3 A BIN FIXED	(3) (3,3) REAL FIX BIN (15)
	3 B CHARACTER (8)	(4) 4,4) FIX CH 8C
	2 V	(5) STRUCTURE (5,7)
	3 C BIN FIXED	(6) (6,6) REAL FIX BIN (15)
	3 D BIN FIXED	(7) (7,7) REAL FIX BIN (15)
Z.A	= 3	(8) : = 3 C, ((np_Z, rg_Z), (3,3))

3.2. Les références aux valeurs des objets

On peut classer les références aux valeurs des objets en trois catégories

- les références aux valeurs globales des objets quel que soit leur mode, seront exprimées en $\&i_1$ par un certain nombre d'opérations de derepérage appliquées à la désignation de l'objet en $\&i_1$ qui est le descriptif, ou la référence à la désignation d'un paramètre formel.

La syntaxe de ℓi_1 recouvrant ces deux premiers cas est

[{<derepérage>}*]<descriptif> | {<dereperage>}* (entier,entier,entier)

- les références aux valeurs d'éléments d'objets composés. On calcule d'abord le nom par une série de triplets dépendant des constructions qui interviennent dans le mode de l'objet, la valeur de l'élément est le conten de cette adresse (il faut donc faire une opération de dérepérage sur le nom).

La syntaxe correspondante de ℓi_1 est :

*(entier T)

- Les références aux valeurs de constantes décrites explicitement.

La syntaxe est :

<valeur> C

avec

<valeur> = < valeur mode primitif> | <valeur construction>

Nous ne préciserons pas plus ici la description des valeurs de mode primitif, cela est sans intérêt.

Pour les valeurs d'une construction tableau, nous utiliserons les notations des propositions collatérales d'ALGOL 68 par exemple (9,10,11) est un rang de trois nombres entiers.

Pour les valeurs d'une construction ensemble, nous utiliserons la notation entier[{entier...entier}] où le premier nombre entier situé immédiatement après le crochet ouvrant a pour valeur le numéro du code, les entiers situés entre accolades décrivent les valeurs scalaires.

4. INSTRUCTIONS D'AFFECTATION

La traduction en ℓi_1 d'une affectation du langage source ne fera pas jouer le même rôle à la partie gauche qu'à la partie droite. Considérons une instruction source A = B ; cette instruction signifie que la valeur de A est modifiée par la valeur de B, il faut donc traduire la partie gauche d'une affectation par une adresse et la partie droite par une valeur

En ℓi_1 on dira que la référence à l'objet A est une référence à son nom et que la référence à l'objet B est une référence à sa valeur.

La syntaxe d'une instruction d'affectation en ℓi_1 est :

<affectation> = : = <description mode> <reference valeur>, <reference adresse> |
<mode primitif> TAB entier (<ℓentier>) <reference valeur> , (<ℓentier>)
<reference adresse>

<description mode> = <mode primitif> | ENS

Les exemples qui suivent illustrent cette syntaxe (nous rappelons d'abord la traduction des déclarations) :

[10] ent t;	(1) ALLC (np_t,rg_t) TAB 1 (2) DIM E 1 C, 10 C (3) E
rep ent z = t [i] ;	(4) ORIG (np_t, rg_t) TAB 1 (5) IND E *(np_i, rg_i) (6) E (7) ALLC (np_x, rg_x) E (8) : = E 3C, 6T
ent x;	(9) : = E *(np_z, rg_z), (np_x, rg_x)
z : = 3;	(10) : = E *(np_x, rg_x), 6T
x : = z;	(11) ALLC (np_{zz}, rg_{zz}) A
z : = x;	(12) : = A (np_z, rg_z), (np_{zz}, rg_{zz})
rep rep ent zz;	(13) ALLC (np_u, rg_u) TAB 1
zz : = z;	(14) DIM E 1C, *(np_x, rg_x)
[1 : x] ent u;	(15) E
t : = u ;	(16) : = TAB 1 (14) *(np_u, rg_u), (2) *(np_t, rg_t)

14 est la référence au triplet qui décrit la dimension de u,

2 est la référence au triplet qui décrit la dimension de t.

Ces informations permettront de générer le code de comparaison des bornes à l'exécution.

5. OPERATIONS SUR LES DONNEES

5.1. Les opérations associées aux modes primitifs.

A chaque mode primitif d'un langage algorithmique sont associés des opérateurs binaires ou unaires.

Aux modes arithmétiques sont associés les opérateurs binaires, d'addition, de soustraction, les opérateurs de relations, les opérateurs unaires (moins, conversions).

Au mode logique on peut associer les opérateurs binaires intersection, uni l'opérateur unaire négation... . Au mode primitif chaîne on associera naturelle l'opérateur de concaténation.

Les modes primitifs sont un paramètre de langage ℓi_1 , les opérateurs associés à ces modes primitifs sont donc des paramètres du langage ℓi_1. Dans la traduction d'un langage source particulier en ℓi_1 on ne pourra atteindre que quelques unes des alternatives de la règle de définition des opérateurs, celles qui correspondent à ce langage source spécifique. Les opérations portent sur les valeurs des objets qui sont soit des objets de mode primitif, soit des parties d'objets de mode primitif. La syntaxe des opérateurs associés aux modes primitifs peut se formaliser de la manière suivante :

<opération simple> = <binaire simple> <mode primitif> <opérande>, <opérande> |
<unaire simple> <mode primitif> <opérande>

<opérande> = <référence valeur>

<binaire simple> = + | - | × | / | < | ⩽ | > | ⩾ | EG | ‖ | ET | OU

<unaire simple = - | NON

Exemples

a ⩾ b	⩾ E *(np$_a$, rg$_a$), *(np$_b$, rg$_b$)
i + j	+ E *(np$_i$, rg$_i$), *(np$_j$, rg$_j$)
t(i,j) + k	(1) ORIG (np$_t$, rg$_t$) TAB 2 (2) IND E * (np$_i$, rg$_i$) (3) IND E * (np$_j$, rg$_j$) (4) E (5) + E *(4T), *(np$_k$, rg$_k$)

5.2. Les opérations associées aux constructions

5.2.1. *Les opérations associées à la construction tableau.*

En PL I il est possible d'effectuer des opérations infixées entre deux tableaux de même mode ayant même dimension et dont les bornes sont identiques. Dans notre système la vérification des modes et des dimensions se fait pendans la première étape de traduction ainsi que la vérification des bornes si elles sont constantes.

Si les bornes sont dynamiques la sémantique du code opérateur en ℓi_1 , comportera la génération de code permettant de tester l'égalité de ces bornes à l'exécution.

Exemple

DCL A(I,J) BIN FIXED;	(1) ALLC (np$_a$, rg$_a$) TAB 2 (2) DIM E I C, *(np$_i$, rg$_i$)
DCL B(I,K) BIN FIXED;	(3) DIM E I C *(np$_j$, rg$_j$)
A = A + B;	(4) E; (5) ALLC (np$_b$, rg$_b$) TAB 2 (6) DIM E I C, *(np$_i$, rg$_i$) (7) DIM E I C, *(np$_j$, rg$_j$) (8) E; ---- (n) + E TAB 2 (2,3) *(np$_a$, rg$_a$), (6,7)* (np$_b$,rg (n+1):=E TAB 2 (2,3) (n T), (2,3) (np$_a$, rg$_a$)

Les bornes des tableaux A et B sont dynamiques, lorsqu'on génère en ℓi_1 le triplet d'addition entre les deux tableaux, il faut mémoriser les informations sur les bornes qui permettront de générer à l'étape suivante les instructions de comparaison entre les bornes. C'est pourquoi nous rappelons les numéros de triplets de description de chaque dimension.

La syntaxe ℓi_1 des opérations globales entre deux constructions tableaux peut donc s'écrire :

<opération tableau> = <opération tableau binaire> | <opération tableau unaire>

<opération tableau binaire> =

<binaire simple> <mode primitif> TAB entier (<&entier>) <reference valeur>, (<&entier>) <reference valeur>

<opération tableau unaire> =

<unaire simple> <mode primitif> TAB entier (<&entier>) <reference valeur>

<&entier> = entier {,entier}*

5.2.2. *Les opérations associées à la construction ensemble.*

Seul le langage PASCAL autorise la construction ensemble. Les opérateurs associés à ces objets sont : l'union, l'intersection, la différence au sens des ensembles et les opérateurs de relations (tests d'égalité ou d'inégalité, test d'inclusion).

<opération ensemble> = <operateur d'ensemble> ENS <reference valeur>,
<reference valeur>

<opérateur d'ensemble> = + | × | - | = | ≠ | < = | > =

Les opérateurs ainsi décrits conformément à la terminologie de Wirth sont l'union, l'intersection, la différence, les tests d'égalité, d'inégalité, d'inclusion.

En PASCAL on peut tester l'appartenance d'une valeur de type scalaire à un ensemble, cet opérateur agit donc sur une valeur de mode primitif et une construction ; il faut donc définir en $\&i_1$ une syntaxe particulière pour traduire la relation d'appartenance :

<opération appartenance> =

DANS CODE entier *<descriptif>, ENS *<descriptif>

on recherche si la valeur de la variable de type scalaire est une des valeurs composant l'ensemble.

Exemple de traduction de quelques instructions de PASCAL :

```
type jour = (lundi, mardi, mercredi, jeudi, vendredi, samedi, dimanche)
type semaine = set of jour
Var s, travail, libre : semaine
      j : jour

begin travail : = [ ];
      libre   : = [ ];

 s : =[lundi,mardi,...dimanche];
 j : = samedi ;
libre : = [j] + libre +[dimanche]
```

```
(1) ALLC (np_s, rg_s) ENS
(2) CODE I (1...7)
(3) ALLC (np_travail, rg_travail) ENS
(4) CODE 1 (1...7)
(5) ALLC (np_libre, rg_libre) ENS
(6) CODE I (1...7)
(7) ALLC (np_j, rg_j) CODE 1 (1...7)
(8) := ENS C [ ] , (np_travail, rg_travail)
(9) := ENS C [ ] , (np_libre, rg_libre)
(10) := ENS C [1{1...7}] , (np_s, rg_s)
(11) := CODE 1 C {6} , (np_j, rg_j)
(12) + ENS C [*(np_j, rg_j)] , *(np_libre,rg_libr
(13) + ENS (12 T), C [1{7}]
(14) = ENS (13 T), (np_libre, rg_libre)
```


On peut faire quelques remarques à propos de la traduction de la dernière instruction libre : = [j] + libre + [dimanche]

J a été déclarée comme variable de type scalaire, en plaçant j entre crochets on créé un ensemble dont la portée est l'instruction, en $\&i_1$ la valeur de cet ensemble est représentée par

C [*(np_j, rg_j)] constante de mode ensemble qui comporte un seul élément la valeur de j *(np_j, rg_j). De même, dimanche est une valeur scalaire dont on fait un ensemble en le plaçant entre crochets, en $\&i_1$ on représente cette constante de mode par [1{7}] où 1 est le codage du type scalaire et 7 la valeur appropriée.

6. LES TEXTES DE ROUTINE

Rappelons la syntaxe LI_1 de la déclaration d'un texte de routine.

<texte de routine> = ROUTINE <descriptif> entier ; <&trip> ; END;|
ROUTINE * (entier,entier,entier) entier ; <&trip> ; END

Il est inutile de déclarer les types des paramètres formels, mais dans l
références aux paramètres formels dans les instructions, il faut évidemment
indiquer. Il faut également désigner en $\&i_1$ les paramètres formels comme on
désigné les variables locales, un moyen simple de coder cette désignation es
concaténer à la désignation du texte de routine, un nombre entier qui est le
rang du paramètre formel. La désignation d'un paramètre formel est donc un
triplet de nombres entiers.
Au résultat d'une procédure on attribuera par convention le rang 0. Le résul
d'une procédure doit avoir une mémoire qui lui est réservée, on fera donc un
déclaration de variable pour le résultat.

6.1. La traduction des procédures d'ALGOL 68

proc (ent,ent) ent c : (ent a, ent b) = ax(a+b)

```
proc (ent,ent) ent d ;                (1) ROUTINE (np_c, rg_c) 2
d := c ;                              (2) ALLC (np_c, rg_c, 0) E
                                      (3) + E  (np_c, rg_c, 1)  (np_c, rg_c, 2)
                                      (4) x E  (np_c, rg_c, 1), 3T
                                      (5) := E 4T, (np_c, rg_c, 0)
                                      (6) END
                                      (7) ALLC (np_d, rg_d) Et
                                      (8) := Et 1T, (np_d, rg_d)
```

Pour traduire la déclaration de variable procédure d, on génère le triplet (7
qui est une réservation d'étiquette. En effet, l'adresse d'un texte de rou
est une adresse dans le code, et il faut distinguer l'adresse dans le code d
l'adresse en mémoire pour l'étape suivante de traduction.
L'affectation d'un texte de routine à cette variable peut donc s'exprimer en
$\&i_1$ par une affectation d'étiquette, la valeur de l'étiquette du texte de ro
est le triplet (1).
En ALGOL 68 le mécanisme de passage des arguments s'exprime par définition p
une identité entre le paramètre formel et le paramètre effectif. Dans l'exem

qui précède le mécanisme de passage des paramètres peut s'écrire :

```
ent a = 3 ;
ent b = 2 ;
```

a et b étant des paramètres formels, on ne peut évidemment pas effectuer la propagation de constantes dans le code du texte de la routine, puisque le paramètre effectif varie avec chaque appel. Il faut donc traiter le paramètre effectif comme une valeur calculée et remplir une zone de mémoire attribuée au paramètre formel avec la valeur calculée du paramètre effectif. Ceci sera réalisé par une opération d'affectation.

```
d (3,2) ;                             APPEL * (np_d, rg_d) 2
                                      := E 3C, (*(np_d,rg_d), 1)
                                      := E 2C, (*(np_d,rg_d), 2)
                                      FINAPPEL
```

un triplet de code opération . APPEL désigne la procédure appelée, il comporte également le nombre d'arguments. Pour chaque argument on exprime l'équivalence paramètre formel paramètre effectif par une affectation.
La syntaxe d'un appel de procédure en $\&i_1$ est :

APPEL <descriptif> entier ; [<&equi>] ; FINAPPEL
<&equi> = {<affectation>}*

Dans l'exemple précédent le passage des arguments se traduit par une affectation des valeurs. Dans l'exemple qui suit le passage des arguments se traduit par une affectation de noms.

```
proc kwic (rep chaîne r) neutre :     (1) ROUTINE (np_kwic, rg_kwic) 1
début ... fin ;                       (2)
chaîne s := "algol" ;
                                      (n) END
                                      (n+1) APPEL (np_kwic, rg_kwic) 1
kwic (s) ;                            (n+2) := A (np_s, rg_s), (np_kwic, rg_kwic, 1)
```

6.2. La traduction des procédures d'ALGOL 60

En ALGOL 60 les modes de passage sont par valeur ou par nom.

6.2.1. *La traduction du passage par valeur.*

Si un paramètre formel est déclaré par valeur, une zone de mémoire lui est effectivement réservée, cette zone est initialisée avec la valeur de l'argument. La valeur du paramètre formel peut être modifiée en cours d'exécution du corps de la procédure, le paramètre formel est une variable au sens d'ALGOL 68. Dans le passage par valeur la procédure ne peut en aucune façon

modifier la valeur de l'argument. En ℓi_1 on réserve une zone de mémoire au paramètre passé par valeur en générant un triplet ALLC. Le passage des paramè s'exprime en ℓi_1 par l'affectation des valeurs des arguments aux paramètres.

```
procedure h (x,y) ; value x,y ;
integer x,y ;
begin
  x := x + 5
  y := y - 1 .
end ;
h(0,1) ;
```

```
(1) ROUTINE (np_h, rg_h) 2
(2) ALLC (np_x, rg_x) E
(3) := E (np_h, rg_h, 1), (np_x, rg_x)
(4) ALLC (np_y, rg_y) E
(5) := E (np_h, rg_h, 2), (np_y, rg_y)
(6) + E 5C, (np_x, rg_x)
(7) := E 6T, (np_x, rg_x)
(8) - E (np_y, rg_y), 1C
(9) := E 8T, (np_y, rg_y)
(10) END
(11) APPEL (np_h, rg_h) 2
(12) := E 0C, (np_h, rg_h, 1)
(13) := E 1C, (np_h, rg_h, 2)
(14) FINAPPEL
```

6.2.2. *La traduction du passage par nom.*

Le mécanisme du passage par nom d'ALGOL 60 peut se traduire par la génération d'une procédure qui calcule l'adresse de l'argument. A chaque référence au paramètre formel dans le corps de la procédure, on génère un appel à ce sous-programme.

```
procédure r (x,i) ;
x : = 2,
i : = 5 ;
end ;
r(b [j×2],j)
```

```
(1) ROUTINE (np_r, rg_r) 2
(2) APPEL *(np_r, rg_r,1) 0
(3) FINAPPEL
(4) : = E 2C,  (* (np_r, rg_r, 1),0)
(5) APPEL *(np_r, rg_r,2) 0
(6) FINAPPEL
(7) : = E 5C,  (* (np_r, rg_r,2),0)
(8) END
(9) ROUTINE (np_ax,rg_ax) 0
(10) ALLC (np_ax,rg_ax,0) A
(11) ×E 2C , *(np_j,rg_j)
(12) ORIG (np_b,rg_b) TAB 1
(13) IND E 11T
(14) E
(15) :=A 14T , (np_ax,rg_ax,0)
(16) END
```



```
(17) ROUTINE (np_ai,rg_ai) 0
(18) ALLC (np_ai,rg_ai,0) A
(19) :=A (np_j,rg_j) , (np_ai,rg_ai,0)
(20) END
(21) APPEL (np_r,rg_r) 2
(22) :=ET 9T , (np_r,rg_r,1)
(23) :=ET 17T , (np_r,rg_r,2)
(24) FINAPPEL
```

Avant l'appel de la procédure r, on génère les textes de routine de calcul de l'adresse des arguments.

Le texte de routine (np_{ax}, rg_{ax}) calcule l'adresse de b [j×2], le texte de routine (np_{ai}, rg_{ai}) calcule l'adresse de j.

A chaque référence au paramètre formel x dans la procédure, on génère un appel au texte de routine (np_{ax}, rg_{ax}) pour transmettre l'adresse du dernier texte de routine, on considère que le paramètre formel x est de mode adresse, donc le passage de paramètre de r se traduit par une identité des adresses du paramètre formel x et de la routine (np_{ax}, rg_{ax}). Soit l'identité suivante

= A (np_{ax}, rg_{ax}), (np_r, rg_r, 1)

Pour traduire l'instruction ALGOL x : = 2, on génère d'abord un appel à la routine de calcul de l'adresse de x: cette routine a pour adresse (np_r, rg_r, 1).

6.2.3. *La désignation des paramètres formels*

Dans ce qui précède, nous avons désigné un paramètre formel par un triplet d'entiers constitué par le descriptif de la procédure auquel se rapporte ce paramètre formel suivi du rang de ce paramètre formel. On aurait pu se contenter du rang puisque dans les références aux paramètres formels dans le corps de la procédure, il n'y a pas d'ambiguité et dans l'appel non plus. Cependant dans le cas où le paramètre effectif est le paramètre formel d'une autre procédure, on ne peut pas se contenter du rang pour décrire le paramètre formel d'une autre procédure.

