Etude de langages interprétables par une machine langage de haut niveau

Seung Kyu Park

THESE

présentée à

l' Institut National Polytechnique de Grenoble

pour obtenir le grade de
DOCTEUR INGENIEUR
(Génie Informatique)

par

Seung Kyu PARK

ETUDE DE LANGAGES INTERPRETABLES PAR UNE MACHINE LANGAGE DE HAUT NIVEAU

Thèse soutenue le 26 mars 1982 devant la commission d'examen :

Monsieur	*L. BOLLIET*	:	*Président*
Messieurs	*G. MAZARE*	}	*Examinateurs*
	T. MUNTEAN		
	J.P. SCHOELLKOPF		
	S. GUIBOUD RIBAUD	:	*Rapporteur*

INSTITUT NATIONAL POLYTECHNIQUE de GRENOBLE

Année universitaire 1981-1982

Président : Daniel BLOCH

Vice-Présidents : René CARRÉ
Hervé CHERADAME
Marcel IVANES

PROFESSEURS DES UNIVERSITES

ANCEAU François	E.N.S.I.M.A.G
BARRAUD Alain	E.N.S.I.E.G
BESSON Jean	E.N.S.E.E.G
BLIMAN Samuel	E.N.S.E.R.G
BLOCH Daniel	E.N.S.I.E.G
BOIS Philippe	E.N.S.H.G
BONNETAIN Lucien	E.N.S.E.E.G
BONNIER Etienne	E.N.S.E.E.G
BOUVARD Maurice	E.N.S.H.G
BRISSONNEAU Pierre	E.N.S.I.E.G
BUYLE-BODIN Maurice	E.N.S.E.R.G
CAVAIGNAC Jean-François	E.N.S.I.E.G
CHARTIER Germain	E.N.S.I.E.G
CHENEVIER Pierre	E.N.S.E.R.G
CHERADAME Hervé	M.C.P.P
CHERUY Arlette	E.N.S.I.E.G
CHIAVERINA Jean	M.C.P.P
COHEN Joseph	E.N.S.E.R.G
COUMES André	E.N.S.E.R.G
DURAND Francis	E.N.S.E.E.G
DURAND Jean-Louis	E.N.S.I.E.G
FELICI Noël	E.N.S.I.E.G
FOULARD Claude	E.N.S.I.E.G
GENTIL Pierre	E.N.S.E.R.G
GUERIN Bernard	E.N.S.E.R.G
GUYOT Pierre	E.N.S.E.E.G
IVANES Marcel	E.N.S.I.E.G
JAUSSAUD Pierre	E.N.S.I.E.G
JOUBERT Jean-Claude	E.N.S.I.E.G
JOURDAIN Geneviève	E.N.S.I.E.G
LACOUME Jean-Louis	E.N.S.I.E.G
LATOMBE Jean-Claude	E.N.S.I.M.A.G
LEROY Philippe	E.N.S.H.G
LESIEUR Marcel	E.N.S.H.G
LESPINARD Georges	E.N.S.H.G
LONGEQUEUE Jean-Pierre	E.N.S.I.E.G
MAZARE Guy	E.N.S.I.M.A.G
MOREAU René	E.N.S.H.G
MORET Roger	E.N.S.I.E.G
MOSSIERE Jacques	E.N.S.I.M.A.G
PARIAUD Jean-Charles	E.N.S.E.E.G
PAUTHENET René	E.N.S.I.E.G
PERRET René	E.N.S.I.E.G
PERRET Robert	E.N.S.I.E.G

PIAU Jean-Michel	E.N.S.H.G
POLOUJADOFF Michel	E.N.S.I.E.G
POUPOT Christian	E.N.S.E.R.G
RAMEAU Jean-Jacques	E.N.S.E.E.G
RENAUD Maurice	M.C.P.P
ROBERT André	M.C.P.P
ROBERT François	E.N.S.I.M.A.G
SABONNADIERE Jean-Claude	E.N.S.I.E.G
SAUCIER Gabrielle	E.N.S.I.M.A.G
SCHLENKER Claire	E.N.S.I.E.G
SCHLENKER Michel	E.N.S.I.E.G
SERMET Pierre	E.N.S.E.R.G
SOUQUET Jean-Louis	E.N.S.E.E.G
SILVY Jacques	M.C.P.P
SOHM Jean-Claude	E.N.S.E.E.G
VEILLON Gérard	E.N.S.I.M.A.G
ZADWORNY François	E.N.S.E.R.G

PROFESSEURS ASSOCIES

GANDINI Alessandro	M.C.P.P
MAXWORTHY Thony	E.N.S.H.G
MROVEC Stanislas	E.N.S.E.E.G
PARRIAUX Olivier	E.N.S.I.E.G
PEISNER Janos	E.N.S.E.R.G

PROFESSEURS E.N.S MINES SAINT ETIENNE

RIEU Jean
SOUSTELLE Michel

CHERCHEURS DU C.N.R.S (Directeurs et Maîtres de recherche)

FRUCHART Robert	Directeur de recherche
ALLIBERT Michel	Maître de recherche
ANSARA Ibrahim	Maître de recherche
CARRE René	Maître de recherche
DAVID René	Maître de recherche
DRIOLE Jean	Maître de recherche
KAMARINOS Georges	Maître de recherche
KLEITZ Michel	Maître de recherche
LANDAU Ioan-Doré	Maître de recherche
MERMET Jean	Maître de recherche
MUNIER Jacques	Maître de recherche
VERDILLON André	Maître de recherche