Exemple

```
begin
procedure  p(u); integer u;
value u;
   procédure g(y); integer y;
      value y,

   g(u);
```

```
ROUTINE (np_p, rg_p) 1
ALLC (np_p, rg_p, 1) E
  ROUTINE (np_g, rg_g) 1
  ALLC (np_g, rg_g, 1) E
APPEL (np_g, rg_g) 1
  = E*(np_p, rg_p, 1), (np_g, rg_g, 1)
FINAPPEL
```

6.2.4. *La génération de la vérification des types.*

Lorsque le paramètre formel est une procédure spécifiée par nom, il n'est pas possible de vérifier statiquement la concordance des types. Il faut générer en ℓi_1 une routine de vérification des types. Montrons comment cela peut être réalisé sur un exemple.

```
begin integer x; real y;
  procedure h(a,b): integer a;
    real b;                                  A  ROUTINE (np_h, rg_h) 2
                                                END

  procedure p (g); procédure g                  ROUTINE (np_p, rg_p) 1
    g(x,y)                                   B  APPEL (np_verif, rg_verif)
                                                = chaîne 'E'C, *(np_verif, rg_verif, 1)
                                                = chaîne 'R'C, *(np_verif, rg_verif, 1)
                                                FINAPPEL
                                             C  APPEL  (np_p, rg_p, 1) 2
                                                = E* (np_x, rg_x), (*(np_p, rg_p, 1) 1
                                                = E* (np_y, rg_y),  (*(np_p, rg_p, 1),2)
                                                FINAPPEL
                                                END
p(h)                                         D  ROUTINE (np_verif, rg_verif) 2
                                                *Comparer la valeur du 1er paramètre
                                                 formel et la chaîne constante 'E'
                                                 type du 1er paramètre formel de h
                                                *Comparer la valeur du 2ème paramètre
                                                 formel et la chaîne constante 'R' type
                                                 du 2ème paramètre formel de h
                                                 END
                                             E   APPEL (np_p, rg_p) 1
                                                 :=ET 1T , (np_p,rg_p,1)
                                                 FINAPPEL
```

Lors de l'appel de p ; le traducteur connaît le nom h, donc les types des paramètres formels de h ; entier, réel.

On peut donc définir à ce niveau le texte d'une routine désigné en ℓi_1 par (np_{verif}, rg_{verif}) à deux paramètres formels de type chaîne. Dans le corps de cette procédure on compare la valeur du type du 1er paramètre formel de h (la constante chaîne 'E') à la valeur du paramètre formel ; on effectue le même test pour la valeur du 2ème paramètre formel.

Puis on génère en ℓi_1 l'appel de la procédure p proprement dite dans la séquence de code E, le paramètre effectif A étant une procédure, le passage de paramètre se traduit par une affectation d'adresse dans le code, c'est-à-dire d'étiquette or la valeur de l'étiquette de la routine h est le triplet numéro 1 :

:=ET 1T , $(np_p, rg_p, 1)$

Pour traduire l'appel g(x,y) il faut d'abord vérifier que les types de x et y sont corrects, pour cela on appelle en ℓi_1 la procédure (np_{verif}, rg_{verif}) avec pour arguments les chaînes de caractère codant les types de x et y. Ceci est constitué par la séquence de code ℓi_1 B. Puis on appelle la procédure dont l'adresse est la valeur du 1er paramètre formel de p sa désignation est $*(np_p, rg_p, 1)$ et on traduit le passage des arguments x et y. Ceci est constitué en LI_1 par la séquence de code C.

6.3. La traduction des procédures PL 1.

En PL 1 les paramètres sont passés par référence ; c'est-à-dire qu'on communique à la procédure qui va s'exécuter les adresses des arguments. Le passage des paramètres s'exprime donc en ℓi_1 par des affectations de noms.

```
dcl i ;                 (1) ALLC (np_i,rg_i) REAL FIX BIN (15)
dcl tab(20) ;           (2) ALLC (np_tab,rg_tab) TAB 1
f:proc(j) ;             (3) DIM E 1C, 20C
dcl j ;                 (4) REAL FIX BIN (15)
dcl k ;                 (5) ROUTINE (np_f,rg_f) 1
j=k ;                   (6) ALLC (np_k,rg_k) REAL FIX BIN (15)
end ;                   (7) := REAL FIX BIN (15) *(np_k,rg_k) , (np_f,rg_f,1)
call f(tab(i)) ;        (8) END
                        (9) ORIG (np_f,rg_f) TAB 1
                       (10) IND E *(np_i,rg_i)
                       (11) REAL FIX BIN (15)
                       (12) APPEL (np_f,rg_f) 1
                       (13) := A 11T , (np_f,rg_f, 1)
                       (14) FINAPPEL
```

7. LES STRUCTURES DE CONTROLE DE LI 1

En résumé on peut dire que les structures de contrôle des langages algorithmiques sont constituées par les instructions de branchement inconditionnelles, les instructions de branchement conditionnelles, les boucles.

7.1. Les instructions de branchement inconditionnelles.

En langage source, elles se présentent sous la forme d'une instruction de branchement à une constante étiquette alphanumérique, ou une variable indicée de mode primitif étiquette (cas des tableaux d'étiquettes de PL 1). On peut traiter un aiguillage d'ALGOL 60 comme un tableau à une dimension d'étiquettes.

7.1.1. *La traduction des branchements aux constantes étiquettes.*

Au cours de la traduction du langage source en langage ℓi_1 , il est possible d'associer à chaque définition de constante étiquette, le numéro du triplet où commence la traduction de l'instruction étiquetée. Dans une instruction de branchement, on remplace la référence à une constante étiquette par le numéro du triplet où commence la traduction de l'instruction étiquetée.

Dans un langage à structure de blocs, il est possible qu'à l'exécution cette instruction de branchement provoque un changement de niveau de procédure. Comme le traducteur peut mémoriser le niveau de procédure de la définition de l'étiquette, il est possible de reporter ce niveau dans l'instruction de branchement. Cette information permettra dans la suite du processus de traduction de restaurer les registres locaux comme pour une instruction de sortie de procédure.

La syntaxe d'une instruction de branchement à une constante étiquette est donc en ℓi_1

ALLERA (entier) entier T

Exemple

goto toto;	(n) ALLERA (np_{toto}) n+3 T
	(n+1) + E !C, *(np_j,rj_j)
i : = j + 1;	(n+2) : = E n+1 T, (np_i,rj_i)
toto : ℓ : = i ;	(n+3) : = E *(np_i rg_i), (np_ℓ,rg_ℓ)

7.1.2. *La traduction des branchements aux étiquettes d'un tableau.*

En PL 1 il est possible de déclarer un tableau constant d'étiquettes de la façon suivante :


```
     DCL Z(3) LABEL;
Z(1) : ———
       ———
Z(2) : ———
Z(3) : ———
     goto Z(i);
```

A la traduction, il est possible d'associer à chaque variable indicée Z(1), Z(2), Z(3), le numéro du triplet où commence la traduction des trois instructions étiquetées. On va donc traiter en ℓi_1 le tableau Z comme un tableau constant d'étiquettes dont les valeurs sont les adresses dans le code ℓi_1 c'est-à-dire des numéros de triplets. Une instruction de branchement goto Z(i) se traduit par un branchement à une adresse dans le code qui est le contenu de l'adresse Z(i)

dcl Z(3) label;	(1) ALLC (np_Z,rg_Z) TAB 1
	(2) DIM E !C, 3C
	(3) Et
	(4) := Et TAB 1 (2) (n_1, n_2)T, (2) (np_{sw},rg_{sw}
Z(1) : ———	(n_1) ———
Z(2) : ———	(n_2) ———
Z(3) : ———	(n_3) ———
	(n) ORIG (np_Z,rg_Z) TAB 1
goto Z(i)	(n+1) IND E * (np_i,rg_i)
	(n+2) Et
	(n+3) ALLERA * (n+2 T)

Ici il est inutile de préciser le niveau de procédure, puisqu'on reste toujours dans le niveau en cours.

On peut traiter de façon absolument identique un aiguillage d'ALGOL 60.

begin aiguillage sw : E,F,	(1) ALLC (np_{sw},rg_{sw}) TAB 1
	(2) DIM E !C, 2C
	(3) Et
	(4) := Et TAB 1 (2) (n_1, n_2)T, (2) (np_{sw},rg_{sw}
E : ———	(n_1) ———
F : ———	(n_2) ———
goto sw [i+j]	(n) + E* (np_i,rg_i), (np_j,rg_j)
end	(n+1) ORIG (np_{sw},rg_{sw}) TAB 1
	(n+2) IND E * (n T)
	(n+3) Et
	(n+4) ALLERA* (n+3 R)

7.1.3. *La traduction des branchements aux variables étiquettes.*

En PL I on peut déclarer une variable de type LABEL. Elle peut prendre comme valeurs des constantes étiquettes d'instructions, par l'intermédiaire d'instructions d'affectation.

Une instruction de branchement peut être traduite par une instruction de branchement à la valeur d'une variable de type étiquette.

Mais il se peut qu'en certains points du programme une référence à cette valeur soit illégale.

Exemple :

```
begin dcl x label;
x = b;
goto x;
b : ———
    begin
c : ———
    x = c;
    end;
goto x;      |*la référence à x est incorrecte ici car x a pour valeur c
end;           qui est inconnu en ce point du programme*|
```

En général le contrôle de la validité du branchement n'est possible qu'à l'exécution. Il faut donc conserver les informations qui permettent d'effectuer ce contrôle à l'exécution, dans la syntaxe de ℓi_1 . Le niveau d'imbrication des blocs n'est pas une information suffisante car il ne permet pas de lever l'ambiguité entre deux blocs parallèles.

A la traduction entre PL I et ℓi_1 on pourra associer deux numéros :

- un numéro d'ordre,
- et le niveau d'imbrication du bloc.

L'exemple suivant montre comment on peut appliquer cette numérotation.

```
h : proc ;
  ⌈begin                    (1,1)
  | ⌈begin                  (2,2)
  | | ⌈begin                (3,3)
  | | ⌊end
  | | ⌈begin                (4,3)
  | | ⌊end
  | ⌊end
  | ⌈begin                  (5,2)
  | | ⌈begin                (6,3)
  | | | ⌈begin              (7,4)
  | | | |
  | | |
  ⌊end
```


En ℓi_1 on assurera à chaque constante étiquette le niveau de procédure, le numéro d'ordre du bloc, le niveau d'imbrication du bloc où elle est définie. Dans une instruction de branchement à une variable label on indique le niveau de procédure en cours, le numéro d'ordre du bloc, le niveau d'imbrication du bloc en cours.

```
begin dcl  x label;          (1) ALLC (np_x,rg_x) Et
      x = b                  (2) : = Et (np,(1,1))n_1 T, (np_x,rg_x)
      goto x;                (3) ALLERA (np,(1,1))    *(np_x,rg_x)
b : ———                      (n_1) ———
      begin
c : ———                      (n_2) ———
      x = c                  (n) : = Et (np,(2,2))n_2T, (np_x,rg_x)
      end
      goto x;                (m) ALLERA (np,(1,1)) *(np_x rg_,x)
```

7.1.4. *La syntaxe* LI_1 *des instructions de branchements :*

```
<branch>  = ALLERA entier T | ALLERA *(entier T) |
            ALLERA (entier,(entier,entier)) *<descriptif>
```

7.2. Les instructions de branchement conditionnelles.

En langage source, elles comportent toutes un test de la valeur d'une expression. Le résultat de ce test peut posséder une valeur binaire (cas des instructions conditionnelles d'ALGOL 60, PL/1) ou ternaire (cas des "IF" arithmétiques de FORTRAN) ou n-aire (CASE de PASCAL, GOTO calculé de FORTRAN).

Dans tous les cas on peut décomposer le test en une suite de tests ayant chacun pour résultat une valeur binaire. Nous adoptons donc en ℓi_1 la forme syntaxique correspondant au test d'une expression logique. Comme on peut toujours générer le code de la clause "alors" à la suite du triplet de test, le triplet de test fait référence seulement au début de la clause "sinon".

SINON entier T

Exemple 1 : traduction du IF arithmétique de FORTRAN.

```
  IF (a+b) 1,2,3      (1) + a,b
1 ———                 (2)´<É IT, OC
                      (3) SINON n T
                                 ⎧traduction de la suite
2 ———          (a+b) <0        ⎨d'instructions commençant
                                 ⎩par l'étiquette 1
```

```
                    ALLERA n" T
3 ———————           (n) EG E 1T, 0C
                    (n_1) SINON n' + 1 T
                          { traduction de la suite d'instructions
                    (a+b) = 0 { commençant par l'étiquette 2
                       (n') ALLERA n" T
                       (n'-1 { traduction de la suite d'instructions
                             { commençant par l'étiquette 3
                    (a+b) >0
```

Exemple 2 : instruction conditionnelle d'ALGOL 60.

```
IF i < j + 1 THEN IF i < j-2 THEN L := 2 ELSE L := 1 ELSE L := 0
   (1) + E *(np_i,rg_i), *(np_j,rg_j)
   (2) < E *(np_i,rg_i), (1 T)
   (3) SINON 11 T
   (4) - E *(np_j,rg_j), 2C
   (5) < E *(np_i,rg_i), (4T)
   (6) SINON 9 T
   (7) : = E 2C, (np_ℓ,rg_ℓ)
   (8) ALLERA 12 T
   (9) : = E 1C, (np_ℓ,rg_ℓ)
  (10) ALLERA 12 T
  (11) : = E 0C, (np_ℓ,rg_ℓ)
```

7.3. Les boucles.

Dans le langage source, elles possèdent la même forme syntaxique externe : après un code opération indiquant qu'il s'agit d'une boucle (DO ou FOR), on définit une variable de contrôle agissant sur les instructions de la boucle.

Or l'interprétation d'une instruction de boucle diffère suivant les langages : en PL 1 les valeurs limites de la variable de contrôle sont calculées une fois pour toutes (la variable de contrôle, et les variables intervenant dans les expressions limites peuvent cependant être modifiées en cours d'exécution dans la boucle) ; en ALGOL 60 la valeur limite supérieure de la variable de contrôle et le pas sont recalculés à chaque itération.

C'est pourquoi une expansion complète de la boucle combinant le test et les instructions de saut est la seule traduction compatible avec les différentes sémantiques des langages considérés.

Exemple 1 : Traduction d'une boucle d'ALGOL 60.

```
for i : = a + b step 2 until a + 2 × b   (1) + E *(np_a,rg_a, (np_b,rg_b)
do ...                                   (2) : = E 1T, *(np_i,rg_i)
begin                                    (3) × E 2C, *(np_n,rg_b)
a : = a + 1                              (4) + E (np_a,rg_a), 3T
end                                      (5) ≤ E (np_i,rg_i), 4T
                                         (6) SINON n+5 T
                                                 ⋮
                                         (n) + E 1C, *(np_a,rg_a)
                                       (n+1) : = E n T, (np_a,rg_a)
                                       (n+2) + E (np_i,rg_i), 2C
                                       (n+3) : = E n+2 T, (np_i,rg_i)
                                       (n+4) ALLERA 3T
```

Exemple 2 : Traduction d'une boucle PL 1

```
do i = 2 to a × b;                       (1) : = 2 2C, *(np_i,rg_i)
                                         (2) × E *(np_a,rg_a), *(np_b,rg_b)
   ⋮                                     (3) < E *(np_i,rg_i), 2T
                                         (4) SINON n+3 T
end;                                             ⋮
                                         (n) + E 1C, *(np_i,rg_i)
                                       (n+1) : = E nt, *(np_i,rg_i)
                                       (n+2) ALLERA 3T
```

Pour les boucles PL 1, puisque les valeurs limites de la variable de contrôle sont calculées une fois pour toutes, on doit effectuer l'expansion de la boucle en générant d'abord le test sur les variables de contrôle. Certaines boucles possèdent une clause supplémentaire, la clause tant que (boucles FOR WHILE d'ALGOL 60, DO WHILE de PL 1, REPEAT WHILE de FORTRAN). On doit donc faire l'expansion de la boucle en introduisant un test supplémentaire.

Exemple de traduction d'une boucle DO WHILE de PL 1

```
Do i = 1 to 10 while (A = B);            (1) : = E 1C, *(np_i,rg_i)
                                         (2) < E *(np_i,rg_i), 10 C
   ⋮                                     (3) SINON n+3 T
```

```
end;                                (4) EG E  *(np_a,rg_a),  *(np_b,rg_b)
                                    (5) SINON n+3 T
                                          ⋮
                                    (n) + E 1C, *(np_i,rg_i)
                                  (n+1) = E n T, *(np_i,rg_i)
                                  (n+2) ALLERA 2T
                                  (n+3)
```

7.4. La syntaxe ℓi_1 des instructions de contrôle.

<contrôle> = <branch> | SINON entier T

<branch> = ALLERA entier T | ALLERA *(entier T) |
ALLERA (entier,(entier,entier))*<descriptif>

8. LA SYNTAXE DE LI 1

<prog> = <ℓcode>

<ℓcode> = <ℓcode> <code> | <code>

<code> = <texte de routine> | <ℓtrip> | <call>

<ℓtrip> = <ℓtrip> <trip> | <trip>

<texte de routine> = ROUTINE <descriptif> entier; <ℓtrip>; END |
ROUTINE *(entier,entier,entier) entier; <ℓtrip>; END

<trip> = <declaration> | <affectation> | <opération> | <contrôle>

<declaration>= ALLC <descriptif> <déclareur effectif> |
ALLC (entier,entier,entier) <declareur effectif>

<declareur effectif> = <mode primitif> | <declareur effectif de tableau> |
<declareur effectif de structure> | <declareur d'ensemble>

<declareur effectif de tableau> =
TAB [COMPRESSE] entier; <dim> ; [{<dim>;}*] <déclareur effectif>

<dim> = DIM E <borne>, <borne>

<borne> = <constante entière> | <ℓdereperage> <descriptif> |*(entier T) | Id

<ℓdereperage> = <derep> { <derep>}*

<derep> = *

<declareur effectif de structure> =
STRUCTURE [COMPRESSE] (entier,entier) <champ effectif>; { <champ effectif>}*

<champ effectif> = (entier,entier) <declareur effectif>

<declareur d'ensemble> = ENS <declareur effectif>

<call> = APPEL <desproc> entier ; [{<affectation>}*] ; FINAPPEL

<desproc> = <descriptif> | *<descriptif> | *(entier,entier,entier)

<affectation> = : = <description mode> <reference valeur>, <reference adresse> |

<mode primitif> TAB entier (<ℓentier>) <reference valeur>,(<ℓentier>)
<reference adresse>)

<description mode> = <mode primitif> | ENS

<ℓentier> = entier { ,entier}*

<reference adresse> = <reference adresse objet> | <reference adresse element>

<reference adresse objet> = <descriptif> | (entier,entier,entier)

<reference adresse element> = <adresse variable indicee> | <adresse tranche>
<adresse sous-structure> | <adresse feuille>

<adresse variable indicee> =
ORIG <origine> TAB entier; <indice> { <indice>}* <déclareur effectif>

<origine> = <descriptif> | entier T

<indice> = IND E <valeur entière>

<valeur entière> = <constante entière> |*<descriptif> |** <descriptif> | entier

<adresse tranche> =
TRANCHE entier ORIG <origine> TAB entier; <rang>;{ <rang> ;}* <declareur effecti
<rang> = <gamme> | <indice>
<gamme> = IND E <domaine>
<domaine> = ? | <valeur entière>, <valeur entière>
<adresse sous structure> = (<descriptif>, (entier,entier))
<adresse feuille> = (<descriptif>, (entier,entier))
<reference valeur> = [{ <dereperage>}*] <descriptif> | { <dereperage>}*
(entier,entier,entier) |
*(entier T) | <valeur> C | entier T
<operation> = <operation simple> | <operation tableau> | <operation ensemble>
<operation appartenance>
<opération simple> = <binaire simple> <mode primitif> <operande> , <operande
<unaire simple> <mode primitif> <operande>
<operande> = <reference valeur>
<binaire simple> = + | - | × | / | < | ⩽ | > | ⩾ | EG | ≠ | ET | OU
<unaire simple> = - | NON
<opération tableau> = <opération tableau binaire> | <operation tableau unaire>
<operation tableau binaire> =
<unaire simple> <mode primitif> TAB entier (<entier>) <reference valeur>
<operation ensemble> = <operateur d'ensemble> ENS <reference valeur> , <refere
valeur>
<operateur d'ensemble> = + | × | - | EG | ≠ | < = | > =
<contrôle> = <branch> | SINON entier T
<branch> = ALLERA entier T | ALLERA *(entier T) |
ALLERA (entier,(entier,entier))* <descriptif>

Deuxième Partie

LA PREMIERE ETAPE DE TRADUCTION

1. LA PREMIERE TRADUCTION

Rappelons que notre système d'aide à l'écriture de traducteurs comporte la définition syntaxique des langages ℓi_1 et ℓi_2, ainsi que les modules de traduction de ℓi_1 vers les deux langages ℓi_2.