CHERCHEURS DU MINISTERE DE L'INDUSTRIE

(Directeurs et Maîtres de recherche - E.N.S Mines Saint Etienne)

LESBATS Pierre	Directeur de recherche
BISCONDI Michel	Maître de recherche
KOBYLANSKI André	Maître de recherche
LE COZE Jean	Maître de recherche
THEVENOT François	Maître de recherche
TRAN MINH Canh	Maître de recherche
LALAUZE René	Maître de recherche
LANCELOT Francis	Maître de recherche

PERSONNALITES HABILITEES A DIRIGER DES TRAVAUX DE RECHERCHE

(Décision du Conseil Scientifique)

E.N.S.E.E.G

BERNARD Claude
BONNET Roland
CAILLET Marcel
CHATILLON Catherine
COULON Michel
EUSTATHOPOULOS Nicolas
HAMMOU Abdelkader
JOUD Jean-Charles
MALMEJAC Yves (CENG)
RAVAINE Denis
SAINFORT (CENG)
SARRAZIN Pierre
TOUZAIN Philippe
URBAIN Georges (Laboratoire des Ultraréfractaires, ODEILLO)

E.N.S.M Saint Etienne

GUILHOT Bernard
THOMAS Gérard
DRIVER Julian

E.N.S.E.R.G

BOREL Joseph
CHEHIKIAN Alain

E.N.S.I.E.G

BORNARD Guy
DESCHIZEAUX Pierre
GLANGEAUD François
LEJEUNE Gérard
PERARD Jacques

E.N.S.H.G

DELHAYE Jean-Marc

E.N.S.I.M.A.G

A mes parents,

à mon épouse Dokeum,

à Soyeon née à Grenoble pendant cette étude.

Je tiens à remercier,

Monsieur le Professeur L. BOLLIET, professeur à l'Université des Sciences Sociales de Grenoble, qui a bien voulu me faire l'honneur de présider le jury de cette thèse, qui m'a accepté dans son équipe avec cordialité à mon arrivée à Grenoble, et qui a vivement encouragé ce travail,

Monsieur R. BOUTTAZ, Ingénieur au CNRS et Responsable de l'Atelier de Micro Informatique de Grenoble, qui a guidé mon travail dans le cadre de l'Atelier,

Monsieur S. GUIBOUD RIBAUD, Ingénieur à Hewelett Packard Grenoble, qui a bien voulu donner son avis sur le contenu de ma thèse,

Monsieur G. MAZARE, Professeur à l'Institut National Polytechnique de Grenoble, qui a bien voulu participer au jury de soutenance,

Monsieur T. MUNTEAN, Maître Assistant Associé à l'Université Scientifique et Médicale de Grenoble, qui m'a prodigué ses conseils pour améliorer la présentation et le contenu lors de l'élaboration de ce document,

Monsieur J.P. SCHOELLKOPF, Ingénieur à l'Institut National de Recherches en Informatique et Automatique du Laboratoire IMAG, qui a suggéré cette étude et m'a conseillé dans mes premiers balbutiements,

Madame C. CHALAND, qui a assuré la dactylographie de ce texte dans des conditions difficiles, avec beaucoup de patience et une grande compétence,

et enfin le Service de Reprographie de l'IMAG qui a assuré la réalisation matérielle de ce document avec le soin habituel,

Seung Kyu PARK.

TABLE DES MATIÈRES

AVANT-PROPOS

La conception et la technologie des systèmes informatiques ont rapidement évolué pendant ces dernières années. Dans le domaine du matériel, la technologie a atteint l'ère des systèmes VLSI qui nous permet de concevoir diverses nouvelles architectures. De plus en plus, des modules auparavant réalisés par le logiciel, tendent à être inclus dans les nouvelles architectures matérielles.

L'apparition des machines langage de haut niveau (LHN) est une conséquence naturelle de cette démarche. En effet, la préoccupation est de trouver le moyen pour le matériel, de simplifier le coût des systèmes dû aux différentes couches de logiciel, maintenant réalisables par le matériel dont le coût diminue.

Il est donc naturel de remarquer que dans les études sur les différentes structures des langages machine (LM), on laissait la place aux études de conception des architectures adaptées aux langages de haut niveau (LHN). Le souci d'efficacité, également pour les phases de compilation, d'optimisation des codes, ..., se voit ainsi nettement augmenté. Ce langage machine (LM) non seulement simplifie les compilateurs mais améliore également la taille et la vitesse des programmes. Les machines LHN sont maintenant des machines système spécialisées (machines base de données), qui sont essentiellement apparues dans les dernières années.

L'étude des machines LHN est donc la combinaison de divers facteurs, mais les domaines suivants comportent encore des problèmes qu'il conviendra de résoudre [8] :

- une meilleure combinaison du logiciel, du matériel et des microprogrammes pour une machine LHN donnée ;
- une conception automatique des outils de développement ;
- le développement des machines multi-LHN, des machines système d'exploitation et des machines bases de données, etc ..
- une conception de la configuration des processeurs fonctionnels autonomes.

Etant donné ce qui vient d'être dit, dans quel contexte se situe notre étude ?

Etant donné un LHN existant, selon l'approche de la machine LHN il s'agit d'abord d'établir le langage machine (LM) de cette nouvelle machine et ensuite de construire l'architecture qui exécute directement ce LM [1]. L'établissement du LM constitue la décomposition du LHN en couches des langages intermédiaires.

Une fois fixé le niveau du LM, la traduction du LHN en LM, soit par le logiciel, soit par le matériel, constitue la phase de compilation, tandis que l'exécution du LM par le matériel constitue la phase d'interprétation. Autrement dit, le choix du niveau du LM interprétable par l'architecture, définit la limite entre la compilation et l'interprétation.

Notre étude vise à établir ce niveau de LM pour que :

1) le LM ait les mêmes caractéristiques que le LHN, à l'exception du décodage des identificateurs,
2) dans l'architecture correspondante, le LHN doit être simplement considéré comme la représentation symbolique du LM,
3) une limite idéale entre la compilation et l'interprétation implique que le compilateur soit simple et systématiquement construit,
4) la machine LHN doit permettre une décomposition du système en couches verticales (éventuellement réalisables par des systèmes VLSI).

La première partie de cette thèse présente les aspects généraux des machines LHN. Le premier chapitre introduit la méthodologie qui sera adoptée pour la structuration des systèmes. Les différentes approches des machines LHN sont analysées dans le deuxième chapitre. Cette analyse conduit aux conditions nécessaires d'une machine LHN idéale. Dans le chapitre 3, les critères d'une machine LHN sont établis (par l'isomorphisme) pour satisfaire les conditions présentées au chapitre précédent. Enfin, le traducteur de la machine LHN, est analysé à la fin de cette première partie.

La deuxième partie décrit successivement le LM de la machine PASCAL (I-PASCAL) et le traducteur de PASCAL. Les critères définis dans la première partie sont appliqués au langage I-PASCAL. L'isomorphisme entre PASCAL et I-PASCAL est ensuite analysé. Cette présentation se termine par la réalisation du traducteur de PASCAL en I-PASCAL, dans lequel on applique les règles de décomposition. La construction systématique du traducteur de PASCAL est également discutée.

Enfin, en annexe, on donne les syntaxes modifiées utilisées pendant la construction du traducteur, ainsi que les structures de I-PASCAL : les descripteurs, les instructions, etc ...

PREMIERE PARTIE

CHAPITRE 1 - DÉCOMPOSITION DES SYSTÈMES

CHAPITRE 2 - APPROCHE DE LA MACHINE LANGAGE DE HAUT NIVEAU

CHAPITRE 3 - ISOMORPHISME DES LANGAGES

CHAPITRE 4 - TRADUCTEUR ISOMORPHE

CHAPITRE 1

DÉCOMPOSITION DES SYSTÈMES

1.1. INTRODUCTION

Dans cette première partie nous procédons à une étude systématique d'une machine langage de haut niveau (LHN).

Les progrès acquis actuellement dans la technologie des composants électroniques constitue l'environnement de départ de la flexibilité de la conception de nouvelles architectures, notamment par l'avènement des systèmes VLSI [5]. Ces techniques permettent de concevoir des architectures nouvelles qui ne sont pas basées sur l'architecture de von Neumann.

C'est le cas des architectures des machines LHN. Le niveau d'interprétation est poussé au maximum afin d'améliorer les performances du système. Cela implique d'utiliser des composants dans lesquels les blocs logiciels sont câblés.

Une méthodologie de conception descendante permet la conception structurée dans les différents modules. Le contexte dans lequel se situe le formalisme présenté dans cette thèse est le suivant :

1) la technologie avancée sur le matériel autorise une flexibilité accrue de la conception,
2) le logiciel est décomposé en blocs spécialisés,
3) les blocs fonctionnels sont de plus en plus identifiés aux blocs matériels grâce à la technologie des composants,
4) une méthodologie est nécessaire à cette identification systématique,
5) l'approche adoptée pour la machine LHN est un cas typique où les contextes précédents sont applicables,
6) il reste néanmoins nécessaire d'établir un formalisme propre en ce qui concerne la conception d'une machine LHN.

1.2. DÉCOMPOSITION EN COUCHES INDÉPENDANTES

On utilisera ici le terme "système" dans un sens très général, comme celui défini dans [6].

Un système est défini comme une agrégation ou un assemblage d'"objets" joints par certaines interactions régulières ou interdépendantes [6]. Cependant, cette définition est si globale que les systèmes statiques y sont inclus. Mais l'intérêt principal sera porté sur les systèmes dynamiques dans lesquels les interactions causent des changements d'état dans le temps.

Une étape importante dans la modélisation d'un système est de choisir une limite entre le système et son environnement. Le choix dépend du but de l'étude. Le terme "endogène" est utilisé pour décrire les activités intérieures d'un système. De même, le terme "exogène" signifie les activités situées dans l'environnement qui influence le système. Un système qui n'a pas d'activités exogènes est appelé un système fermé.

Dans notre étude, nous analysons un système suivant ses représentations en couches indépendantes et pour que chaque couche soit réalisée par des modules indépendants. Ceci nécessite d'établir une borne nouvelle entre les activités endogènes et les activités exogènes en fonction de la dynamique du système.

Ceci fait apparaître les sous-systèmes ; les anciennes activités endogènes deviennent les activités exogènes de ces sous-systèmes (cf. figure 1). L'application itérative de cette démarche constitue la structuration du système en multiples "couches" dans lesquelles les sous-systèmes se trouvent. On appelle la décomposition en couches la "décomposition verticale" et la décomposition d'une couche en sous-systèmes, la "décomposition horizontale". Les sous-systèmes constitutifs d'une couche seront appelés "modules".

La structuration d'un système de télécommunication en couches indépendantes depuis le niveau d'application jusqu'au niveau physique, constitue un exemple typique de cette conception [4], [7]. Les critères de décomposition verticale dans ce système déterminent les protocoles correspondant à chaque niveau.