L'auteur d'un compilateur doit écrire lui-même les traductions du langage source en langage ℓi_1 et du langage ℓi_2 et langage objet. La façon d'écrire cette traduction dépend des outils dont il dispose. A l'IRIA nous disposons du système de métacompilation DELTA. A titre d'exemple nous avons écrit complètement la traduction d'ALGOL 60 en ℓi_1. Avant de décrire les actions sémantiques nécessaires à la réalisation de cette traduction, nous allons rappeler brièvement comment fonctionne le système DELTA. On peut distinguer trois phases de traitement dans DELTA.

- la construction syntaxique qui bâtit un analyseur syntaxique à partir de la définition syntaxique du langage. Une grammaire hors contexte* représentée sous forme de règle BNF constitue cette définition syntaxique. A chaque règle syntaxique sont associés un ou plusieurs modèles sémantiques. Un modèle décrit les attributs mis en oeuvre ainsi que le calcul de l'attribut sujet du modèle.
- l'analyse syntaxique d'un programme construit l'arbre décoré de ce programme c'est-a-dire que les noeuds de l'arbre syntaxique sont numérotés de manière injective de telle sorte que l'on puisse distinguer les occurrences des attributs aux noeuds de l'arbre, comme des couples (a,x) où a est l'attribut de l'étiquette du noeud numéroté x.

* Une grammaire hors contexte est définie par G = {N, T, Z, P}
où N est l'ensemble fini des symboles non terminaux
T est l'ensemble fini des sybmoles terminaux
Z ∈ N est l'axiome de la grammaire
P est l'ensemble fini des règles de production.

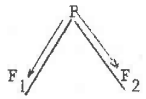
A l'arbre syntaxique ainsi décoré, on associe les systèmes effectifs des équations ayant pour inconnues les occurrences des attributs à ces noeuds.

- la résolution de ce système d'équations constitue l'évaluation sémantique et produit un programme exécutable écrit dans le langage de définition d'attributs.

Le système DELTA permet de traiter des attributs :

- hérités : un attribut hérité d'un noeud ne dépend que du prédécesseur immédiat. L'information part de la racine pour se propager vers les feuilles.
- synthétisés : un attribut synthétisé d'un noeud ne dépend que des successeurs immédiats. L'information remonte dans l'arbre syntaxique, elle part des feuilles et se propage vers la racine.
- mixtes : un attribut mixte en en fait l'association d'un attribut hérité et d'un attribut synthétisé qui se complètent pour le traitement d'une information. Soit l'attribut mixte AD, lorsqu'on aura besoin d'hériter de l'information, on utilisera l'attribut hérité H.AD ; lorsqu'on aura besoin de synthétiser de l'information, on utilisera l'attribut synthétisé S.AD.

Le cheminement d'un attribut mixte figure sur l'arbre suivant



dans la règle syntaxique <P> = <F_1> <F_2> associée à cet arbre, on écrit les définitions d'attributs suivantes :

$H.AD(F_1) = H.AD(P)$
$H.AD(F_2) = S.AD(F_1)$
$S.AD(P) = S.AD(F_2)$

Nous écrirons les modèles sémantiques dans le langage de définition d'attributs défini pour DELTA, qui permet de manipuler des types discrets et des types structurés.

Nous allons maintenant décrire l'utilisation de DELTA pour la traduction d'ALGOL 60 en ℓi_1. Ceci peut être considéré comme une méthodologie de la traduction d'ALGOL 60 en ℓi_1 qui comporte la définition d'attributs comportant leur nature (hérité, synthétisé ou mixte) leur type, leur fonction. Cette méthodologie peut servir de référence à un auteur de compilateur qui aurait décrit la grammaire de ce langage de façon un peu. différente avec son propre vocabulaire de non terminaux.

Parmi les attributs ainsi définis un grand nombre ne sont pas spécifiques à ALGOL 60. Ils sont spécifiques soit à la traduction des langages procéduraux à structure de blocs (il en est ainsi des attributs permettant de former le descriptif d'un objet, de l'attribut table des symboles), soit à la génération proprement dite (citons l'attribut permettant de numéroter les triplets générés, et l'attribut portant la chaîne de caractères générée).

Pour traduire un autre langage source comme PL/1, d'autres attributs sont nécessaires. Nous ne ferons pas ici une liste exhaustive de ces derniers, nous citerons seulement deux exemples.

Nous avons montré au paragraphe 7.1.3 qu'il est nécessaire d'associer à la valeur d'une variable label dans une instruction de branchement le niveau de procédure en cours, le numéro d'ordre et le niveau d'imbrication du bloc en cours. La syntaxe ℓi_1 de la traduction d'une instruction de branchement à une variable label étant :

ALLERA (entier,(entier,entier))* <descriptif>

Dans la traduction de PL/1 il faudra donc associer à chaque bloc 2 attributs un attribut mixte numéro d'ordre du bloc et un attribut hérité portant le niveau d'imbrication du bloc. Les structures de données spécifiques à PL/1 comme les structures fournissent un autre exemple de définition d'attribut.

En effet, dans la description de la construction strucutre, nous avons fait intervenir le rang du triplet de fin de génération de chaque sous-structure. Rappelons que la syntaxe d'un déclareur effectif de structure est

<déclareur effectif de structure> =

STRUCTURE[COMPRESSE](entier,entier),<champ effectif>,{<champ effectif>}*

champ effectif = (entier,entier) déclareur effectif

Il faudra donc prévoir un attribut qui décompte localement le nombre de triplets générés pour chaque sous-structure.

2. LA TRADUCTION D'ALGOL 60 EN LI_1

Nous ne décrirons ici que les attributs ayant une fonction générale dans la traduction en ℓi_1 des langages à structure de blocs, ainsi que cela a été indiqué dans le paragraphe précédent.

Dans ce qui suit, nous préciserons la fonction de chaque attribut, en spécifiant dans quelles règles syntaxiques il doit apparaitre, dans quelles règles on doit remplir sa valeur, dans quelles règles on doit la modifier.

Nous détaillerons successivement l'adressage en ℓi_1, la construction de la table des symboles, le traitement des étiquettes, la génération de triplets.

2.1 L'adressage des objets : les attributs du descriptif

L'adresse ℓi_1 d'une donnée est un doublet de deux nombres entiers se composant :

- du niveau de procédure
- du rang de l'objet dans la procédure où il est déclaré (nous l'appellerons ci-dessous adresse relative pour éviter la confusion avec rang de génération).

Nous définissons donc deux attributs :

- NP attribut hérité a pour valeur le niveau de procédure
- AD attribut mixte a pour valeur l'adresse relative.

2.1.1 *L'attribut hérité NP niveau de procédure*

Le bloc le plus externe porte le niveau de procédure 0.

L'attribut NP est incrémenté de 1 à chaque déclaration de procédure, il est transmis à chaque déclaration de donnée.

<PROG> = <BLOC>

≠ NP(BLOC) = 0

Le bloc le plus externe porte le niveau de procédure 0.

<PROG> = <COMPOSINS>

≠ NP(COMPOSINS)

L'instruction composée la plus externe porte le niveau de procédure 0.

<BLOC> = <DECPART><INSPART>

≠ NP(DECPART) = NP(BLOC)

≠ NP(INSPART) = NP(BLOC)

<DECPART> = BEGIN <DEC>

≠ NP(DEC) = NP(DECPART)

<DECPART> = <DECPART><DEC>

≠ NP(DEC) = NP(DECPART)

≠ NP(DECPART') = NP(DECPART)

deux procédures parallèles font obligatoirement partie de ce sous-arbre, c'est pourquoi on transmet la valeur initiale du niveau de procédure à chaque descendant.

<COMPOSINS> = BEGIN <INSPART> END

≠ NP(INSPART) = NP(COMPOSINS)

<INSPART> = <INS><END>

≠ NP(INS) = NP(INSPART)

<INSPART> = <INS> ; <INSPART>

≠ NP(INS) = NP(INSPART)

≠ NP(INSPART') = NP(INSPART)

<INS> = <APPELPROC>

≠ NP(APPELPROC) = NP(INS)

<INS> = <COMPOSINS>

≠ NP(COMPOSINS) = NP(INS)

<DEC> = <DECPROC>

≠ NP(DECPROC) = NP(DEC)

<DECPROC> = PROCEDURE <TETEPROC><CORPSPROC>

≠ NP(TETEPROC) = NP(DECPROC)

≠ NP(CORPSPROC) = NP(DECPROC) + 1

on transmet au non terminal <TETEPROC> le niveau de la procédure courante, car cette règle syntaxique définit la reconnaissance d'un objet procédure qui doit être adressé dans la procédure courante. On incrémente NP uniquement pour le corpsde la procédure.

Quand on a fini de lire la procédure, on revient automatiquement au niveau précédent par la sémantique de la règle

<DECPART> = <DECPART><DEC>

<DEC> = <ARITDEC>

≠ NP(ARITDEC) = NP(DEC)

<ARITDEC> = <TYPE><LVAR>

≠ NP(LVAR) = NP(ARITDEC)

<CORPSPROC> = <INS>

≠ NP(INS) = NP(CORPSPROC)

2.1.2 *L'attribut mixte AD : adresse relative dans la procédure*

On doit utiliser un attribut mixte pour transmettre à la suite des déclarations la nouvelle adresse relative dans la procéure.

H.AD attribut hérité contient la valeur de l'adresse relative, S.AD attribut synthétisé contient l'adresse relative à transmettre à la donnée suivante (sauf pour les déclarations factorisées de données de même type pour lesquelles c'est S.AD).

Dans la règle sémantique de déclaration de procédure, il faut réinitialiser à 1 la valeur de l'adresse relative.

<PROG> = <BLOC>

≠ H.AD(BLOC) = 1

la valeur de l'adresse relative de la première donnée déclarée est 1.

<PROG> = <COMPOSINS>

≠ H.AD(COMPOSINS) = 1

la valeur du descriptif de la première donnée déclarée est 1.

<BLOC> = <DECPART><INSPART>

≠ H.AD(DECPART) = H.AD(BLOC)

≠ H.AD(INSPART) = S.AD(DECPART)

≠ S.AD(BLOC) = S.AD(INSPART)

<DECPART> = BEGIN <DEC>

≠ H.AD(DEC) = H.AD(DECPART)

≠ S.AD(DECPART) = S.AD(DEC)

<DECPART> = <DECPART><DEC>

≠ H.AD(DECPART') = H.AD(DECPART)

≠ H.AD(DEC) = S.AD(DECPART') + 1

≠ S.AD(DECPART) = S.AD(DEC)

on prépare l'adresse relative de la déclaration suivante

<DEC> = <ARITDEC>

≠ H.AD(ARITDEC) = H.AD(DEC)

≠ S.AD(DEC) = S.AD(ARITDEC)

<LVAR> = ID

≠ S.AD(LVAR) = H.AD(LVAR)

<LVAR> = <LVAR>, ID

≠ H.AD(LVAR') = H.AD(LVAR)

≠ S.AD(LVAR) = S.AD(LVAR') + 1

<TABDEC> = <TYPE> ARRAY <ARRAYLIST>

≠ H.AD(ARRAYLIST) = H.AD(TABDEC)

≠ S.AD(TABDEC) = S.AD(ARRAYLIST)

<ARRAYLIST> = <ARRAYSEG>

≠ H.AD(ARRAYSEG) = H.AD(ARRAYLIST)

≠ S.AD(ARRAYLIST) = S.AD(ARRAYSEG)

<ARRAYLIST> = <ARRAYLIST>, <ARRAYSEG>

≠ H.AD(ARRAYLIST') = H.AD(ARRAYLIST)

≠ H.AD(ARRAYSEG) = S.AD(ARRAYLIST')

≠ S.AD(ARRAYLIST) = S.AD(ARRAYSEG)

<ARRAYLIST> = ID(<LISTBORNES>)

≠ S.AD(ARRAYSEG) = H.AD(ARRAYSEG) + 1

un tableau comme n'importe quelle donnée occupe une unité de la mémoire logique de ℓi_1.

<DEC> = <DECPROC>

≠ H.AD(DECPROC) = H.AD(DEC)

≠ S.AD(DEC) = H.AD(DEC)

après avoir exploré la procédure imbriquée dans <DECPROC> , on revient au niveau précédent pour adresser éventuellement les objets déclarés dans la suite.

<DECPROC> = PROCEDURE <TETEPROC><CORPSPROC>

≠ H.AD(TETEPROC) = H.AD(DECPROC)

≠ H.AD(CORPSPROC) = 0

≠ S.AD(DECPROC) = S.AD(CORPSPROC)

l'objet de mode procédure ainsi déclaré est désigné en ALGOL 60 par un identificateur reconnu dans <TETEPROC> , on transmet donc l'adresse relative par H.AD(TETEPROC) = H.AD(DECPROC).

On initialise à 0 l'adresse relative pour les objets déclarés dans le corps de la procédure par H.AD(CORPSPROC) = 0.

2.2. L'attribut mixte TAB : table des symboles

La table des symboles contient les objets valides dans chaque bloc. Il faut mémoriser une description complète de chaque objet (identificateur, descriptif, mode) de façon à contrôler les références aux objets et compléter la description des références aux objets en ℓi_1.

Ces informations sont mémorisées dans l'attribut mixte TAB, dont le type structuré est décrit ci-dessous . Nous appelons symbole ce type.


```
type symbole =[id:chaîne ; np : rang ; al : rang ; t : typescalaire ;
     g : genre cas g = "sca" : [] ; "tableau" : [dim : rang] ;
     "procedure" :[parametres : liste de param] ;
      autre , fcas] ;
type param = [objet : symbole ; passage : mode]
type typescalaire = ["entier", "réel", "logique"]
type mode = ["valeur", "nom"]
```

En ALGOL 60 le type d'une variable simple ne peut être que arithmétique (entier, réel) ou logique. Le mode de passage des paramètres mémorisé dans le type discret mode ne peut être que valeur ou nom.

La seule construction d'objet possible à partir d'un type simple est la construction "tableau".

Nous allons maintenant décrire le cheminement des parties héritée et synthétisée de l'attribut miste TAB, ainsi que les liens entre eux.

<PROG> = <BLOC>

≠ H.TAB(BLOC) = {∅} initialisation dans l'axiome

<PROG> = <COMPOSINS>

≠ H.TAB(COMPOSINS) = {∅} idem.

<BLOC> = <DECPART> ; <INSPART>

≠ H.TAB(INSPART) = S.TAB(DECPART)

dans la règle de définition d'un bloc, il faut transmettre à la partie instruction, la table des symboles remplie dans la partie déclaration.

≠ H.TAB(DECPART) = H.TAB(BLOC)

les symboles déclarés dans le bloc englobant sont valides dans le bloc englobé.

≠ S.TAB(BLOC) = H.TAB(BLOC)

les symboles valides dans le bloc englobé ne sont pas valides dans le bloc englobant, on revient ainsi au niveau précédent d'imbrication.

<DECPART> = BEGIN <DEC>

≠ H.TAB(DEC) = H. TAB(DECPART)

≠ S.TAB(DECPART) = S.TAB(DEC)

<DECPART> = <DECPART> ; <DEC>

≠ S.TAB(DECPART) = S.TAB(DEC)

≠ H.TAB(DECPART') = H.TAB(DEC)

≠ H.TAB(DEC) = S.TAB(DECPART')

on transmet à la déclaration suivante la table des symboles déjà constituée.

<DEC> = <ARITDEC>

≠ H.TAB(ARITDEC) = H.TAB(DEC)

≠ S.TAB(DEC) = S.TAB(ARITDEC)

<ARITDEC> = <TYPE><LVAR>
≠ H.TAB(LVAR) = H.TAB(ARITDEC)
≠ S.TAB(ARITDEC) = S.TAB(LVAR)
≠ TI(LVAR) = TEXTOBJET(TYPE)
<LVAR> = ID
≠ S.TAB(LVAR) = H.TAB(LVAR) ∪ {id = TEXT(ID) ; np = NP(LVAR),
ad = S.AD(LVAR) ; t = TI(LVAR) ; g = "sca"[]}
<LVAR> = <LVAR> , ID
≠ H.TAB(LVAR') = H.TAB(LVAR)
≠ S.TAB(LVAR) = S.TAB(LVAR') ∪ {id = TEXT(ID) ; np = NP(LVAR) ;
ad = S.AD(LVAR) ; t = TI(LVAR) ; g = "sca"[]}
≠ TI(LVAR') = TI(LVAR)
<DEC> = <TABDEC>
≠ H.TAB(TABDEC) = H.TAB(DEC)
≠ S.TAB(DEC) = S.TAB(TABDEC)
<TABDEC> = <TYPE> ARRAY <ARRAYLIST>
≠ H.TAB(ARRAYLIST) = H.TAB(TABDEC)
≠ S.TAB(TABDEC) = S.TAB(ARRAYLIST)
≠ TI(ARRAYLIST) = TI(TABDEC)
<ARRAYLIST> = <ARRAYSEG>
≠ H.TAB(ARRAYSEG) = H.TAB(ARRAYLIST)
≠ S.TAB(ARRAYLIST) = S.TAB(ARRAYSEG)
≠ TI(ARRAYSEG) = TI(ARRAYLIST)
<ARRAYLIST> = <ARRAYLIST> , <ARRAYSEG>
≠ H.TAB(ARRAYLIST') = H.TAB(ARRAYLIST)
≠ H.TAB(ARRAYSEG) = S.TAB(ARRAYLIST')
≠ S.TAB(ARRAYLIST) = S.TAB(ARRAYSEG)
≠ TI(ARRAYLIST') = TI(ARRAYLIST)
≠ TI(ARRAYSEG) = TI(ARRAYLIST)
<ARRAYSEG> = ID(LISTBORNES)
≠ S.TAB(ARRAYSEG) = H.TAB(ARRAYSEG) ∪ {id = TEXT(ID) ; np = NP(ARRAYSEG) ;
ad = H.AD(ARRAYSEG) ; t = TI(ARRAYSEG) ; g = "tableau" ; dim = NB(LISTBORNES)}

la mémorisation des caractéristiques des paramètres formels d'une procédure se fait de façon semblable.