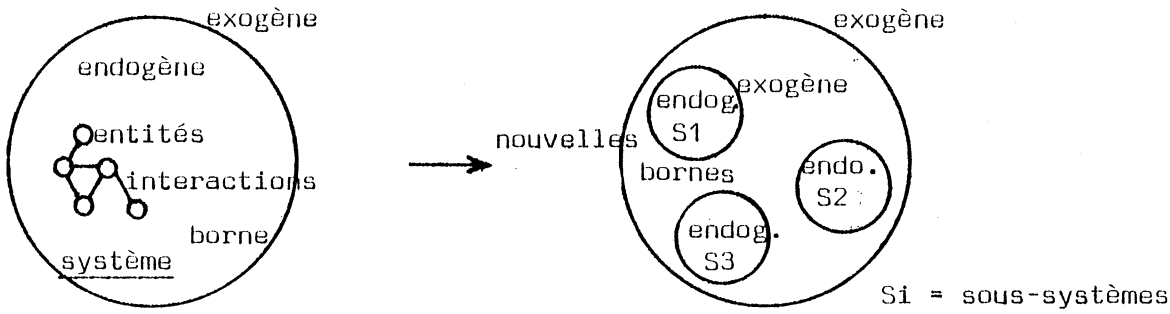


Figure 1 - Décomposition en sous-systèmes

La décomposition que nous venons de présenter se situe dans une des trois catégories suivantes : Soit Sij le sous-système où i est la i-ème couche et j est le j-ème module, et soit Cij le critère correspondant à la décomposition de Sij :
Cas 1 : le critère est unique, tel que C = Cij pour tout i et j ;
cas 2 : le critère de couche est Ci tel que Ci = Cij pour tout j ;
cas 3 : le critère est Cij pour chaque Sij et il est possible que Cij $\neq$ Ci'j' pour i $\neq$ i' ou j $\neq$ j'.

La méthode de programmation structurée reflète le premier cas. Le deuxième cas constitue la décomposition en couches indépendantes. La méthode descendante est basée sur le troisième cas. Or, ces trois catégories sont reliées par une relation d'inclusion. Le cas 1 est un cas spécial du cas 2 et de même, le cas 2 est un cas spécial du cas 3.

Malgré la relation d'inclusion, on adopte ici le cas 2 comme la méthodologie qui sera appliquée. Les couches indépendantes du cas 2 se caractérisent par les indications suivantes :
a) les couches résidentes,
b) la portabilité et la compatibilité par la standardisation,
c) la possibilité de réalisation par le matériel,
d) bien adaptée à l'analyse de la machine LHN.

Le cas 2 fait apparaître une partie supplémentaire ("surcoût") qui ne serait pas apparue si l'approche du cas 3 avait été adoptée. La réorganisation des messages suivant les différents protocoles, la manipulation des descripteurs, et la partie catalogues, ... peuvent être respectivement considérées comme les surcoûts ("overhead") des systèmes de télécommunication, d'exploitation et de bases de données, etc ..

Une fois constituées les couches indépendantes, la conception descendante du cas 3 peut être adoptée pour analyser une couche donnée.

En adoptant les couches indépendantes du cas 2, on aura donc les règles de décomposition suivantes (le mot "bloc" sera utilisé dans le même sens que "module") :

Règles de décomposition :

(1) Etablissement des critères Ci de décomposition.
(2) Décomposition bi-dimensionnelle d'un système en couches indépendantes et en blocs, en appliquant les critères Ci.
(3) Rangement des blocs des surcoûts ("overhead").

La structure d'un système est réalisée par la construction des blocs basée soit sur le logiciel, soit sur le matériel, soit sur la combinaison des deux. Les couches indépendantes d'un système d'exploitation constituent les niveaux d'interprétation depuis l'interface avec les utilisateurs jusqu'à l'interprétation physique.

La technologie avancée permet de plus en plus de remplacer les blocs de logiciel par des blocs matériels. Ces blocs matériels sont traités comme des modules de programmes fonctionnels. De même, il est souhaitable que les blocs de surcoût soient réalisés par le matériel puisque le temps d'exécution du surcoût diminue radicalement par rapport à celui qui résulte du logiciel. Ainsi, malgré l'apparition des surcoût ("overhead"), la vitesse d'exécution du système global ne se dégrade pas.

Les règles de décomposition ci-dessus seront donc étendues ainsi :
(4) Réalisation des blocs de surcoût par le matériel.
(5) Réalisation matérielle des blocs de logiciels spécialisés.

Le système d'interprétation d'un LHN constitue la structure typique des couches. Les couches sont représentées par les langages intermédiaires qui sont exécutés soit par l'étape de compilation, soit par l'étape d'interprétation. Les sous-programmes ou les modules des tâches, sont des blocs créés par la décomposition verticale.

Dans l'approche de la machine LHN, il s'agit de la réalisation totale d'une couche par le matériel. La formalisation des règles de décomposition dans le cas d'un système machine LHN se concrétise donc par l'addition d'une sixième étape :
(6) Réalisation du niveau d'interprétation du LHN par le matériel.

Les étapes (1) à (6) des règles de décomposition sont illustrées dans la figure 2 qui suit.