Examinons maintenant comment on transmet la table des symboles aux instructions pour effectuer des tests contextuels.

<INSPART> = <INS> END
≠ H.TAB(INS) = H.TAB(INSPART)
≠ S.TAB(INSPART) = S.TAB(INS)
<INSPART> = <INS> ; <INSPART>
≠ H.TAB(INS) = H.TAB(INSPART)
≠ H.TAB(INSPART') = H.TAB(INSPART)
≠ S.TAB(INSPART) = S.TAB(INSPART')

2.3 Le traitement des étiquettes

Les étiquettes sont définies dans les instructions étiquetées
- soit une instruction "début de bloc"
- soit une instruction quelconque.

On peut trouver une référence à une étiquette (instruction de branchement) avant ou après la définition de cette étiquette, on retrouve le classique problème des références avant. Pour le résoudre, on va définir deux attributs ayant pour valeur des ensembles d'étiquettes
- l'attribut mixte GET qui chemine dans l'arbre syntaxique de gauche à droite et qui permet de résoudre les étiquettes définies avant leur utilisation,
- l'attribut DET qui chemine dans l'arbre syntaxique de droite à gauche et qui permet de résoudre le problème des références avant.

Lorsqu'on trouve une instruction de branchement, on doit vérifier que l'étiquette se trouve bien dans l'un ou l'autre de ces ensembles.

Par convention, on place les étiquettes d'un bloc étiqueté dans l'ensemble GET.

<PROG> = <BLOC>
≠ H.GET(BLOC) = {∅}
≠ H.DET(BLOC) = {∅}

On initialise l'ensemble des étiquettes

<PROG> = <COMPOSINS>
≠ H.GET(COMPOSINS) = {∅}
≠ H.DET(COMPOSINS) = {∅}

idem.

<BLOCG> = <BLOC>
≠ H.GET(BLOC) = H.GET(BLOCG)
≠ S.GET(BLOCG) = S.GET(BLOC)
≠ H.DET(BLOC) = H.DET(BLOCG)
≠ S.DET(BLOCG) = S.DET(BLOC)

<BLOCG> = ID ; <BLOC>

≠ H.GET(BLOC) = H.GET(BLOCG) ∪ {TEXT(ID)}

≠ S.GET(BLOCG) = S.GET(BLOC)

≠ H.DET(BLOC) = H.DET(BLOCG)

≠ S.DET(BLOCG) = H.DET(BLOC6)

<BLOC> = <DECPART> ; <INSPART>

≠ H.GET(DECPART) = H.GET(BLOC)

≠ H.GET(INSPART) = S.GET(DECPART)

≠ S.GET(BLOC) = H.GET(BLOC)

≠ H.DET(INSPART) = H.DET(BLOC)

≠ H.DET(DECPART) = S.DET(INSPART)

≠ S.DET(BLOC) = H.DET(BLOC)

lorsqu'on a fini de traiter un bloc on revient au niveau inférieur d'imbrication du bloc, puisque seules les étiquettes du bloc externe sont valides.

<DECPART> = BEGIN <DEC>

≠ S.GET(DECPART) = S.GET(DEC)

≠ H.GET(DEC) = H.GET(DECPART)

≠ S.DET(DECPART) = S.DET(DEC)

≠ H.DET(DEC) = H.GET(DECPART)

<DECPART> = <DECPART><DEC>

≠ H.GET(DEC) = S.GET(DECPART')

≠ H.GET(DECPART') = H.GET(DECPART)

≠ S.GET(DECPART) = S.GET(DEC)

≠ H.DET(DECPART') = H.DET(DECPART)

≠ H.DET(DEC) = S.DET(DECPART')

≠ S.DET(DECPART) = S.DET(DECPART')

<INSPART> = <INS> END

≠ S.GET(INSPART) = S.GET(INS)

≠ H.GET(INS) = H.GET(INSPART)

≠ S.DET(INSPART) = S.DET(INS)

≠ H.DET(INS) = H.DET(INSPART)

<INS> = <ETISTAT>

≠ H.GET(ETISTAT) = H.GET(INS)

≠ S.GET(INS) = S.GET(ETISTAT)

≠ H.DET(ETISTAT) = H.DET(INS)

≠ S.DET(INS) = S.DET(ETISTAT)

<ETISTAT> = ID : <NONETIQ>

≠ H.GET(NONETIQ) = H.GET(ETISTAT)

≠ S.GET(ETISTAT) = H.GET(ETISTAT) ∪ {TEXT(ID)}

≠ H.DET(NONETIQ) = H.DET(ETISTAT)

≠ S.DET(ETISTAT) = S.DET(ETISTAT) ∪ {TEXT(ID)}

<NONETIQ> = GOTO ID

≠ S.ET(NONETIQ) = H.ET(NONETIQ)

si TEXT(ID) ∈ S.GET(NONETIQ) ∪ S.DET(NONETIQ)

alors générer code du branchement

sinon "erreur"

2.4 La génération de triplets : les attributs RG et TEXTOBJET

2.4.1 *Le principe*

Le langage ℓi_1 est constitué de triplets, il faut donc compter les triplets au fur et à mesure de leur production. On utilise à cet effet l'attribut mixte RG (rang du triplet produit).

La partie héritée de l'attribut mixte RG, H.RG porte le numéro du premier triplet de la génération.

La partie synthétisée de l'attribut mixte RG, S.RG porte le numéro du dernier triplet de la génération. Les liens entre H.RG et S.RG sont parfaitement définis, suivant qu'un non terminal produit respectivement 0, 1, 2 triplets, ces relations sont respectivement :

S.RG = H.RG - 1

S.RG = H.RG

S.RG = H.RG + 1

Le compteur de triplets est incrémenté chaque fois qu'on reconnaît une nouvelle instruction ou une nouvelle déclaration. Ainsi dans la règle sémantique définissant une suite d'instructions, on incrémente la valeur du compteur de triplets :

<INSPART> = <INS> ; <INSPART>

H.RG(INSPART') = S.RG(INS) + 1

De même dans la règle syntaxique définissant une suite de déclarations on incrémente la valeur du compteur de triplets :

<DECPART> = <DECPART><DEC>

H.RG(DEC) = S.RG(DECPART') + 1

Le programme généré en $\&i_1$ est une suite de triplets. Chaque triplet généré est une chaîne de caractères qu'on concatène à la chaîne de caractères déjà générée pour compléter le programme.

On synthétise ainsi la chaîne du programme généré à l'aide de l'attribut TEXTOBJET. Les triplets générés pouvant contenir des références à d'autres triplets l'attribut TEXTOBJET met en oeuvre RG. Nous allons donc définir les modèles sémantiques correspondant à ces deux attributs dans ce qui suit.

<PROG> = <BLOC>
    ≠ H.RG(BLOC) = 1
      (Par convention le premier triplet du programme porte le numéro 1)
    ≠ S.RG(PROG) = S.RG(BLOC)
    ≠ TEXTOBJET(PROG) = TEXTOBJET(BLOC)

<PROG> = <COMPOSINS>
    ≠ H.RG(COMPOSINS) = 1
      (Par convention le premier triplet du programme porte le numéro 1)
    ≠ S.RG(PROG) = S.RG(COMPOSINS)
    ≠ TEXTOBJET(PROG) = TEXTOBJET(COMPOSINS)

<BLOC> = <DECPART> , <INSPART>
    ≠ H.RG(DECPART) = H.RG(BLOC)
    ≠ H.RG(INSPART) = S.RG(DECPART) + 1
    ≠ S.RG(BLOC) = S.RG(INSPART)
    ≠ TEXTOBJET(BLOC) = TEXTOBJET(DECPART) || TEXTOBJET(INSPART)
      (On concatène la chaîne générée pour les instructions à la chaîne générée pour les déclarations).

<DECPART> = BEGIN <DEC>
    ≠ H.RG(DEC) = H.RG(DECPART)
    ≠ S.RG(DECPART) = S.RG(DEC)
    ≠ TEXTOBJET(DECPART) = TEXTOBJET(DEC)

<DECPART> = <DECPART> , <DEC>
    ≠ H.RG(DECPART') = H.RG(DECPART)
    ≠ H.RG(DECPART) = S.RG(DECPART') + 1
    ≠ S.RG(DECPART) = S.RG(DEC)
    ≠ TEXTOBJET(DECPART) = TEXTOBJET(DECPART') || TEXTOBJET(DEC)

<INSPART> = <INS> END
    ≠ H.RG(INS) = H.RG(INSPART)
    ≠ S.RG(INSPART) = S.RG(INS)
    ≠ TEXTOBJET(INSPART) = TEXTOBJET(INS)

<INSPART> = <INS><INSPART>
    ≠ H.RG(INS) = H.RG(INSPART)
    ≠ H.RG(INSPART') = S.RG(INS) + 1
    ≠ S.RG(INSPART) = S.RG(INSPART')
    ≠ TEXTOBJET(INSPART) = TEXTOBJET(INS) || TEXTOBJET(INSPART')

2.4.2 *La génération de la déclaration des variables arithmétiques*

<DEC> = <ARITDEC>
    ≠ H.RG(ARITDEC) = H.RG(DEC)
    ≠ S.RG(DEC) = S.RG(ARITDEC)

<ARITDEC> = <TYPE><LVAR>
    ≠ H.RG(LVAR) = H.RG(ARITDEC)
    ≠ S.RG(ARITDEC) = S.RG(LVAR)
    ≠ TEXTOBJET(ARITDEC) = TEXTOBJET(LVAR)

<LVAR> = ID
    ≠ S.RG(LVAR) = H.RG(LVAR)
    ≠ TEXTOBJET(LVAR) = 'ALLC' || '(NP(LVAR), S.AD(LVAR)) || 'H.T(LVAR)'
      (On génère dans cette règle un triplet de déclaration de variable arithmétique, conformément à la syntaxe $\&i_1$
      ALLC <descriptif> <modeprimitif>
      H.T est la partie héritée d'un attribut mixte T qui a pour valeur le type arithmétique ; la partie synthétisée est définie dans la règle de définition syntaxique du type : exemple <TYPE> = ENTIER S.T(TYPE) = E).

<LVAR> = <LVAR> , ID
    ≠ H.RG(LVAR') = H.RG(LVAR)
    ≠ S.RG(LVAR) = S.RG(LVAR') + 1
    ≠ TEXTOBJET(LVAR) = TEXTOBJET(LVAR') || 'ALLC' || '(NP(LVAR), S.AD(LVAR))' ||
      'H.T(LVAR)'

2.4.3 *La génération des déclarations de tableaux*

<DEC> = <TABDEC>
≠ H.RG(TABDEC) = H.RG(DEC)
≠ S.RG(TABDEC) = S.RG(DEC)
≠ TEXTOBJET(DEC) = TEXTOBJET(TABDEC)
<TABDEC> = <TYPE> <ARRAYLIST>
≠ H.RG(ARRAYLIST) = H.RG(TABDEC)
≠ S.RG(TABDEC) = S.RG(ARRAYLIST)
≠ TEXTOBJET(TABDEC) = TEXTOBJET(ARRAYLIST)
≠ H.T(ARRAYLIST) = S.T(TYPE)
<ARRAYLIST> = <ARRAYSEG>
≠ H.RG(ARRAYSEG) = H.RG(ARRAYLIST)
≠ S.RG(ARRAYLIST) = S.RG(ARRAYSEG)
≠ TEXTOBJET(ARRAYLIST) = TEXTOBJET(ARRAYSEG)
<ARRAYLIST> = <ARRAYLIST><ARRAYSEG>
≠ H.RG(ARRAYLIST') = H.RG(ARRAYLIST)
≠ H.RG(ARRAYSEG) = S.RT(ARRYLIST') + 1
≠ S.RG(ARRAYLIST) = S.RG(ARRAYSEG)
≠ TEXTOBJET(ARRAYLIST) = TEXTOBJET(ARRAYLIST') || TEXTOBJET(ARRAYSEG)
<ARRAYSEG> = ID (<LISTBORNES>)
≠ H.RG(LISTBORNES) = H.RG(ARRAYSEG) + 1
≠ S.RG(ARRAYSEG) = S.RG(LISTBORNES)
≠ TEXTOBJET(ARRAYSEG) = 'ALLC' || '(NP(ARRAYSEG), H.AD(ARRAYSEG))' || 'TAB' ||
'NB(LISTBORNES)' ; '|| TEXTOBJET(LISTBORNES)' ; || 'H.T(ARRAYSEG)'
(conformément à la syntaxe $\mathcal{L}i_1$ de déclaration d'un tableau
ALLC <descriptif> TAB[COMPRESSE]entier;<dim>;{<dim>}* <modeprimitif>
Le nombre de dimensions est porté par l'attribut synthétisé NB(LISTBORNES)
rempli dans les règles suivantes de définition des bornes).
<LISTBORNES> = <PAIREBORNE>
≠ H.RG(PAIREBORNE) = H.RG(LISTBORNES)
≠ S.RG(LISTBORNES) = S.RG(PAIREBORNE)
≠ TEXTOBJET(LISTBORNES) = TEXTOBJET(PAIREBORNE)
≠ NB(LISTBORNES) = 1.

<LISTBORNES> = <LISTBORES> , <PAIREBORNE>
≠ H.RG(LISTBORNES') = H.RG(LISTBORNES)
≠ H.RG(PAIREBORNE) = S.RG(LISTBORNES) + 1
≠ S.RG(LISTBORNES) = S.RG(PAIREBORNE)
≠ TEXTOBJET(LISTBORNES) = TEXTOBJET(LISTBORNES') || TEXTOBJET(MAIREBORNE)
≠ NB(LISTBORNES) = NB(LISTBORNES') + 1
<LISTBORNES> = <PAIREBORNE>
≠ H.RG(PAIREBORNE) = H.RG(LISTBORNES)
≠ S.RG(LISTBORNES) = S.RG(PAIREBORNE)
≠ TEXTOBJET(LISTBORNES) = TEXTOBJET(LISTBORNES') || TEXTOBJET(PAIREBORNE)
<PAIREBORNE> = <EXP> : <EXP'>
≠ H.RG(EXP) = H.RG(PAIREBORNE)
≠ H.RG(EXP') = S.RG(EXP) + 1
≠ TEXTOBJET(PAIREBORNE) = TEXTOBJET(EXP) || TEXTOBJET(EXP') || 'DIM E' ||....
(pour générer le triplet de descriptif d'une dimension, il faut savoir
si EXP et EXP' sont de véritables expressions arithmétiques. On peut en
décider grâce à l'attribut genre défini au paragraphe suivant
- si EXP et EXP' sont de véritables expressions on génèrera :
'DIM E' || 'S.RG(EXP)' || 'T' || ',' || 'S.RG(EXP')' || 'T'
- si la valeur de chaque borne est la valeur d'une variable on génère
'DIM E' || '(NP(EXP),S.AD(EXP))'|| ',' || '(NP(EXP'),S.AD(EXP'))' ...).

2.4.4 *Le traitement des expressions arithmétiques*

Rappelons la syntaxe des opérations associées aux modes primitifs
<opérationsimple> = <binairesimple> <modeprimitif> <opérande>, <operande>
<unairesimple> <modeprimitif> <opérande>
<opérande> = <référencevaleur>
<référencevaleur> = {<dérepérage>*}<descriptif> | *(entier T) | <valeur> c | entier 1

Pour savoir si la valeur de l'opérande est la valeur d'un triplet on va synthétiser un attribut de type logique GENRE.
Si GENRE = 1 la valeur de l'opérande est la valeur d'un triplet
Si GENRE = 0 la valeur de l'opérande n'est pas la valeur d'un triplet.

Les règles suivantes permettent de calculer l'attribut GENRE

- le genre d'un non terminal décrivant une opération arithmétique est toujours 1
- le genre d'une variable indicée est 1 car on doit d'abord générer le calcul de l'adresse. La valeur d'une variable indicée est obtenue en dérepérant la valeur du triplet adresse.

A chaque triplet d'opération on doit associer le mode primitif de son résultat conformément à la syntaxe précédente. Le type du résultat dépend du type des opérandes ; on va donc synthétiser les types des opérandes par un attribut T

- soit une règle syntaxique décrivant une opération arithmétique, si les types des opérandes sont différents, il faudra convertir le type d'un des opérandes conformément aux règles de conversion du langage,
- si les types des opérandes sont identiques, le type du résultat est identique aux types des opérandes.

```
<EXP> = <TERME>
     ≠ GENRE(EXP) = GENRE(TERME)
     ≠ T(EXP) = T(TERME)
     ≠ H.RG(EXP) = H.RG(EXP)
     ≠ TEXTOBJET(EXP) = TEXTOBJET(TERME)
<EXP> = <EXP> + <TERME>
     ≠ GENRE(EXP) = 1
     ≠ T(EXP)
       si(T(EXP') = 'E' et T(TERME) = 'E') alors T(EXP) = 'E'
       sinon T(EXP) = 'R'
     ≠ H.RG(EXP') = H.RG(EXP)
     ≠ H.RG(TERME) = S.RG(EXP') + 1
     ≠ S.RG(EXP)
       si(T(EXP') = T(TERME) alors S.RG(EXP) = S.RG(TERME) + 1
       sinon S.RG(EXP) = S.RG(TERME + 2)
       (si conversion il y a on génère deux triplets).
     ≠ TEXTOBJET(EXP)
       si(GENRE(EXP') ∩ GENRE(TERME)) = 1
          si T(EXP') = T(TERME) alors TEXTOBJET(EXP) =
       '±' || 'T(EXP)' || 'S.RG(EXP')' || 'T' || ',' || 'S.RG(TERME)' || 'T'
          sinon TEXTOBJET(EXP) = CONVERSION ... ||
        ± || 'T(EXP)' || 'S.RG(EXP')' || 'T' || ',' || 'S.RG(TERME)' || 'T'
        sinon si (GENRE(EXP') ∪ GENRE(TERME)) = 0
          alors si T(EXP') = T(TERME) alors TEXTOBJET(EXP) =
       '±' || 'T(EXP')' || TEXTOBJET(EXP') || ',' || TEXTOBJET(TERME)
              sinon TEXTOBJET(EXP) = CONVERSION ...  ||
```



```
        ± || 'T(EXP')' || TEXTOBJET(EXP') || TEXTOBJET(TERME)
        sinon si GENRE(EXP') = 1
          alors si T(EXP') = T(TERME) alors TEXTOBJET(EXP) =
        ± || 'T(EXP')' || 'S.RG(EXP')' || 'T' || ',' || TEXTOBJET(TERME)
          sinon TEXTOBJET(EXP) = CONVERSION ... ||
        ± || 'T(EXP')' || 'S.RG(EXP')' || 'T' || ',' || TEXTOBJET(TERME)
          sinon si T(EXP) = T(TERME) alors TEXTOBJET(EXP) =
        ± || 'T(EXP')' || TEXTOBJET(EXP') || ',' || 'S.RG(TERME)' || 'T'
          sinon TEXTOBJET(EXP) = CONVERSION ...
<TERME> = <FACTEUR>
     ≠ GENRE(TERME) = GENRE(FACTEUR)
     ≠ T(TERME) = T(FACTEUR)
     ≠ H.RG(FACTEUR) = H.RG(TERME)
     ≠ S.RG(TERME) = S.RG(FACTEUR)
     ≠ TEXTOBJET(TERME) = TEXTOBJET(FACTEUR)
<TERME> = <TERME> <FACTEUR>
     ≠ GENRE(TERME) = 1
     ≠ T(TERME)
       si (T(TERME') = 'E' et T(FACTEUR) ='E') alors T(TERME) = 'E'
       sinon T(TERME) = 'R'
     ≠ TEXTOBJET(TERME)
       faire un traitement analogue à celui de la règle
        EXP  =  EXP  =  TERME
<TERME> = <TERME> / <FACTEUR>
     ≠ GENRE(TERME) = 1
     ≠ T(TERME) = 'R'
       (le résultat de la division de deux nombres entiers est un nombre réel).
     ≠ TEXTOBJET(TERME)
       faire un traitement analogue à celui de la règle
<FACTEUR> = <PRIMAIRE>
     ≠ GENRE(FACTEUR) = GENRE(PRIMAIRE)
     ≠ T(FACTEUR) = T(PRIMAIRE)
     ≠ TEXTOBJET(FACTEUR) = TEXTOBJET(PRIMAIRE)
```

Troisième partie

LE LANGAGE ℓi_2 POUR LES MACHINES A REGISTRES ET MEMOIRE ADRESSABLES.