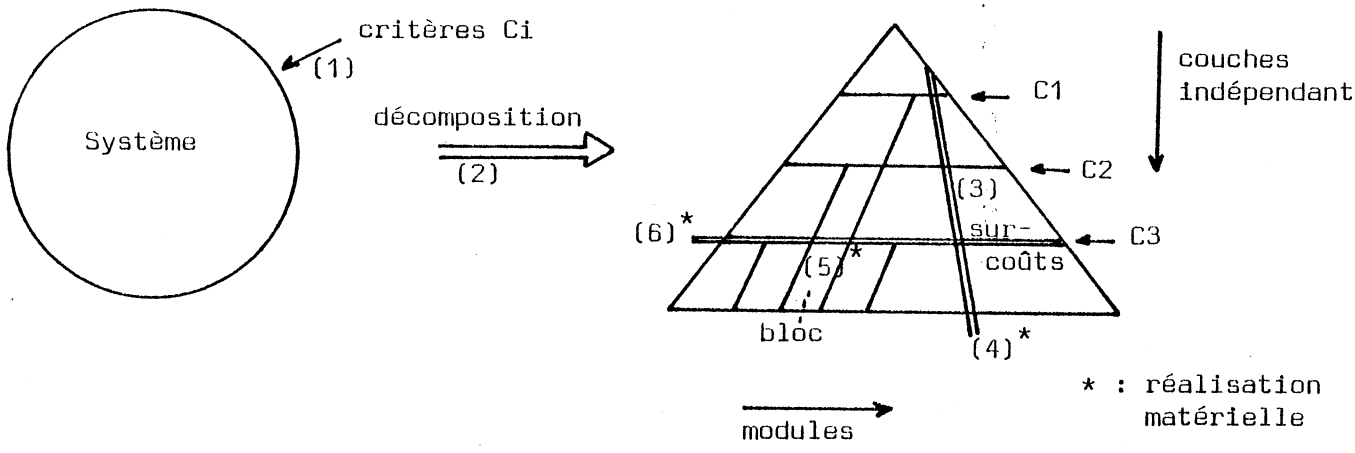


Figure 2 - Règles de décomposition

CHAPITRE 2

APPROCHE DE LA MACHINE LANGAGE DE HAUT NIVEAU

2.1. INTRODUCTION

L'apparition de langages de haut niveau (LHN) comme moyen efficace pour l'écriture des programmes a eu, dans les années 1960, une influence considérable sur les nouvelles architectures proposées. Le premier pas vers des architectures orientées langages a été franchi avec Burroughs B5500, où est apparue la notion de pile, aspect important pour l'exécution des programmes écrits en LHN. C'est dans le courant des années 1970 qu'ont été conçues des architectures orientées langages. Nous en donnons ci-après une liste non exhaustive [8] :

- processeur ALGOL, par Anderson 1961, et CHU 1967 à l'Université de Maryland,
- processeur FORTRAN, par Melbourne et Pugmire 1965, traduction et interprétation basées sur la microprogrammation,
- machine sous-ensemble de FORTRAN, par Baskow,
- processeur EULER, par Wirth et Weber 1967, basé sur le micro-code IBM 360/30,
- processeur PL/1, par Sugimoto 1969, traducteur et interpréteur par plusieurs processeurs autonomes,
- processeur ADAM, 1963, processeur de symbole,
- processeur SYMBOL, 1971, architecture significative de la machine LHN, avec 8 processeurs autonomes pour traduction et exécution,
- machine B1700, ordinateur à plusieurs LHN, machine de micro-logiciels différents spécialisés dans les applications,
- processeur APL, au début des années 1970, caractérisé par le parallélisme inhérent et par les types des données dynamiques,
- processeur SNOBOL, LIST, aérospatial, etc ...

Depuis 1973 au Laboratoire IMAG de Grenoble, la machine PASCAL, "PASC-HLL" a été étudiée, construite et est en cours de test [2]. C'est une architecture pipe-line réalisée par des microprocesseur microprogrammables en tranches. I-PASCAL, le langage machine (LM) de cette architecture, est conçu à partir de PASCAL [3]. Nous présenterons dans une seconde partie ce langage dans la forme proposée par notre formalisme.

On a également vu au début des années 1970 d'autres exemples comme le HP 3000 de Hewlett Packard et le 32/S de Microdata. Ce sont des architectures à pile réalisées sur des mini-ordinateurs [9]. La machine PASCAL Microengine est basée sur un microprocesseur commercialisé [10]. C'est également une architecture à pile dont le LM est un P-code.

La microprogrammation et les micro-circuits ont offert aux concepteurs des outils souples et efficaces pour la conception de ces nouvelles classes d'architectures.

En somme, le développement des machines LHN se situe dans le contexte suivant. Le premier type d'ordinateurs était basé sur l'architecture de Von Neumann, et les langages disponibles à cette époque étaient les langages d'assemblage qui sont une r présentation symbolique du LM. L'évolution vers les langages de très haut niveau était en discordance avec l'évolution relativement lente des architectures qui restaient au niveau de Von Neumann. Ce décalage implique nécessairement l'existence de compilateurs pour ces langages. Grâce aux progrès rapides de la technologie du matériel et des composants, des études ont montré que le système d'interprétation des LHN peut être remplacé par des parties réalisables par le matériel. Cette possibilité fait apparaître les nouvelles architectures des machines LHN.

D'autres motivations pour l'apparition des machines LHN sont :

1) le logiciel est la partie la plus coûteuse des systèmes d'exploitation. En éliminant les multiples couches de logiciel, ont peut diminuer son coût de manière significative ;

2) le LM conventionnel est si primitif, qu'il est très difficile de poursuivre l'exécution des programmes par les utilisateurs. En rapprochant le niveau LM des LHN, l'utilisateur peut éventuellement poursuivre l'exécution des programmes en mode interactif ;

3) on peut concevoir de manière systématique une meilleure architecture adaptée aux LHN qui soit plus efficace [11], [12] ;

4) la technologie des composants est remarquablement avancée, si bien que les blocs fonctionnels du logiciel peuvent être réalisés avec grande flexibilité par la technologie microprogrammée ;

5) l'apparition des micro et mini-ordinateurs personnels qui peuvent être des machines LHN, évite aux utilisateurs la connaissance spécifique du logiciel des systèmes [13].