1. LES CARACTERISTIQUES DU LANGAGE ℓi_2

Dans les machines à registres et à mémoire adressable, une grande partie des traitements consiste à prendre des mots en mémoire pour les amener dans les registres généraux, à les traiter dans l'unité centrale et à renvoyer les mots, constituant les résultats en mémoire centrale.

Pour l'essentiel, ces machines se différencient par le nombre de registres généraux et la méthode de calcul de l'adresse effective (par utilisation de déplacement, registre de base, indexation, indirection combinée ou non).

D'une façon générale, la caractéristique principale des langages d'assemblage est que la partie adresse des instructions contient l'adresse de la donnée et non la donnée elle-même. Seules les instructions traitant les quantités immédiates agissent sur les données.

Nous supposerons donc que le langage ℓi_2 est le langage d'assemblage d'une machine fictive dont les caractéristiques de l'unité de traitement sont les suivantes :

- le nombre de registres généraux et des registres d'index n'est pas borné.
- les registres généraux peuvent être utilisés comme registres de base et la forme générale du calcul de l'adresse effective est D (X,B) où,

D est un déplacement numérique
X un registre d'index
B un registre de base

l'indirection s'applique toujours avant l'indexation.

A chaque code opération d'un triplet est associé une sémantique particulière, donc une génération de code ℓi_2 spécifique. De même la référence à un objet qui est le descriptif en ℓi_1 , doit se faire en ℓi_2 par la référence à l'adresse en mémoire. Chaque fois qu'on trouvera dans le texte ℓi_2 un doublet (np, ad) il faudra le traduire par un calcul d'adresse. Afin d'éviter de répéter cette séquence de code ℓi_2 chaque fois qu'on trouve ce doublet dans le texte ℓi_2 , on va définir un sous-programme dont les paramètres d'entrée sont np, ad et le paramètre de sortie l'adresse de l'objet.

Ce sous-programme est un sous-programme ouvert puisqu'il constitue un squelette que l'on vient équiper pour constituer un programme utilisable. La spécification de ce sous-programme constitue donc une définition de macro.

Une macro est donc ici une séquence de code enfermée entre les mots clés MACRO et FINMACRO. Le nom de la macro apparait dans la ligne qui suit le mot clé MACRO, il est suivi par la liste des paramètres numériques et des paramètres adresses. Par convention les noms des adresses paramètres des macros sont

précédés d'un dollar.

Exemple :

```
MACRO
RES n, $VR
------
------
FINMACRO
```

est une macro possédant un paramètre numérique n et un paramètre de type adresse $VR.

Comme il est nécessaire de transmettre outre les adresses, des paramètres numériques nous dirons que les instructions globales dans lesquelles le code opération est constitué par le nom du sous-programme s'appelent des méta instructions.

Le traducteur de ℓi_2 en langage d'assemblage de la machine cible possède donc les caractéristiques d'un macro-assembleur.

Rappelons en le principe. Le macro-assembleur lors de l'analyse des instructions détecte les codes opérations non valables et va voir si ces symboles sont les noms de macro-définitions.

Dans ce cas il effectue la construction des instructions en prenant le prototype de la macro-instruction correspondante et en effectuant les substitutions demandées des symboles effectifs aux symboles indiqués dans la pseudo-instruction MACRO.

A l'intérieur de chaque macro les instructions ℓi_2 manipulent des registres. Comme le nombre de registres généraux de la machine fictive n'est pas borné, on n'a pas besoin de manipuler de mémoires de manoeuvre. Quand on a besoin d'utiliser un registre général et demander d'allouer un registre,

ALLOUER (R1) R1 est le nom du registre alloué

Pour un registre d'index on effectue une opération semblable

ALLOUERINDEX (X1)

Lorsqu'on n'a plus besoin du contenu d'un registre, on explicite dans le code ℓi_2 l'opération de libération, par la primitive LIBERER, qui accepte le nom de ce registre comme paramètre d'entrée.

La primitive ALLOUER rend le numéro du premier registre libre, et marque ce registre "occupé". La primitive LIBERER efface la marque "occupé" pour le regis libéré. Les prototypes de macros sont indépendants les uns des autres, en particulier pour ce qui concerne la gestion des registres.

Par conséquent à l'intérieur d'une macro, l'allocation des registres est indépe dante des allocations effectuées dans le reste du code ℓi_2. Le premier registr

alloué à l'intérieur d'une macro sera toujours le registre numéro 1. Lorsque le macro-assembleur fera l'expansion des prototypes, on trouvera donc dans le code ℓi_2 des allocations imbriquées du même registre : exemple

	Après expansion du prototype de TOTO
ALLOUER(R1)	ALLOUER(R1)
TOTO $A	ALLOUER(R1)
	LIBERER(R1)
LIBERER(R1)	LIBERER(1)

La signification de R1 doit être différente dans le deuxième allocation. Pour passer de ces registres fictifs aux registres effectifs, on fera donc une gestion en pile.

2. LA SYNTAXE DU LANGAGE ℓi_2

Comme le langage ℓi_2 est un langage d'assemblage, le format de ses instructions comporte de façon classique :

- un code opération.
- un champ indiquant le nom d'un registre.
- un champ d'adressage qui peut éventuellement comporter une quantité immédiate.

Afin de prévoir dans le code opération, les informations qui permettront de générer dans l'étape finale de traduction les langages d'assemblage des différentes machines cibles, il nous faut tenter de faire une classification entre petites et grosses machines pour ce qui concerne le langage d'assemblage. On peut dire que dans les petites machines les registres sont spécialisés (un seul registre étendu servant d'accumulateur, les registres d'index étant bien définis). Les codes opérations sont donc dédoublés pour les opérations de transfert des données.

Comme le langage ℓi_2 doit s'adapter à la traduction de ces deux types de langage d'assemblage nous allons donc dédoubler aussi le code dans le langage ℓi_2 .

Certains registres ont un rôle particulier dans le processus de traduction leur nom est un mot clé du langage ℓi_2 . Ce sont : le registre contenant l'adresse mémoire libre dans la pile dynamique (nous notons ce registre AML), le registre contenant l'adresse de base de la procédure en cours (nous notons ce registre RL), le registre contenant l'adresse de base de la procédure appelante (nous notons ce registre RLA), le registre contenant l'adresse de retour dans le code (noté AR).

<registre> = R entier | AML | RL | RLA | AR

On note un registre d'index <index> = X entier

Le champ adresse peut être décrit par la syntaxe suivante :

<adresse effective> = [<indirection>] <adresse>

<adresse> = ID [, (X entier)] . [POSIT (entier)] |
[<dep>] (X entier, R entier)

<dep > = entier | - entier | ID + entier | ID - entier.

L'emploi de POSIT (entier) dans la description de l'adresse effective a été défini pour introduire un déplacement supplémentaire exprimé en nombre de bits. Le bit n'étant pas une unité adressage directement dans les langages d'assemblage, on est en général obligé d'utiliser des masques et des instructions de décalage pour positionner une donnée à l'intérieur d'un octet.

Afin d'éviter d'alourdir inutilement le code ℓi_2 et puisque les instructions de décalage sont très variables en fonction des machines cibles, le langage ℓi_2 ne comporte pas d'instruction de décalage.

Comme dans tout langage d'assemblage, on peut distinguer les instructions de transfert de données, les opérations arithmétiques et logiques, les instructions de branchement.

2.1. Les instructions de transfert de données

Elles comportent les instructions de chargement et les instructions de rangement. On peut charger dans les registres des quantités situées dans la mémoire ou des quantités immédiates.

On peut ranger dans la mémoire le contenu d'un registre général ou d'un registre d'index.

<transfert donnée> = <chargement> | <chargement immédiat> |
<rangement>

<chargement> = <code chargement> <registre> , <adresse effective> |
LR <registre> , <registre> |
LX <index> , ID

<code chargement> = LB | LH | L | LD

LB signifie chargement d'un octet

LH signifie chargement d'un demi-mot

L signifie chargement d'un mot

L signifie chargement d'un demi-mot

<chargement immédiat> = LI <registre> , <valeur> |
LI <registre> , <adresse> |
LI <registre> , $ [<dep>] |
LIX <index> , <valeur>

<rangement> = <code rangement> <registre> , <adresse effective>
SX <index> , <adresse effective>

<code rangement> = SB | SH | S

SB signifie rangement dans un octet

SH signifie rangement dans un demi-mot

S signifie rangement dans un mot

2.2. Les instructions arithmétiques

Une opération arithmétique s'effectue entre deux opérandes dont l'un est toujours situé dans un registre qui joue alors le rôle d'accumulateur, le deuxième opérande peut se trouver soit dans la mémoire, soit dans un autre registre, soit être une quantité immédiate. On peut incrémenter ou décrémenter une quantité immédiate du contenu d'un registre d'index.

<arithmétique> = <aritnem> <registre> , <adresse effective> |
<aritreg> <registre> , <registre> | COMPL <registre>
<opim> <registre> , <valeur> |
<opindex> <index> , <valeur>

<aritnem> = AM | AH | SUB | SH | MM | MH | DM | DH

AM signifie ajouter au contenu du registre le contenu du mot

AH signifie ajouter au contenu du registre le contenu du demi-mot

Les autres alternatives se définissent de façon semblable.

<aritreg> = AR | SR | MR | DR

AR signifie ajouter le contenu des deux registres

SR signifie soustraire le contenu des registres

COMPL signifie complémenter le continu d'un registre

<opim> = AI| SI| MI| DI

AI signifie ajouter au contenu du registre la quantité immédiate.

SI signifie soustraire au contenu du registre la quantité immédiate.

<opindex> = AIX | SIX

AIX signifie incrémenter le contenu du registre d'index d'une quantité immédiate

SIX signifie décrémenter le contenu du registre d'index d'une quantité immédiate

2.3. Les instructions logiques

On peut définir pour les instructions logiques une syntaxe tout à fait analogue.

<logique> = <logmem> <registre> , ID [(X entier)] |
<logreg> <registre> , <registre>

<logmem> = E | O | X

E code du et logique

O code du ou logique

X code du ou exclusif

<logreg> = ER | OR | XR

Les significations sont identiques aux significations précédentes mais les deux opérandes sont situés dans des registres.

2.4. Les instructions de transfert de commande

Elles se composent de branchement inconditionnel et de branchement conditionnel. Comme tous les langages d'assemblage des machines cibles ne sont pas pourvus de code condition, le langage ℓi_2 ne possède pas d'instructions de branchement conditionnel faisant référence à un code condition.

Les instructions de branchement conditionnel font toutes référence au contenu du registre chargé dans l'instruction qui les précède.

On teste si le contenu de ce registre est nul, différent de zéro, positif, positif ou nul, négatif, négatif ou nul.

(Les codes opérations respectifs sont BZE, BNE, BP, BPN, BN, BNN).

Pour traduire les appels de sous-programme, il faut pouvoir disposer d'instructions de branchement dans lesquelles on mémorise l'adresse de retour dans le code dans le registre AR.

La syntaxe ℓi_2 des instructions de branchement est donc

<branchement> = <branchement inconditionnel> | <branchement conditionnel> |
<branchement retour>

<branchement inconditionnel> =

BRU <registre> | BRU <adresse> | BRU $ [<inc>]

<inc> =+ entier |- entier

<branchement retour> = BAL AR, R entier | BAL AR INS lettre

l'adresse de branchement est soit une adresse contenue dans un registre, soit une étiquette d'instruction INS lettre

<branchement conditionnel> = <bcond> <registre> | <bcond> <adresse> |
<bcond> $ <inc>

<bcond> = BZE | BNE | BP | BPN | BN | BNN

2.5. Les instructions de manipulation de chaînes

Afin d'éviter de décomposer les opérations de manipulation de chaînes en instructions logiques, on va prévoir pour les machines à octets des opérations de manipulations directes d'octets.

Celles-ci comportent une instruction de rangement immédiat d'un octet, une instruction de mouvement d'une chaîne d'un champ émetteur dans un champ récepteur, une instruction de transcodage. Leur syntaxe est :

MVI <adresse> , <valeur> | MVC <adresse> (entier), ID |
TR <adresse> (entier), <adresse>

3. DESCRIPTION DE L'ENCOMBREMENT DES DONNEES

Dans le texte $\&i_2$ produit comme objet de la première déclaration, les triplets de déclaration des objets (de code opération ALLC) ne se trouvent pas forcément tous en tête de chaque routine. En effet lorsque ce texte provient de la traduction d'un texte source ALGOL 68, les déclarations des variables sont effectuées au début de la traduction des propositions fermées.

Au cours de la seconde étape de traduction, on effectue une première passe sur le texte $\&i_2$ pour réarranger les déclarations en déplaçant tous les triplets ALLC en tête de chaque texte de routine.

C'est au cours de cette deuxième étape de traduction, qu'on fixe les choix d'implantation en spécifiant la place occupée en mémoire par chaque mode primitif, et en décrivant la fonction d'accès de chaque construction à l'aide de son vecteur de renseignements. On peut donc transformer au cours de la première passe sur le texte $\&i_2$, le rang de chaque donnée dans la procédure, en adresse relative exprimée en mots.

A titre d'exemple considérons le programme source suivant écrit en CPL/1 (CPL/1 est un interpréteur conversationnel qui accepte en entrée un sous-ensemble de PL/1)

```
E : PROC ;
    DCL FLIPFLOP LABEL ;
    DCL I FIXED (15) ;
    DCL J FIXED
    DCL K FIXED (15) ;
    DCL A CHAR (10) VARYING ;
    DCL T (1 : 5, - 1 : 2, 2 : 6) REAL BINARY FLOAT (24)
    ——
    ——

F : PROC ;
    DCL TT (1 : 5, 1 : K) REAL BINARY FLOAT (24) ;
    DCL CI CHAR (90) VARYING ;
    DCL P, Q POINTER,
    ——
    ——

    P = Q ;
    CI = CI|| A ;
    END F ;
    I = J + K ;
```


Rappelons que REAL est le seul attribut de mode reconnu par CPL/1, que BINARY est le seul attribut de base reconnu par CPL/1. L'échelle peut être soit FIXED, soit FLOAT. La précision est repérée par un entier sans signe dont les valeurs possibles sont 7, 15, 31, 24, 56.

Pour l'échelle FIXED, les précisions 7, 15, 31 traduisent le fait que la variable correspondante est rangée en mémoire sur un octet, un demi-mot, un mot respectivement.

Pour l'échelle FLOAT l'attribut de précision peut prendre pour valeur 24 ou 56, ce qui traduit le fait qu'un nombre flottant est codé sur un mot ou un double mot et que la mantisse occupe 24 ou 56 bits.

La fonction d'accès choisie pour implanter les tableaux à bornes variables est "la méthode des multiplicateurs". Nous verrons plus loin comment cette méthode est codée en $\&i_2$, rappelons simplement ici que le vecteur de renseignements est une zone de longueur égale à 2n+1 mot (si n est le nombre de dimensions du tableau).

Pour décrire l'adressage d'une chaîne de longueur variable, le vecteur de renseignements comporte 2 mots (le premier mot contient l'adresse du premier caractère de la chaîne, dans le deuxième mot sont mémorisées la longueur courante et la longueur maximale).

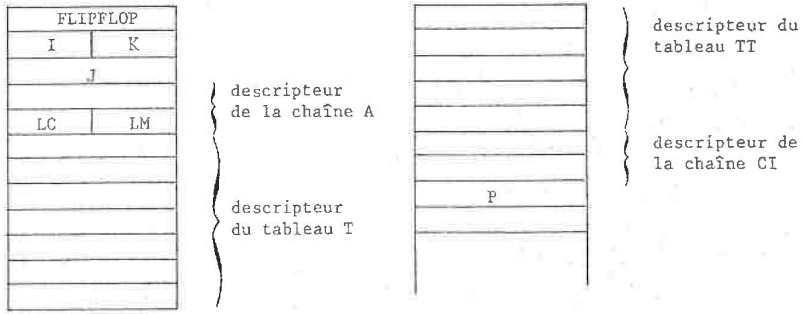
En fonction de ces renseignements, on va transformer dans le texte $\&i_2$ le rang en adresse relative.

```
ALLC (0,1) ET                    ALLC (0,1) ET
ALLC (0,2) FIX BIN (15)          ALLC (0,2) FIX BIN (15)
ALLC (0,3) FIX BIN (31)          ALLC (0,2.5) FIX BIN (15)
ALLC (0,4) FIX BIN (15)          ALLC (0,3) FIX BIN (31)
ALLC (0,5) VAR CH 10 C           ALLC (0,4) VAR CH 10 C
ALLC (0,6) TAB 3                 ALLC (0,6) TAB 3
DIM E 1 C, 5 C                   DIM E 1 C, 5 C
DIM E - 1 C, 3 C                 DIM E - 1 C, 3 C
DIM E 2 C, 6 C                   DIM E 2 C, 6 C
FLOAT BIN (24)                   FLOAT BIN (24)
——                               ——
——                               ——
ROUTINE (0,7) 0                  ROUTINE (0,7) 0
ALLC (1,1) TAB 2                 ALLC (1,1) TAB 2
DIM E 1 C, * (0,2)               DIM E 1 C,  * (0,2)
DIM E 1 C, * (0,3)               DIM E 1 C,  * (0,3)
FLOAT BIN (24 )                  FLOAT BIN (24)
ALLC (1,2) VAR CH 10 C           ALLC (1,6) VAR CH 10 C
```

```
    ALLC (1,3) A                        ALLC (1,8) A
    ALLC (1,4) A                        ALLC (1,9) A
    :=A  *(1,4), (1,3)                  :=A  *(1,9),*(1,8)
(n) || VAR CH 10 C  *(0,5), *(1,2)      || VAR CH 10 C  *(0,4),  *(1,6)
    := VAR CH 10 C n T, (1,2)           := VAR CH 10 C n T, (1,6)
    ——                                  ——
    END                                 END
(m) + FIX BIN (31)  *(0,3),  *(0,4)     + FIX BIN (31)  *(0,3),  *(0,2.5)
    := FIX BIN (15) m T, (0,2)          := FIX BIN (15) m T, (0,2)
    —
    —
    —
```

Au cours de cette passe on conserve dans une table la correspondance entre l'adresse relative et le rang de chaque donnée et dans les instructions on remplace le rang par l'adresse relative.