2.2. HIÉRARCHIE DES LANGAGES

On peut distinguer les deux étapes suivantes :

(1) la définition d'un LHN qui soit le mieux adapté à l'expression des applications,

(2) la conception de l'architecture adaptée à ce LHN basée sur les règles de décomposition.

L'étape (2) consiste d'abord à définir le LM qui est plus proche du LHN et ensuite à construire l'architecture qui exécute directement ce LM.

Si le LM n'est pas le LHN, l'étape de la traduction du LHN en LM constitue la phase de compilation et ensuite l'exécution du LM constitue l'étape d'interprétation par le matériel. Cette relation est illustrée par la figure 3.

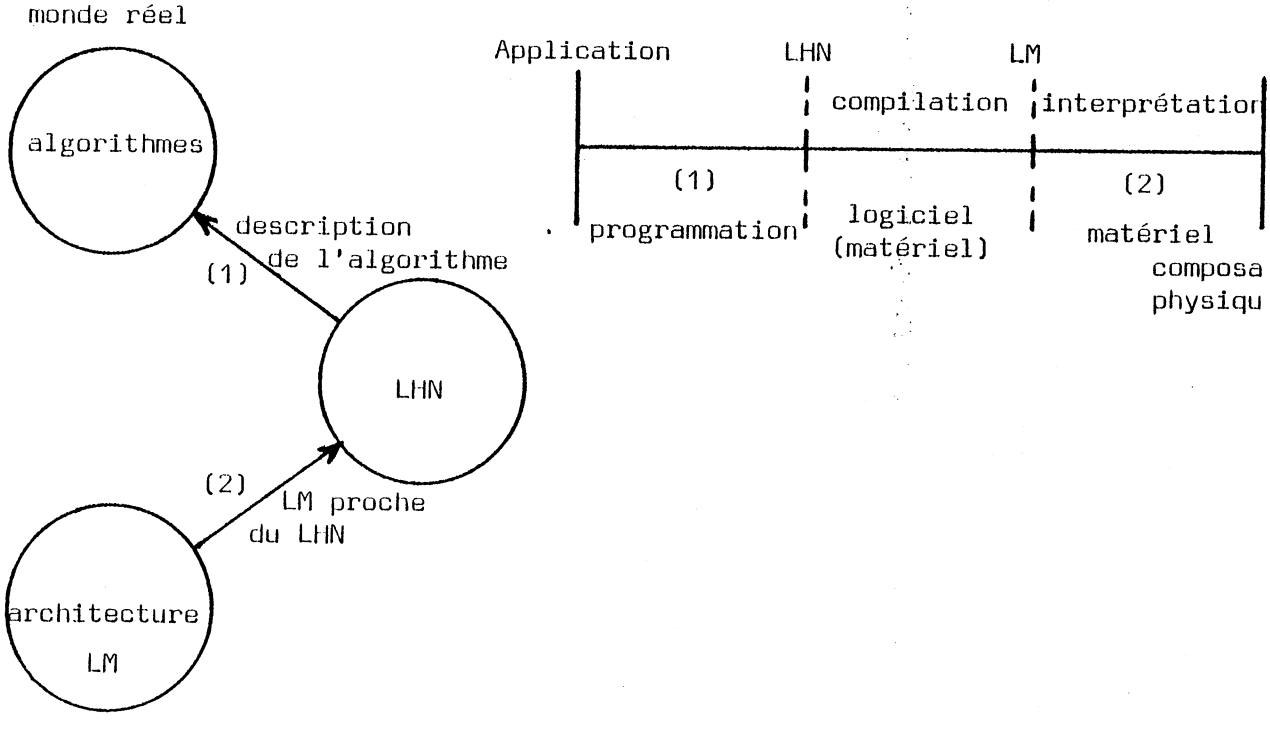


Figure 3 - Approche de la machine LHN

Afin de définir le contexte dans lequel se situe le LHN, précisons que la conception du langage peut être considérée comme dépendante des caractéristiques des applications visées. Cela signifie que plusieurs LHN peuvent exister selon la catégorie d'espace des problèmes dans lesquels se situent les applications.

Etant donné l'espace des problèmes p, soit R la relation définie par (p1, p2) ϵ R si l'algorithme du problème p1 et celui de p2 peuvent être classés dans la même catégorie. Cette relation constitue une relation d'équivalence entre différents LHN.

Ainsi la définition d'un LHN peut-elle se donner en deux étapes :
1) déterminer les classes d'équivalences correspondantes,
2) spécification du LHN adapté à la classe d'équivalences donnée.