Les données locales aux procédures E et F pourront être implantées de la façon suivante :



données locales à Z — données locales à F

Remarque :

Afin d'optimiser l'encombrement des données de la procédure E dans la pile dynamique, on range la variable K qui occupe un demi-mot à la suite de J.

Nous allons maintenant préciser la sémantique des triplets de déclarations "ALLC".

Pour la déclaration d'un mode primitif la sémantique de ces triplets est : réserver dans la pile dynamique la place pour cette variable en fonction de l'encombrement associé au mode primitif. On peut exprimer cela en ℓi_2 en incrémentant le contenu du registre qui contient l'adresse mémoire libre de la taille de la donnée traitée.

Exemple :

On traduit les triplet ALLC (0,3) FIX BIN (31)
par AI AML, 1

Augmenter le contenu du registre AML de une unité puisque la taille d'une donnée de mode FIX BIN (31) est un mot.

Pour une construction tableau, il faut réserver en mémoire la taille du vecteur des renseignements et générer le code ℓi_2 qui permettra de remplir à l'exécution ce vecteur de renseignements.

L'adresse du vecteur de renseignements sera donc le paramètre de différentes MACRO qui permettent de remplir ce vecteur de renseignements ; on va donc ranger cette adresse dans un mot de la mémoire. La macro RES effectue la réservation de la place du vecteur de renseignements et conserve l'adresse de ce vecteur dans

```
$VR
MACRO
RES n, $VR
ALLOUER (RI)
LR RI, AML
S RI, $VR
AI AML, n
LIBERER (RI)
FINMACRO
```

4. LE TRAITEMENT DES PROCEDURES

4.1. La traduction des langages procéduraux : rappel

Activer une procédure c'est théoriquement insérer le corps de la procédure à la place de l'appel et l'exécuter après avoir fait le remplacement des paramètres par les arguments.

La récursivité des procédures dans les langages à structure de blocs implique que le corps de chaque procédure soit compilé en un sous-programme réentrant. Pour cela on crée à chaque appel d'une procédure une zone de travail dans la mémoire organisée en pile dynamique. L'adresse de début de cette zone s'appelle adresse de base de la procédure.

Cette zone de travail contient les données locales à la procédure ainsi que les mémoires de travail qu'elle utilise.

Nous allons maintenant préciser les fonctions de ces mémoires de travail en récapitulant les opérations à effectuer pour le traitement des procédures.

4.1.1. *Adressage dynamique des variables*

Chaque variable est caractérisée par deux numéros qui constituent l'adresse dynamique de la variable.

- np niveau d'imbrication de la procédure où elle a été déclarée.
- ad adresse relative par rapport à la fin des données de liaison de la procédure où elle a été déclarée.

Lorsqu'on trouve dans une procédure une référence à une variable quelconque, il faut pouvoir adresser cette variable qui a pu être déclarée dans un niveau inférieur d'imbrication.

Pour cela à l'exécution doivent être conservées dans un "display" toutes les origines des mémoires locales des niveaux inférieurs au niveau courant.

L'adresse de base de la procédure courante se trouve toujours dans un registre reservé à cet effet qu'on appelle registre local (RL).

Grâce au niveau des procédures, on retrouve dans le display la base de la procédure qu'on recherche. Il suffit d'ajouter la place prise par les données des liaisons et l'adresse relative à cette base pour obtenir l'adresse recherchée.

4.1.2. *Appel de procédure*

- On effectue les opérations de passage des paramètres.
- On effectue un branchement vers l'adresse du corps de la procédure.

4.1.3. *Entrée dans le corps de la procédure*

- On range en pile dynamique l'adresse retour.
- L'adresse de base de la procédure appelante.
- On constitue le nouveau display dans la pile dynamique. Celui-ci est formé du display de la procédure appelante auquel on ajoute l'adresse de la procédure courante.
- On réserve la zone affectée aux variables locales à la procédure.
- On remplit un pointeur qui permet d'effectuer les retours en arrière : ce pointeur a pour valeur l'adresse de la mémoire libre en pile dynamique.
- On positionne le registre local avec la nouvelle adresse de base.

4.1.4. *Sortie du corps de la procédure*

- On restaure le registre local.
- On restaure l'adresse mémoire libre.
- On effectue un branchement vers l'adresse retour dans la procédure appelante.

4.2. Le traitement des procédures en ℓi_2

Nous avons indiqué au § 3 comment on transforme le rang d'une variable en adresse relative, et comment on calcule la longueur de la zone affectée aux variables locales à la procédure (on a appelé L cette longueur).

Nous allons maintenant décrire en ℓi_2 les trois fonctions d'appel de procédure, d'entrée dans le corps de la procédure, de sortie du corps de la procédure.

Dans ce qui précède on constate que la base d'adressage, l'adresse de retour, l'adresse mémoire libre en pile dynamique jouent un rôle fondamental on va donc attribuer à chacune de ces adresses un registre spécifique. Les noms de ces registres sont des mots clés du langage ℓi_2

RL est le registre local : il contient la base d'adressage de la procédure courante.

RLA est le registre local appelant : il contient la base d'adressage de la procédure appelante.

AR est le registre adresse retour.

AML est le registre adresse mémoire libre : il contient l'adresse du premier mot libre en pile dynamique.

L'ordre dans lequel on range ces différentes informations en pile dynamique n'est pas unique, nous proposons une organisation particulière et les macros associées aux trois fonctions précédentes pour cette organisation. Si un auteur de compilateur désire mémoriser ces informations autrement en pile dynamique, il pourra écrire lui-même les nouvelles macros correspondant au traitement de ces trois fonctions.

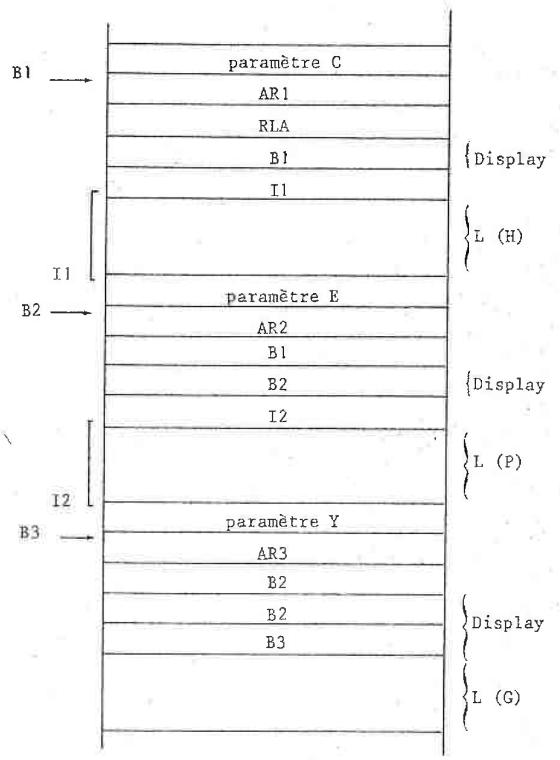
Notre organisation consiste à ranger dans la pile dynamique les informations dans l'ordre suivant quand on entre dans le corps de la procédure.

- adresse retour contenu du registre AR.
- base de la procédure appelante.
- display.
- pointeur de retour.

L'exemple qui suit montre l'état de la pile dynamique au point d'exécution x du programme suivante

```
'begin' 'intéger' a,b,c ;
'procedure' p (x) ;'integer' x ;
     'begin' real y ;
     'procedure' g (u) ;'real' u :
           'begin''real' z ;

        'end' ;
       g (y) ,
  end,
 'procedure' h (t) ; 'integer' t ;
    'begin' 'integer' d,e,f ;
       p (e) ;
    'end' ;
    h (c)
   'end' ;
```

4.2.1. *L'appel d'une procédure*

Rappelons la syntaxe ℓi_2 d'un appel de procédure

<appelproc> = APPEL <descriptif> , [<lequi>] ; FINAPPEL

<lequi> = [<equi>]*

<equi> = <affectation>

Cette suite d'affectations exprime l'équivalence entre le paramètre formel et le paramètre effectif correspondant.

Il nous faut traduire en ℓi_2 cette équivalence : un moyen de le faire est d'attribuer au renseignement concernant chaque paramètre effectif une zone

de mémoire dans la pile dynamique, cette zone de mémoire a pour désignation en ℓi_1, la désignation du paramètre formel.
Ce renseignement peut être la valeur ou l'adresse du paramètre effectif suivant l'indication du descripteur de mode qui figure dans l'équivalence.
Exemple 1 :
: = A (np_i, ad_i), (np_f, ad_f,1)
Cette affectation exprime qu'on va mémoriser l'adresse qui correspond à la désignation npi, adi dans la zone du paramètre effectif (ce cas provient de la traduction d'un passage par référence d'une procédure PLI ou du traitement d'un paramètre de mode rep ... en ALGOL 68)
Exemple 2 :
: = E OC (np_n,ad_n , 1)
Cette affectation exprime que la valeur entière 0 doit être mémorisée dans la zone du premier paramètre effectif.
(ce cas provient de la traduction d'un passage par valeur d'une prodédure ALGOL 60, ou du traitement d'un paramètre effectif de mode ent en ALGOL 68).

Pour traduire la première affectation on calcule l'adresse de i par la macro ADRESSE, on change cette adresse immédiate dans un registre de travail et on range le contenu de ce registre dans l'adresse mémoire libre.

Pour traduire la deuxième affectation en ℓi_2 , on change la valeur immédiate 0 dans un registre de travail et on range le contenu de ce registre dans l'adresse mémoire.

Quand on a fini de ranger dans la pile les informations concernant les paramètres effectifs, la nouvelle adresse mémoire.

On génère alors un branchement vers l'adresse dans le code ℓi_2 du corps de la procédure appelée (en mémorisant le registre AR l'adresse du retour)
Remarque :
On range les renseignements concernant les paramètres effectifs dans l'ordre inverse de la liste des affectations de l'appel. Nous verrons pourquoi au paragraphe 4.2.4.

4.2.2. *L'entrée dans le corps d'une procédure*

A l'entrée dans le corps d'une procédure il faut ranger dans la pile dynamique les données de liaison, réserver la place du résultat s'il existe et réserver la place des données locales à la procédure.

Pour effectuer ces opérations on appelle la macro CORPSPROC pour une procédure, la macro CORPSFONC pour une fonction.


```
MACRO
CORPSPROC np, l
ALLOUER (R1)            /* mémoriser l'adresse mémoire libre qui sera
                           la nouvelle base */

LR R1, AML
S AR, 0 (0,AML)         /* ranger l'adresse retour dans la pile */
AI AML, 1
S RL, 0 (0, AML)        /* ranger la base de la procédure appelante
                           dans la pile */
LR RL, R1               /* positionner la nouvelle base */
AI AML, l
DISPLAY np - 2          /* ranger le display dans la pile dynamique */
AI AML, 1               /* réserver la place du pointeur */
LR R1, AML              /* mémoriser l'adresse du pointeur */
AI AML, 1               /* réserver la place des variables locales */
S AML, 0 (0,R1)         /* remplir la zone pointeur avec l'adresse
                           mémoire libre */

LIBERER (R1)
FINMACRO

MACRO
CORPSFONC np, l, r      /*   longueur des variables locales */
ALLOUER (R1)            /* r longueur du résultat */
L R1, AML               /* mémoriser l'adresse mémoire libre qui sera
                           la nouvelle base */

S AR, 0 (0, AML)        /* ranger l'adresse retour dans la pile */
AI AML, 1
S RL, 0 (0, AML)        /* ranger la base de la procédure appelante dans
                           la pile */
LR RL, R1               /* positionner la nouvelle base */
AI AML, 1
DISPLAY np - 2          /* ranger le display */
AI AML, 1               /* réserver la place du pointeur */
LR R1, AML              /* mémoriser l'adresse du pointeur */
AI AML, r + 1           /* réserver la place du résultat et des variables
                           locales */

S AML, 0 (R1)           /* remplir la zone pointeur avec l'adresse
                           mémoire libre */
```

```
LIBERER (R1)
FINMACRO


MACRO
DISPLAY np2              /* la macro DISPLAY est appelée avec le paramètre
ALLOUER (R1)                effectif np - 2 */
ALLOUER INDEX (X1)
LI R1, np2
BN $ + 7                 /* si np   2 on recopie le display de la
BOUCLE j = 0, np2           procédure appelante */
LIX X1, j
L R1, 0 (X1, RLA)
L AML, R1
AI AML, 1
FINBOUCLE
L AML, RLA               /* on ajoute la nouvelle base locale */
AI AML, 1
LIBERER INDEX (X1)
LIBERER (R1)
FINMACRO
```

4.2.3. *Sortie du corps d'une procédure*

A la sortie du corps d'une procédure il faut sauver le résultat s'il existe, restaurer l'adresse mémoire libre, restaurer le registre local, générer un branchement vers l'adresse retour.

Pour effectuer ces opérations, on appelle la macro FINPROC pour une procédure, la macro CORPS FONC pour une fonction.

```
MACRO
FINPROC npp              /*  la macro FINPROC est appelée avec le paramètre
ALLOUER (R1)                 np + 2 */
L R1, np + 2 (0, RLA)
LR AML, R1               /*  restaurer l'adresse mémoire libre */
LR RL, RLA               /*  restaurer le registre local */
B*(0, AML)               /*  générer branchement vers l'adresse retour */
```



```
LIBERER (R1)
FINMACRO

MACRO
FINFONC npp, r           / * la macro FINFONC est appelée avec le paramètre
ALLOUER (X1)                 np + 2 */
ALLOUER (r, R)
BOUCLE j = 1, r
LIX X1, j                /*  ranger le résultat dans une suite de registre */
L Rj, np + 2 (X1, Rj)
FINBOUCLE
L R1, np + 2 (0, RLA)    /*  restaurer l'adresse mémoire libre */
LR AML, R1
LR RL, RLA               /*  restaurer le registre local */
B  *(0, AML)             /*  générer branchement vers l'adresse retour */
LIBERER INDEX (X1)
LIBERER (r, R)
FINMACRO
```

4.2.4. *L'adressage des paramètres formels dans le corps d'une procédure*

Seuls les paramètres formels déclarés passés par valeur d'ALGOL 60, possèdent une mémoire réservée dans le corps de la procédure, ils sont donc désignés en ℓi_1 par un descriptif comme n'importe quelle donnée locale. Pour obtenir leur adresse en ℓi_2 , on appelle donc la macro ADRESSE.

Les autres paramètres formels sont désignés en ℓi_1 par un triplet de la forme (np, ad, n) où (np, ad) est le descriptif de la procédure, et n est le rang du paramètre formel considéré.
Lorsque dans le code ℓi_1 du corps de la procédure, on trouve la désignation d'un paramètre formel, il faut remplacer cette désignation par le renseignement concernant le paramètre effectif. Or nous avons convenu de placer les renseignements concernant les paramètres effectifs dans la zone située juste avant les données de liaison.

Nous avons vu que l'on rangeait les renseignements concernant les paramètres effectifs dans l'ordre inverse de leur apparition dans le texte ℓi_1 ; par conséquent le renseignement concernant le premier paramètre effectif se trouve placé juste avant l'adresse de base.

On traduit donc (np, ad, 1) par l'adresse (RL) - 1
*(np, ad, 1) par ((RL) - 1)
**(np, ad, 1) par ((RL) - 1)

Nous allons reprendre les exemples traités dans le chapitre 6 de la première partie, en explicitant le code ℓi_2 généré pour chacun d'eux.

Traduction de l'exemple du paragraphe 6.1

```
(1) ROUTINE (np_c, ad_c) 2                INS C CORPSFONC np, 0, I
(2) ALLC (np_c, ad_c, 0) E                      ALLOUER (R1)
(3) + E *(np_c, ad_c, 1), *(np_c, ad_c, 2)      L R1, - 1 (0, RL)
(4) x E *(np_c, ad_c, 1), 3 T                   AM R1, - 2 (0, RL)
(5) := E 4 T, (np_c, ad_c, 0)                   MM R1, - 1 (0, RL)
(6) END                                         S R1, np + 2 (0, RL)
                                                LIBERER (R1)
                                                FINFONC np + 2, 1
(7) ALLC (np_d, ad_d) A                         ALLOUER (R1)
(8) := Et (np_c, ad_c), (np_d, ad_d)            LI R1 INSC
                                                ADRESSE np, npd, add, AD
                                                S R1, AD
(n) APPEL *(np_d, ad_d) 2                       LIBERER (R1)
(n + 1) := E 3 C, (*(np_d, ad_d),1)             ALLOUER (R1)
(n + 2) := E 2 C, (*(np_d, ad_d),2)             LI R1, 2
(n + 3) FINAPPEL                                S R1, 0 (0, AML)
                                                AI AML, 1
                                                LI R1, 3
                                                S R2, 0 (0, AML)
                                                AI AML, I
                                                L R1, AD
                                                BAL AR, R1
```

Le triplet (3) exprime qu'il faut ajouter les valeurs des paramètres formels, on change dans un registre le contenu de la zone contenant le renseignement concernant le premier paramètre effectif : L R1, - 1 (0, RL) et on ajoute au contenu de ce registre le contenu de la mémoire d'adresse (RL) - 2 : AM, R1, - 2 (0, RL)

Le triplet (4) est traité de façon semblable. Le résultat du triplet (4) se trouve dans le registre R1.

La sémantique du triplet (5) est : ranger la valeur du triplet (4) dans le résultat. En ℓi_2 on range le contenu du registre R1 dans la zone réservée au résultat dans la zone dynamique np + 2 (0, RL).

La sémantique du triplet (8) est : ranger dans la zone de désignation (np_d, ad_d) l'adresse dans le code de l'objet (np_c, ad_c). Cette adresse de code est l'étiquette du corps de la routine C : INS C. Pour traduire le triplet (8) on génère donc un chargement immédiat de l'étiquette INS C.

Au triplet (n) on appelle la procédure de désignation * (np_d, ad_d). L'adresse ℓi_2 correspondant à la désignation (np_d, ad_d) est calculée par la macro ADRESSE qui rend l'adresse AD. L'adresse de branchement est le contenu de AD. L'adresse de retour est conservée dans le registre AR.

La génération correspondante en ℓi_2 est :

```
L R1, AD
BAL AR, R1
```

Traduction de l'exemple du paragraphe 6.2.1.

```
                                          INS h CORPSPROC np, 2, 0
(1) ROUTINE (nph, adh) 2                        ALLOUER (R1)
                                                ADRESSE npx, adx, AX
(2) ALLC (npx, adx) E                           L R1, - 1 (0, RL)
(3) : = E (nph, adh, 1), (npx, adx)             S R1, AX
(4) ALLC (npy, ady) E                           ADRESSE np, npy, ady, AY
(5) : = E (nph, adh, 2), (npy, ady)             L R1, - 2 (0, RL)
                                                S R1, AY
(6) + E 5 C,  *(npx, adx)                       LI R1, 5
                                                AM R1, AX
(7) : = E 6T, (npy, ady)                        S R1, AX
                                                L R1, AY
(8) - E  *(npy, ady), 1 C                       MI R1, 1
(9) : = E 8 T, (npy, ady)                       S R1, AY
                                                FINPROC np + 2
(10) END
——                                              ——
——                                              ——
(n) APPEL (nph, adh)                            LI R1, 1
                                                S R1, 0 (0, AML)
(n + 1) : = E OC, (nph, adh, 1)                 AI AML, 1
(n + 2) : = E 1 C, (nph, adh, 2)                LI R1, 0
(n + 3) FINAPPEL                                S R1, 0 (0, AML)
                                                AI AML, 1
                                                BAL AR, INS h
```

Ci-dessus figure un exemple de traduction où les deux paramètres formels possèdent une zone propre dans le corps de la procédure. Cette zone est initialisée avec les valeurs contenues dans les zones afférant aux renseignements des paramètres effectifs par les triplets (3) et (5).

Pour traduire le triplet (3) par exemple, il suffit de charger le contenu de la zone du renseignement dans un registre général R1 et de ranger le contenu de ce registre dans la zone du paramètre formel dans la procédure. La séquance de code ℓi_2 générée est la suivante :

```
ADRESSE npx, adx, AX
L R1, - 1 (0, RL)
S R1, AX
```

L'instruction de branchement au corps de la procédure s'exprime en ℓi_2 par BAL AR, INSf où INSf est l'étiquette désignant le début du code du corps de la procédure.

Traduction de l'exemple du paragraphe 6.3.

	INS f CORPSPROC np, &
(4) ROUTINE (np_f, ad_f) !	ADRESSE npk, adk, AK
	ALLOUER (R1)
(5) + E 1 C, *(np_k, ad_k)	LI R1, 1
(6) : = E 5 T, (np_k, ad_k)	MM R1, AK
(7) + E 1 C, **(np_f, ad_f, 1)	S R1, AK
(8) : = E 7 T, *(np_f, ad_f, 1)	LI R1, 1
	ALLOUER (R2)
(9) END	L R2, - 1 (0, RL)
	AML R1, 0 (0, R2)
	S R1, 0 (0, R2)
	FINPROC np + 2
	ADRESSE np, npk, adk, AK
(10) : = E 3 C, (np_k, ad_k)	LI R1, 3
(11) : = E 3 C, (np_i, ad_i)	S R1, AK
	ADRESSE np, npi, adi, AI
	LI R1, 3
	S R1, AI
(12) APPEL (np_f, ad_f) !	LI R1, AI
(13) : = A (np_i, ad_i), (np_f, ad_f,1)	S R1, 0 (0, AML)
	AI AML, 1
(14) FINAPPEL	BAL AR, INS f

5. L'ADRESSAGE EN ℓi_2

Après la première passe sur le texte ℓi_2 , chaque donnée est caractérisée par un doublet (np, ad) :

Niveau de procédure, adresse relative par rapport à la fin des données de liaison. Il faut traduire ce doublet en adresse effective dans la pile dynamique.

L'adresse de chaque donnée sera évaluée en appelant la macro adresse qui possède trois paramètres :

- le niveau de procédure np
- l'adresse relative de la donnée dans la procédure ad
- l'adresse effective $A

Pour calculer l'adresse il faut tester si la donnée est locale à la procédure courante (de niveau npc). On compare donc np et npc.

- si np = npc la donnée à adresser est locale à la procédure.
  l'adresse effective de cette donnée est
  $$A = (RL) + np + 2 + ad$$
  RL registre local contient l'adresse de base de la procédure.
- si la donnée n'est pas locale à la procédure, à l'aide du display, on recherche l'adresse de base de la procédure où elle a été déclarée
  $$base = \left[(RL) + np - 2\right]$$
  Si on charge cette valeur dans un registre RNL, l'adresse de cette donnée est (RNL) + np + 2 + ad.

Remarque :

Dans l'implantation standard présentée au paragraphe précédent, nous avons vu que les données de liaison occupent une place égale à (np + 2) mots

```
MACRO
ADRESSE npc, np, ad, $A
ALLOUER (R1, R2)
LI R1, np
LI R2, npc
SR R1, R2
BZE $ + 5
L R2, np - 3 (0, RL)
L R1  *np + ad (0, R2)
S R1 $A
BRU $ + 2
L R1, np + ad (0, RL)
FINMACRO
```

6. DESCRIPTION DE QUELQUES FONCTIONS D'ACCES

6.1. Implantation d'un tableau par la méthode des multiplicateurs

6.1.1. *Déclaration*

Considérons la déclaration du tableau suivant en ALGOL 60

REAL ARRAY T $[a_1 : b_1 , a_2 : b_2 , a_3 : b_3]$

Cette déclaration a été traduite dans une première étape par les triplets suivants de ℓi_1 :

ALLC (npt, rgt) TAB 3

DIM E $*(np_{a1}, rg_{a1})$, $*(np_{b1}, rg_{b1})$

DIM E $*(np_{a2}, rg_{a2})$, $*(np_{b2}, rg_{b2})$

DIM E $*(np_{a3}, rg_{a3})$, $*(np_{b3}, rg_{b3})$

R

Au cours d'une première passe sur le texte ℓi_1 , on transforme le rang en adresse relative. La traduction en ℓi_2 portera donc sur le texte suivant :

ALLC (npt, adt) TAB 3

DIM E $*(np_{a1}, ad_{a1})$, $*(np_{b1}, ad_{b1})$

DIM E $*(np_{a2}, ad_{a2})$, $*(np_{b2}, ad_{b2})$

DIM E $*(np_{a3}, ad_{a3})$, $*(np_{b3}, ad_{b3})$

Supposons qu'on implante le tableau "ligne par ligne". L'adresse d'un élément de tableau quelconque est alors fournie par la formule suivante :

$$Ad\ \left[T\ (I_1\ ,\ I_2\ ,\ I_3\ \ldots\ I_n)\right] = BASE - \sum_{j=1}^{n} a_j\ M_j + \sum_{j=1}^{n} I_j\ M_j$$

avec $\ell i = b_i - a_i + 1$

$$M_j = \prod_{k=j+1}^{n} \ell_k \qquad V_j \in [1, n-1]$$

$Mn = 1$

On remarque que le terme $BASE - \sum_{j=1}^{n} a_j\ M_j$ est indépendant des valeurs des indices de l'élément du tableau, il ne dépend que des bornes. Il en est de même des multiplicateurs Mj. Toutes ces informations sont mémorisées en pile dynamique dans le vecteur de renseignements du tableau qui a pour dimension 2 n + 1 (taille connue à la compilation). Les éléments de ce vecteur de renseignements ne seront remplis qu'à l'exécution, puisque les bornes peuvent être variables.

n	BASE - Σ a_j M_j
a_1	b_1
a_2	b_2
a_3	b_3
$M_1 = \ell_2\ \ell_3$	
$M_2 = \ell_3$	
$M_3 = 1$	

} n mots contenant les bornes inférieures et supérieures

} n multiplicateurs.

Lorsqu'on lit le premier triplet de déclaration d'un tableau

ALLC (np_t , ad_t) TAB n

on doit réserver la place du vecteur de renseignements, c'est-à-dire 2 n + 1 mots à partir de l'adresse Vr calculée à partir des paramètres np_t, ad_t (par appel de la macro adresse).

Pour chaque triplet de description d'une dimension, on génère le code qui permettra de mémoriser à l'exécution les valeurs des bornes inférieure et supérieure. Ce code dépend de la forme syntaxique de la description d'une borne :

- Si la valeur de la borne est décrite par une constante : on génère un appel de la macro BORNEC dont le prototype comporte un chargement immédiat dans un registre d'une valeur immédiate, et un rangement du contenu de ce registre dans le demi-mot ad hoc du vecteur de renseignements.
- Si la valeur de la borne est fournie par la valeur d'une variable : on génère un appel de la macro BORNEV . Son prototype comporte un appel de la macro adresse, le chargement du contenu de ce mot dans un registre et le rangement de ce registre dans le demi-mot ad hoc du vecteur de renseignements.
- Si la valeur de la borne est la valeur d'un triplet : on génère un appel de la macro BORNET . Son prototype comporte un chargement dans un registre du contenu de la mémoire temporaire dans laquelle a été rangée la valeur du triplet et le rangement de ce registre dans le demi-mot du vecteur de renseignements.

```
MACRO
MUL n, $L, $VR
ALLOUER INDEX (X2, X3)
BOUCLE J = 1, n - 1
LI R1, 1
LIX X2, J
BOUCLE k = J, n - 1
LIX X3, k
MM R1, $L (X3)
FINBOUCLE
S R1, $VR (X1)
AIX X1, 1
FINBOUCLE
LI R1, 1

S R1, $VR
LIBERER INDEX (X2, X3)
FINMACRO
```

```
MACRO
ADCONS n, $BASE, $VR
ALLOUER INDEX (X2, X3)
BOUCLE J = 0, n
LIX X2, 2J + 1
LIX X2, J + n + 1
LH R1, VR (X2)
MM R1, VR (X3)
AR R2, R1
FINBOUCLE
COMPL R2
AI R2, $CASE
LIX X2, 1
SH R2, VR (X2)
LIBERER INDEX (X2, X3)
FINMACRO
```

Montrons comment ces macros sont utilisées pour traduire la déclaration suivante du tableau $\&i_1$.

```
ALLC (npt, adt) TAB 3

DIM E 1C,  *(npi, adi)

DIM E 5C,  *(npj, adj)

DIM E  *(npk, adk), 10 C

E
```

```
ADRESSE np, npt, adt, VR
RES 7, VR
ALLOUER INDEX (X1)
ALLOUER (R1)
LIX, X1, 2
BORNEC  1, VR
BORNEV  npi, adi, VR
BORNEC  5, VR
BORNEV  npj, adj, VR
BORNEC  10, VR
LIX X1, 4
ALLOUER MEMTEMP (L, 3)
LONG 3, L, VR
MUL 3, L, VR
PLACE P, L
ALLDYN (BASE, p)
RENDRE MEMTEMP (L,3)
ADCONS BASE, VR
LIBERER INDEX (X1)
LIBERER (R1)
```


Le même registre de travail R1 peut être utilisé pour le traitement de l'ensemble de la déclaration du tableau, il est donc alloué à la traduction du triplet de syntaxe ALLC <descriptif> TAB entier.

Un registre d'index est également alloué et initialisé dans la traduction de ce triplet.

Cet exemple illustre l'application de la structure de blocs pour les registres. Dans la macro LONG on a besoin de deux registres d'index. On va allouer X1 et X2, comme on a plus besoin de ces registres à la fin de la macro on les libère. La portée des registres X1 et X2 est donc égale à la longueur du code $\&i_2$ de la macro LONG. On peut remarquer qu'un registre d'index X1 a déjà été alloué au début de la traduction ; la macro LONG constitue donc un bloc imbriqué et la signification de X1 dans la macro LONG n'est pas la même qu'à l'extérieur de cette macro. A la traduction en langage d'assemblage il faudra donc allouer un véritable registre d'index différent.

Les éléments effectifs du tableau seront implantés à l'exécution à partir de l'adresse BASE, l'ensombrement total du tableau est p mots.

Cet encombrement est calculé par la macro PLACE en fonction des longueurs des différentes dimensions et de l'encombrement de chaque variable indicée.

Lorsqu'on a fini de reconnaître tous les triplets de la déclaration d'un tableau, on génère un appel de la macro LONG.

Cette macro mémorise les longueurs des différentes dimensions dans un tableau temporaire.

On génère ensuite un appel de la macro MUL qui calcule les multiplicateurs et les range dans le vecteur de renseignements. Enfin on génère un appel de la macro ADCONS qui range la partie constante de l'adresse d'une variable indicée dans le premier mot du vecteur de renseignements.

Les différentes macros du traitement de la déclaration d'un tableau figurent ci-dessous :

```
MACRO
BORNEC v , $VR
LI R1, v
SH R1, $VR (X1)
AIX X1, 1
FINMACRO
```

```
MACRO
BORNEV  np, npb, cdb, $VR
ADRESSE np, npb, adb, AB
L R1, AB
SH R1, $VR (X1)
AIX X1, 1
FINMACRO
```

```
MACRO
BORNET  $EXP, $VR
L R1, $EXP
SH R1, SVR (X1)
FINMACRO
```

```
MACRO
LONG n, $L, $VR
ALLOUER INDEX (X1, X2)
LIX X1, 2
BOUCLE J = 1, n
LIX X2, J
LH R1, $VR (X1)
SH R1, $L (X2)
FINBOUCLE
LIBERER INDEX (X1, X2)
```

6.1.2. *Traitement de la référence à une variable indicée*

Rappelons la formule de calcul de l'adresse d'une variable indicée citée précédemment.

$$\text{Adresse}\ \left[T(I_1, I_2 \ldots I_n)\right] = \underbrace{\text{BASE} - \Sigma a_j M_j}_{\text{Adresse continue dans le 2° demi-mot de VR}} + \sum_{i=1}^{n} I_j M_j$$

Pour calculer l'adresse d'une variable indicée, il faut donc multiplier terme à terme la valeur d'un indice par la valeur du multiplicateur de même rang. Or la valeur de l'indice peut être décrite sous plusieurs formes syntaxiques en ℓi_1 . Rappelons en effet la syntaxe de description d'une variable indicée en ℓi_1 .

<adresse variable indicée> =
ORIGINE<origine> TAB entier, <indice> { <indice> ;} * <déclareur effectif>
<origine> = <descriptif> | entier T
<indice> = IND E <valeur entière>
<valeur entière> = <constante entière> | * <descriptif> | entier T

A chacune des dérivations de <valeur entière> on associe en ℓi_2 une macro différente : INDC correspond à <constante entière>
INDV correspond à *<descriptif>
INDT correspond à entier T

```
MACRO
INDC  vi, $VR
LI R2, vi
```

```
MACRO
INDV  npc, np, ad, $VR
ADRESSE npc, np, ad, AI
```



```
MM R2, VR (X1)
AR R1, R2
AIX X1, 1
FINMACRO
```

```
L R2, AI
MM R2, VR (X1)
AR R1, R2
AIX X1, 1
FINMACRO
```

```
MACRO
INDT  $TR, $VR
L R2, $TR
MM R2, VR (X1)
AR R1, R2
AIX X1, 1
FINMACRO
```

Pour calculer l'adresse de la variable indicée t (i, 2, α + β), on génère en ℓi_2 le code suivant :

```
ORIG (npt, adt) TAB 3
```

```
ADRESSE np, npt, adt, VR
ALLOUER (R1, R2)
ALLOUER (X1)
LIX X1, 2
L R1, VR (X1)
LIX X1, N
```

On recherche dans le vecteur de renseignements le terme constant de l'adresse et on le charge dans R1

```
IND E  *(npi, adi)
IND E 2 C
IND E n T
```

```
INDV npi, adi, VR
INDC  2, VR
INDT  TR, VR
```

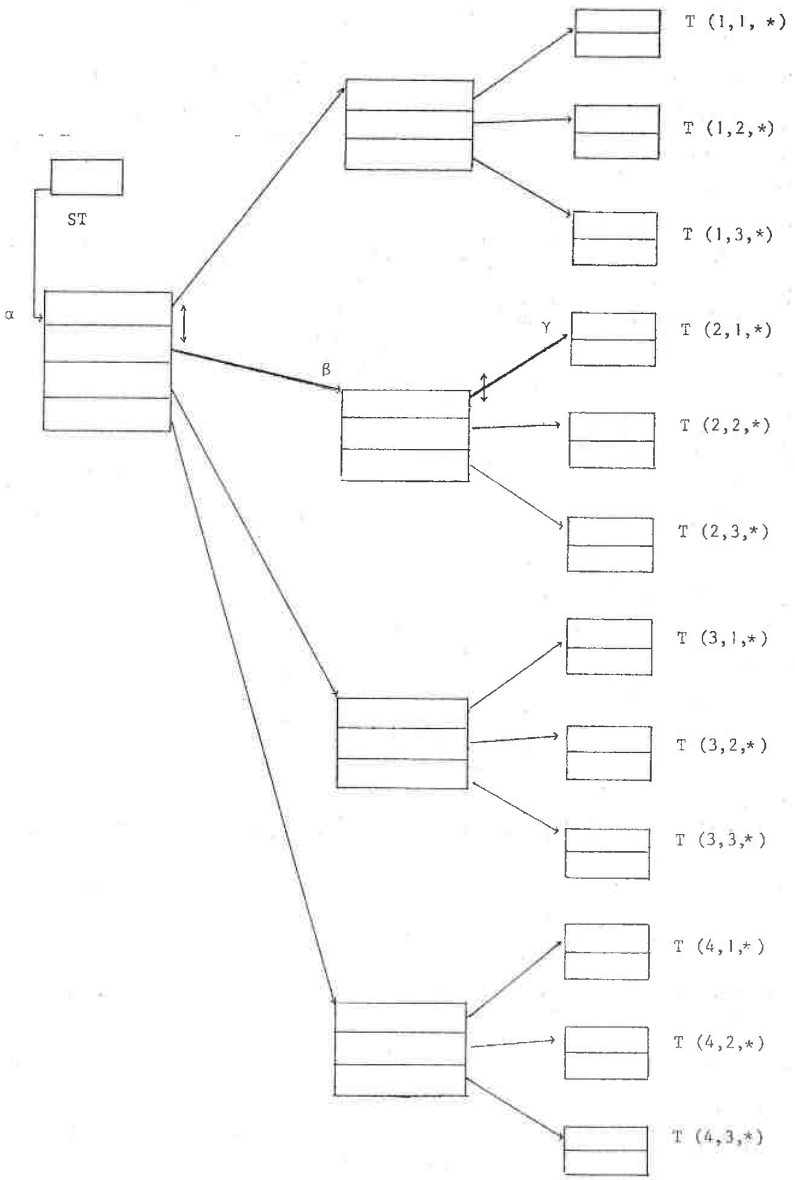
En ce point du programme ℓi_2 l'adresse de la variable indicée se trouve dans R1

6.2. Implantation d'un tableau par la méthode des pointeurs

6.2.1 *Déclaration*

Un tableau est décomposé en un ensemble de sous tableaux où un seul indice seulement varie. On accède à chaque sous tableau par une chaine de tableaux de pointeurs à partir du vecteur de renseignements placé en zone statique. Dans ce cas le vecteur de renseignements d'un tableau est une adresse.

La figure ci-dessous montre la représentation du tableau T [1 : 4, 1 : 3, 1 : 2] et le schéma d'accès à la variable indicée T [2, 1, 2] est représenté en traits forts.



Pour générer en $\&i_2$ l'implantation d'un tableau par cette méthode, il faut effectuer les opérations suivantes :

- calculer l'adresse du vecteur de renseignements.
- générer les séquences de calcul des longueurs de chaque dimension et mémoriser ces longueurs dans un tableau temporaire.
- générer la réservation dynamique de tableaux des pointeurs dont la longueur correspond successivement à la longueur de chaque dimension.

Pour un tableau à n dimensions, on générera n - 1 boucles de réservations dynamiques.

Le calcul de la longueur de chaque dimension est effectué par une macro. Il existe 9 macros de calcul de la longueur d'une dimension correspondant aux combinaisons 2 à 2 des trois formes syntaxiques de description de la valeur d'un indice en $\&i_1$.

Ci-dessous se trouvent les prototypes de ces 9 macros.