Plusieurs niveaux peuvent apparaître. Ici, nous considérons une hiérarchie à cinq niveaux :

1) pseudo-langage :
   ce langage conceptuel est basé sur le maximum de détails qu'un programmeur veut pouvoir expliciter dans l'espace des applications qu'il envisage ;

2) langage de très haut niveau (LTHN) :
   c'est un langage basé sur un domaine spécifique, de telle sorte qu'il n'est pas toujours possible qu'un programme écrit dans un langage LTHN1 puisse être traduit dans un langage LTHN2 ;

3) langage de haut niveau (LHN) :
   le LHN est, dans la conception courante, comme les langages procéduraux, comme par exemple PASCAL, ALGOL, etc .. ;

4) langage de pile (LP) :
   le LP est le langage défini pour manipuler des structures de pile. Ces LP sont utilisables pour l'interprétation intermédiaire de LHN ou pour la portabilité de LHN [14] ;

5) langage de von Neumann (LV) :
   le LV est le langage d'assemblage, de type LM, qui représente l'architecture de von Neumann. Ce langage est basé sur un minimum de répartition de l'espace des applications. Il se situe en bas de la hiérarchie qui, en fait, est déterminée par un gradient des répartitions.

Soit N(L) la cardinalité de la répartition de l'espace des applications qui est déterminée par la relation d'équivalence de catégorisation. La relation ordinale peut être exprimée comme suit :
N(Pseudo.L) > N(LTHN) > N(LHN) > N(LP) > N(LV)

Les mots "langage intermédiaire" ou "langage machine" (LM) sont donc réservés pour signifier un des langages de la hiérarchie :
- langage intermédiaire : le langage qui se trouve entre le langage source et le langage objet ;
- langage machine (LM) : le langage qui est directement exécuté par la machine.

2.3. MACHINES ORIENTÉES LANGAGES DE HAUT NIVEAU

"Machine orientée langage" et "machine LHN" se distinguent par le fait que la première implique n'importe quel niveau de langage dans la hiérarchie.

L'idée principale de la conception d'une machine orientée langage est d'accomoder l'architecture à la syntaxe et la sémantique du langage de façon à ce que :
1) l'espace des données du langage puisse être facilement représenté par la structure des données du LM,
2) les instructions du langage et les instructions du LM soient cohérentes.

Il y a plusieurs catégories de machines orientées langages. Chu [16] classe ces architectures dans quatre types : architecture de von Neumann, architecture orientée syntaxe, architecture d'exécution indirecte, architecture d'exécution directe. Il a concentré ses travaux sur l'étude des architectures d'exécution directe, c'est-à-dire des machines qui exécutent directement le LHN. Chu indique les motivations de l'architecture d'exécution directe de la manière suivante :
1) la réduction du coût de développement du logiciel en éliminant certaines couches
2) la poursuite directe de l'exécution du programme en mode interactif,
3) une conception optimisée et appropriée au langage sans surcharges inutiles.

L'approche de Chu suscite de notre part les remarques suivantes :

1) est-ce une seule architecture d'exécution directe qui satisfait les besoins de l'approche de la machine LHN ?

2) Il faut définir le mot "direct" plus strictement. Chaque architecture peut être considérée comme l'architecture d'exécution directe par rapport à son LM. L'architecture d'exécution directe ne signifie donc pas nécessairement qu'il s'agit de la machine LHN si son LM n'est pas classé comme LHN.

3) Même si l'architecture exécute le LHN directement au niveau matériel, il faut bien établir la limite entre l'étape de compilation et celle d'interprétation. Chu classe la machine qui exécute l'étape de compilation dans une classe d'architecture d'exécution indirecte, bien que ce soit réalisé par le matériel [17]. Autrement dit, l'étape d'interprétation du LHN par le matériel est la seul qu'il considère pour la conception de la machine LHN. Pourtant, si une phase de compilation est si simple et rapide que la machine puisse exécuter des programmes en mode interactif, il n'est pas nécessaire d'insister uniquement sur la machine d'exécution directe.

4) Si un programme est utilisé plusieurs fois sans modification, il suffit d'une seule compilation. Il s'avère inefficace dans ce cas d'utiliser l'architecture d'exécution directe qui fait à chaque fois une analyse syntaxique. Il serait donc raisonnable d'avoir le niveau intermédiaire d'exécution qui permet deux options, avec ou sans compilation, selon la nature des applications, plutôt que l'architecture figée qui ne permet que l'exécution directe du LHN.

5) Le chargement du programme source à exécuter dans la mémoire principale est une opération coûteuse. De plus, il doit rester résident jusqu'à la fin de l'exécution en occupant l'espace mémoire.

Wortmann [18] donne une classification légèrement différente : une machine du langage source et une machine du langage intermédiaire. Il explique l'inefficacité de la machine du langag source (qui correspond à l'architecture d'exécution directe de Chu) par ceci :

1) la recherche linéaire de l'instruction prochaine à exécuter,
2) la préparation d'une pile provisoire pour l'évaluation d'expression, etc ..

Flynn [19] a indiqué que les conditions suivantes doivent être satisfaites pour concevoir la machine LHN idéale (appelons cela les "critères de Flynn") :

1) le processus de compilation doit se faire en une ou deux phases simples ;
2) entre les objets du LHN et ceux du LM ("image" selon Flynn), il faut établir une correspondance biunivoque ;

3) la tâche principale de la traduction (compilation) doit être seulement le décodage des identificateurs du LHN ;
4) la transition d'état entre LHN et LM doit être la même (transparence) ;
5) une instruction du LM doit correspondre à une unité sémantique du programme source LHN. Dans une machine traditionnelle, une unité sémantique de LHN était représentée par plusieurs instructions de LM de la machine hôte.

Nous formaliserons ces critères de Flynn au chapitre 3.