1 La borne inférieure et la borne supérieure sont des constantes.

```
MACRO
LCC bi, bs, $L
LI R1, bs
SI R1, bi
S R1, $L (X1)
AIX X1, 1
FINMACRO
```

2 La borne inférieure a une valeur constante, la borne supérieure est une variable

```
MACRO
LCV bi, npc, npbs, adbs, $L
ADRESSE npbs, adbs, ABS
L R1, ABS
SI R1, bi
S R1, $L (X1)
AIX X1, 1
FINMACRO
```

3 La borne inférieure a une valeur constante, la borne supérieure est un triplet

```
MACRO
LCT bi, $TS, $L
L R1, $TS
SI R1, bi
S R1, $L (X1)
AIX X1, 1
FINMACRO
```

4 La borne inférieure est une variable, la borne supérieure est une constante.

```
MACRO
LVC bs, np, npbi, adbi, $L
LI R1, bs
ADRESSE npbi, adbi, ABI
SI R1, ABI
S R1, $L (X1)
AIX X1, 1
FINMACRO
```

5 La borne inférieure est une variable, la borne supérieure est une variable.

```
MACRO
LVV np,npbi, abi, npbs, abs, $L
ADRESSE np, npbi, abi, ABI
ADRESSE np, npbs, abs, ABS
L R1, ABS
SUB R1, ABI
S R1, $L (X1)
AIX X1, 1
FINMACRO
```

6 La borne inférieure est une variable, la borne supérieure est un triplet.

```
MACRO
LVT np, npbi, adbi, $TS, $L
ADRESSE np, npbi, adbi, ABI
L R1, $TS
SUB R1, ABI
S R1, $L (X1)
AIX X1, 1
FINMACRO
```

7 La borne inférieure est un triplet, la borne supérieure est une constante.

```
MACRO
LTC bs, $TI, $L
LI R1, bs
SUB R1, $TI
S R1, $L (X1)
AIX X1, 1
FINMACRO
```


8 La borne inférieure est un triplet, la borne supérieure est une variable.

```
MACRO
LTV np, npbs, adbs, $TI, $TL
ADRESSE np, npbs, adbs, ABS
L R1, ABS
SUB R1, $TI
S R1, $L (X1)
AIX X1, 1
FINMACRO
```

9 La borne inférieure est un triplet, la borne supérieure est un triplet.

```
MACRO
LTT $TI, $TS, $L
L R1, $TS
SUB R1, $TI
S R1, $L (X1)
AIX X1, 1
FINMACRO
```

Montrons comment on peut traduire la même déclaration de tableau que dans l'exemple 6.1.1.

```
ALLC (np_t , ad_t) 3                ADRESSE np, npt, adt, ST
                                    ALLOUER MEMTEMP (L, 3)
DIM E 1 C,  *(np_i , ad_i)          LCV 1, npc, npi, adi, AI
DIM E 5 C,  *(np_j , ad_j)          LCV 5, npc, npj, adj, AJ
DIM E *(np_k , ad_k), 10 C          LCV, 10, npc, npk, adk, AK
E                                   LONGUEUR 3, L
                                    ALLOUER (R1)
                                    ALLOUER (X1, X2)
                                    LR R1, AML
                                    S R1, ST
                                    ALLDYN L (1) MOT,  *ST
                                    BOUCLE i = 1, L (1)
                                    ALLDYN L (2) MOT, DYN (i)
                                    LIX X1, i
                                    L R1, DYN (i)
                                    S R1,  *ST (X1)
                                    BOUCLE j = 1, L (2)
                                    ALLDYN L (3) MOT, DYN (i,j)
                                    LIX X2, j - 1
                                    LR R1, DYN (2, j)
```

```
                                S R1, DYN (1) (X2)
                                FINBOUCLE
                                FINBOUCLE
                                LIBERER (R1)
                                LIBERER (X1, X2)
```

6.2.1. *Traitement de la référence à une variable indicée*

Le premier triplet de description d'une variable indicée permet de calculer l'adresse du vecteur de renseignements. La valeur du 1er indice permet l'accès au tableau d'adressage de l'ensemble des sous-tableaux où la valeur du premier indice est fixée.

La valeur du 2ème indice permet l'accès au tableau d'adressage de l'ensemble des sous-tableaux pour lesquels les valeurs des deux premiers indices sont fixées...

```
(k - 3) ORIG (npt, adt) TAB 3      ADRESSE np, npt, ST
                                   ALLOUER (R1, R2)
                                   ALLOUER (X1)
(k - 2) IND E *(npi, adi)          PTV npi, adi
                                   L R1, *ST (X1)
(k - 1) IND E 2 C                  PTC, 1
                                   L R2, 0 (X1, R1)
(k)     IND E n T                  PTT $T
                                   ADRESSE np, npa, ada, AA
(k + 1) : = E *(npa, ada), kT      L R1, AA
                                   S R1, 0 (X1, R2)
                                   LIBERER (R1, R2)
                                   LIBERER (X1)
```

La séquence de calcul de l'adresse d'une variable indicée nécessite de disposer de deux registres généraux R1, R2 et d'un registre d'index X1.

Le registre d'index X1 est chargé dans une macro différente suivant la syntaxe de description de la valeur de l'indice.

On accède au 1er sous-tableau à partir du vecteur de renseignements, on génère donc une séquence d'initialisation particulière L R1, * ST (X1)

Pour accéder au sous-tableau suivant, on génère

```
L R2, 0 (X1, R1)
```

pour un indice de rang pair

```
L R1, 0 (X1, R2)
```

pour un indice de rang impair

Le traitement du dernier indice diffère suivant que l'on a besoin de l'adresse de la valeur indicée ou de la valeur

```
MACRO                              MACRO
PTV np, npi, adi                   PTC c
ADRESSE np, npi, adi, AI           LIX X1, c
LX X1, AI                          FINMACRO
SUBIX X1, 1
FINMACRO

MACRO
PTT $T
LX X1, $T
SUBIX X1, 1
FINMACRO
```

La macro PTV correspond au cas où la valeur de l'indice est la valeur d'une variable.

La macro PTC correspond au cas où la valeur de l'indice est une constante entière.

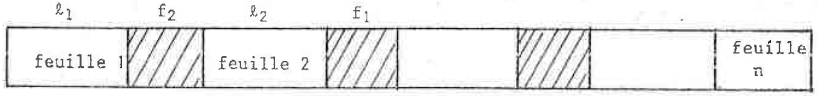
La macro PTT correspond au cas où la valeur de l'indice est la valeur d'un triplet.

6.3. Description de l'adressage d'une structure

6.3.1. *Principe*

Une structure possède en ℓi_2 un descriptif, comme tout autre objet. Dans la deuxième traduction, on fait correspondre à ce descriptif une adresse dans la pile dynamique en appelant la macro ADRESSE. On implante les éléments de la structure à partir de cette adresse.

L'encombrement total de la structure dépend de l'algorithme d'alignement choisi pour les feuilles et les sous structures. Dans tous les cas de configuration en mémoire peut être représentée par le schéma suivant :



Les champs de longueurs f_2, f_3 ... fn sont des éléments de remplissage introduits pour que les éléments feuille 1, feuille 2, ... feuille n aient des alignements corrects.

L'alignement d'un objet est un attribut destiné à indiquer qu'on veut accéder à un objet de la façon la plus rapide quitte à perdre sur l'encombrement en mémoire de la donnée.

Selon la configuration des mots de la machine cible, il existe des facteurs d'alignement variables suivant les données.

Par exemple sur l'IBM 360, l'IRIS 80 CII :

- un nombre réel double précision doit être implanté sur une frontière de double-mot : on dit que le facteur de cadrage est a = 64.
- un nombre entier doit être implanté sur une frontière de mot : on dit que le facteur de cadrage est a = 32.
- la plus petite unité adressable est l'octet, par conséquent, le facteur de cadrage pour les chaînes est a = 8.

Seules les chaînes de bits de longueur constante, peuvent avoir un facteur de cadrage égal à 1 si elles ne sont pas alignées.

A titre d'exemple, nous allons expliciter l'algorithme utilisé dans le compilateur PL/1 de la CII.

La méthode choisie consiste à implanter les données d'une structure sous-structure par sous-structure , de façon à ce que des sous-structures semblables aient des représentations en mémoire identique.

Par conséquent on implante le premier élément d'une sous-structure avec le facteur de cadrage maximum des feuilles de cette sous-structure (et non le facteur de cadrage de l'élément lui-même).

La longueur et le facteur de cadrage de chaque feuille i étant connus, les facteurs de remplissage f_i peuvent être calculés par l'algorithme suivant :

$r = \ell_1$	avec
DO i = 1, n	f (x, y) = 0
f_i = mod (- r, a_i)	si le reste de la division x/y est nul
$r = r + f_i + \ell_i$	f (x, y) = y - reste
END	
$\ell = r - d$	si le reste de la division y/x n'est pas nul

Soit la structure de PL/1 suivante et sa traduction en ℓi_1

DCL 1 S		
2 A	CHAR (5) UNALIGNED	(1) ALLC (np_s, ad_s) STRUCTURE (1,20)
2 B	BIN FIXED ALIGNED	(2) (2,2) COMPRESSE FIX CH 3 C
2 C	BIT (12) UNALIGNED	(3) (3,3) BIN FIXED
2 D	BIN FLOAT (53) ALIGNED	(4) (4,4) COMPRESSE FIX BIT 12 C
2 E	DEC FIXED (5) ALIGNED	(5) (5,5) BIN FLOAT (53)
2 F(3)	CHAR (1) UNALIGNED	(6) (6,6) DEC FIXED (5)
2 G	BIT (12) ALIGNED	(7) (7,9) TAB 1
2 S1		(8) DIM E 1C, 3G
3 H	BIN FIXED ALIGNED	(9) FIX CH 1 C
3 I	CHAR (2) UNALIGNED	(10) (10,10) FIX BIT 12 C
2 S2		(11) (11, 13) STRUCTURE (11, 13)
3 J	BIN FLOAT (53) ALIGNED	(12) (12, 12) BIN FIXED
3 K	BIN FIXED ALIGNED	(13) (13, 13) FIX CH 2 C
2 S3		(14) (14, 17) STRUCTURE (14, 17)
3 L	BIT (20) UNALIGNED	(15) (15, 15) BIN FLOAT (53)
3 M	BIT (8) UNALIGNED	(16) (16, 16) BIN FIXED
3 N	CHAR (2) UNALIGNED	(17) (17, 20) STRUCTURE (17, 20)
		(18) (18, 18) COMPRESSE FIX BIT 20 C
		(19) (19, 19) COMPRESSE FIX BIT 8 C
		(20) (20, 20) COMPRESSE FIX CH 2 C

Sur l'IRIS 80 les données arithmétiques déclarées de mode BIN FIXED sont implantées sur un mot et les données déclarées de mode BIN FLOAT (53) sont implantées sur un double mot.

L'application de l'algorithme précédent permet de calculer les facteurs de remplissage rassemblés dans le tableau suivant :

Nom qualifié	Facteur de cadrage	Facteur de remplissage	Longueur (en bits)
S.A	8	0	40
S.B	32	24	32
S.C	1	0	12
S.D	64	0	64
S.E	32	0	32
S F (3)	8	0	24
S G	32	8	12
S_1 .H	32	20	32
S_1 .I	8	0	16
S_2 .J	64	48	64
S_2 .K	32	0	32
S_3 .L	1	0	20
S_3 .M	1	0	8
S_3 .N	8	4	16

6.3.2. *La traduction en ℓi_2*

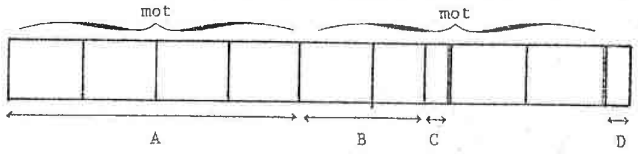
Ce long exemple était destiné à illustrer en algorithme d'implantation en mémoire des structures. Afin d'éviter d'alourdir l'exposé nous allons prendre un exemple plus simple pour montrer comment on effectue la traduction en ℓi_2 .

Soit la déclaration suivante de structure de PL/1, sa traduction en ℓi_1 , et la traduction d'une affectation d'une feuille.

```
DCL 1 Z                          (1) ALLC (npz, rgz) STRUCTURE (1,5)
    2 A   BIN FIXED              (2) (2,2) BIN FIXED
    2 B   BIT (12) UNALIGNED     (3) (3,3) COMPRESSE FIX BIT 12 C
    2 C   BIT ( 3) UNALIGNED     (4) (4,4) COMPRESSE FIX BIT 3 C
    2 D   BIT ( 2) ALIGNED       (5) (5,5) FIX BIT 2 C


Z.C = '101'B                     (n) : = L '101' B C, ((npz,rgz ), (4,4))
```

Conformément aux mécanismes d'implantation précédents, cette structure possède dans la mémoire de l'IRIS 80 la représentation suivante



Les barres verticales figurent les frontières d'octet.

L'implantation de chaque feuille peut être caractérisée par deux doublets.

- un doublet caractérisant l'adresse relative de la feuille par rapport au début de la structure

  D = (DO, DB)

  DO est la partie du déplacement qui peut s'exprimer en octets.

  DB est le reste qui s'exprime en nombre de bits.

- un doublet caractérisant la longueur de la feuille

  L = (LO, LB)

Pour les feuilles de la structure précédente les valeurs des doublets figurent dans le tableau ci-dessous :

	D	L
(Z.A)	(0,0)	(4,0)
(Z.B)	(4,0)	(1,4)
(Z.C)	(5,4)	(0,3)
(Z.D)	(8,0)	(0,2)

Nous avons déjà indiqué au paragraphe 3 qu'au cours d'une première passe sur le texte ℓi_1 , on transformait les rangs des descriptifs en adresses relatives. La correspondance entre rang et adresse relative est conservée dans une table pour être utilisée dans les références aux données.

Pour une structure il faudra conserver la liste des doublets de déplacement et de longueur, calculer l'encombrement total de la structure. Finalement le texte ℓi_2 généré est :

```
ADRESSE npc, npz, adz, AZ
RES 3, AZ
```

```
ALLOUER (R1)
ALLOUER INDEX (X1)
LI R1 '101' B
LIX X1 5
SB R1 AZ, X1 POSIT (5)
```

Il faut ranger le contenu du registre R1 à partir du 5ème bit après l'adresse AZ + (X1).

7 LE TRAITEMENT DES EXPRESSIONS

Les expressions arithmétiques ont été décomposées en triplets au cours de la première étape de traduction. Le passage des triplets aux instructions arithmétiques du langage d'assemblage est aisé. La seule difficulté qu'il présente est la gestion des registres. Or, Sethi et Ullman ont démontré un algorithme qui permet d'évaluer le nombre minimal de registres exigé par une expression arithmétique à partir de son arbre syntaxique et qui permet de trouver l'ordre de génération de la forme intermédiaire correspondante.

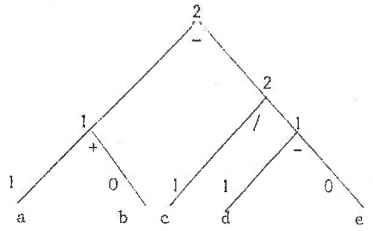
La méthode de traitement des expressions arithmétiques consiste :

- à appliquer l'algorithme de Sethi Ullman [10] à l'arbre syntaxique de l'expression. On génère les triplets d'opération arithmétique de telle sorte que l'opérande à charger dans un registre soit toujours opérande gauche du triplet.
- dans la deuxième traduction, on alloue un registre à chaque valeur de variable qui se trouve être en opérande gauche d'un triplet.

  Dans le cas général, un triplet n'est utilisé qu'une seule fois (c'est-à-dire qu'il n'apparait qu'une seule fois comme opérande d'un autre triplet).

  Mais si on a détecté une sous-expression commune redondante, un triplet pourra être utilisé deux fois ou plus. Le résultat d'un triplet est contenu dans un registre, lorsqu'on trouve dans le texte $\&i_1$ la dernière utilisation de ce triplet en opérande droit, on peut libérer le registre qui le contient.

Nous allons suivre le traitement de l'expression : (a + b) - c / (d - e), au cours des différentes étapes de la traduction.



Rappelons comment on attribue les labels aux feuilles et aux noeuds de l'arbre :

- à une feuille descendant gauche, on attribue le label 1
- à une feuille descendant droit, on attribue le label 0
- soit un noeud j de l'arbre k et l les descendants ayant pour label l_k et l_l

$$l_j = \text{Max}(l_k, l_l) \quad \text{si } l_k \neq l_l$$
$$l_j = l_k + 1 \quad \text{si } l_k = l_l$$

Pour calculer l'expression ci-dessus, il faut au maximum deux registres différents et on génère d'abord le code intermédiaire de la branche qui nécessite le plus de registres.

Le code $\&i_1$ généré est le suivant :

(1) - E $*(np_d, ad_d)$, $*(np_e, ad_e)$
(2) / E $*(np_c, ad_c)$, 1 T
(3) + E $*(np_a, ad_a)$, $*(np_b, ad_b)$
(4) - E 3 T, 2 T

Pour la traduction en $\&i_2$, on demande d'allouer un registre chaque fois que la valeur d'une variable se trouve en opérande gauche d'un triplet. L'adresse de chaque variable est calculée par un appel de la macro ADRESSE.

```
ADRESSE np, npd, add, AD
ADRESSE np, npe, ade, AE
ALLOUER (R1)
L R1, AD
SUB R1, AE
ADRESSE np, npc, adc, AC
ALLOUER (R2)
L R2, AC
DR R2, R1
LIBERER (R1)
ALLOUER (R1)
ADRESSE ne, npa, ada, AA
ADRESSE np, npb, adb, AB
L R1, AA
AM R1, AB
SUB R1, R2
LIBERER (R2)
```

Le registre R1 ne peut être libérer tant qu'on n'a pas trouvé d'utilisation du triplet (4). Si le triplet qui suit est par exemple

:= E 4 T, (np_x, ad_x)

on le traduit en $\&i_2$ par un rangement du contenu du registre R1 dans la mémoire correspondant à la désignation (np_x, ad_x). La séquence suivante d'instructions $\&i_2$ est générée.

```
ADRESSE np, npx, adx, AX
S R1, AX
LIBERER (R1)
```

BIBLIOGRAPHIE

[1] R.J. ORGASS, W.H. WAITE
A base for a Mobile Programming System.
CACM vol 12 n° 9 p 507 (1969)

[2] W.H. WAITE
Bulding a Mobile Programming System.
Report n° 69-2 Computing Center, Université de Colorado Juin 1969

[3] J. STRONG et al
The Problem of Programming Communication with changing machines :
a proposed solution
CACM vol 1 n° 8 p 12 (1958)

[4] T.B. STEEL
A first version of UNCOL
Proceedings AFIPS WJCC n° 19 p 371-378 (1961)

[5] S.S. COLEMAN, P.C. POOLE, W.H. WAITE
The Mobile Programming System. Janus
National Technical Information Center PB 22 0322,
US Department of Commerce, Springfield Va., (1973)

[6] DELTA Système de Description de Langages et Traducteurs par Attributs
B. LORHO
Thèse de Doctorat
De la définition à la traduction des langages de programmation :
méthode des attributs sémantiques (29 novembre 1974)

[7] D.E. KNUTH
Semantics of context free languages
a - Math Systems Theory vol 2 n° 2 p 127-145 (1968)
b - vol 5 n° 1 p 95-96 (1971)

[8] J.C. BEATTY
Register assignment algorithm for generation of highly optimized object code
IBM J Res Develp. (Janvier 1974)

[9] D. GRIES
Compiler construction for Digital Computers
John Wiley & Sons,Inc. New York 1971

[10] R. SETHI, J.D. ULLMAN
JACM vol 17, n° 4 p 715-728 (1970)
The generation of optimal code for artithmetic expressions.