2.4. REMPLACEMENT DES COUCHES DE LOGICIEL PAR LE MATÉRIEL

D'après les règles de décomposition adoptées, le système d'interprétation du LHN peut être décomposé en blocs bi-dimensionnels. Une des étapes de ces règles est d'améliorer le niveau d'interprétation du LHN par le matériel. Cette amélioration peut être réalisée d'abord par le remplacement des blocs qui se trouvent dans une couche donnée (appelée "approche horizontale") et ensuite par le remplacement des niveaux des couches (appelée "approche verticale") par le matériel.

2.4.1. Approche bi-dimensionnelle : approche horizontale

Les unités fonctionnelles d'une structure de programmes de type co-routine ou procédure, constituent la décomposition horizontale. On peut diviser les co-routines en deux catégories :
1) les co-routines définies par les utilisateurs,
2) et les co-routines fonctionnelles (standard) contenues dans une bibliothèque.

La figure 4 indique la place de ces co-routines dans l'approche bi-dimensionnelle (ligne horizontale).

Les lignes-source du LHN peuvent être considérées comme les multiples couches de macro-définition dont les expansions en LM éventuel sont réalisées par le compilateur. Les multiples niveaux de couches sont exprimés sur la ligne verticale de la figure 4.

L'approche de la machine LHN basée sur les règles de décomposition peut donc être réalisée d'une manière bi-dimensionnelle :

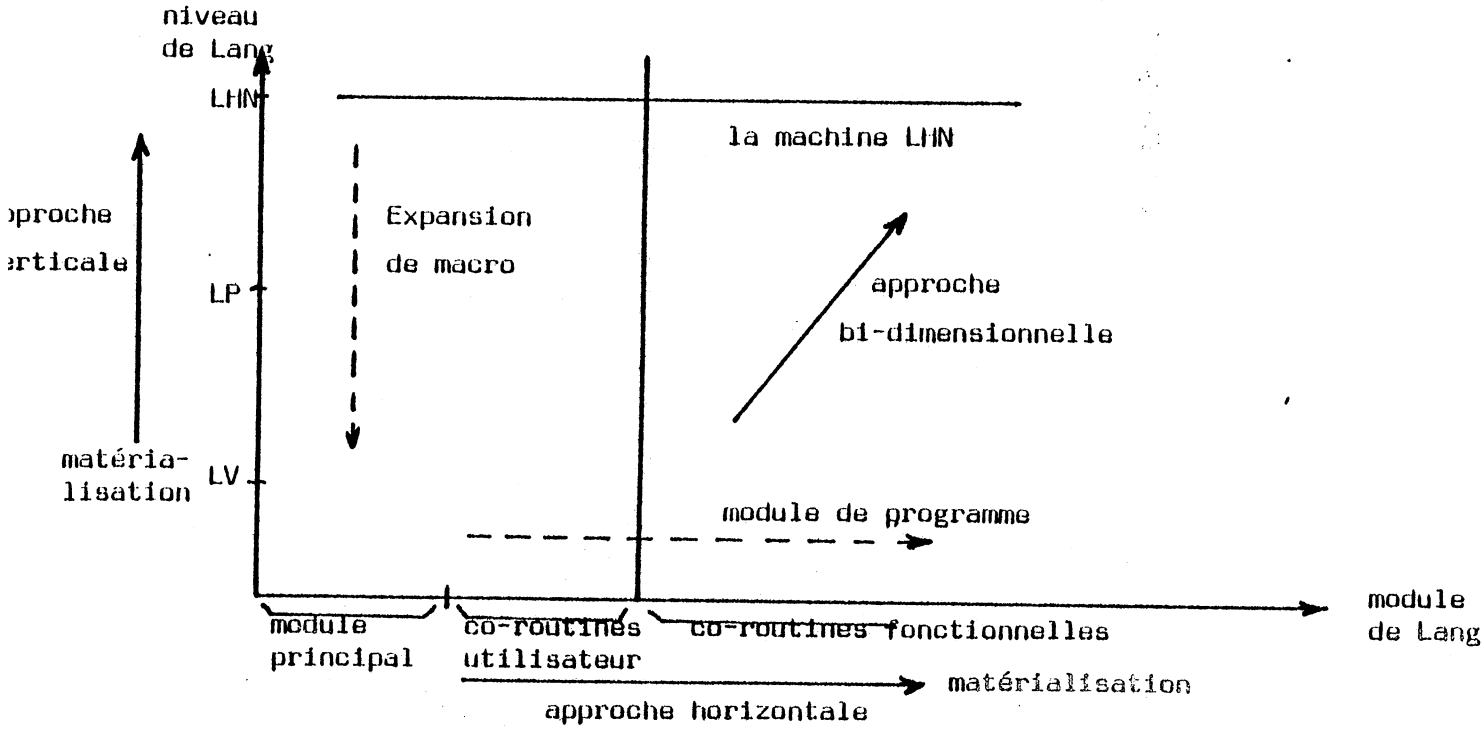
1) Approche horizontale :
   remplacer les co-routines fonctionnelles par le matériel pour un niveau donné. Ainsi, l'accès au module est accompli soit par l'instruction spéciale préparée par ce module, soit par la manipulation comme unité d'entrée/sortie.

2) Approche verticale :
   étudier une machine dont le niveau du nouveau LM soit plus élevé dans la hiérarchie de la figure 4 que celui de l'ancien LM et réaliser l'interpréteur du nouveau LM par le matériel en réécrivant les co-routines de la bibliothèque dans le nouveau LM du système.

3) Approche bi-dimensionnelle :
   réaliser à la fois l'approche horizontale et l'approche verticale.

Le schéma de la figure 4 résume sommairement les approches de la machine LHN.

Dans le système I-PASCAL, des fonctions telles que ABS, SQRT, ODD, TRUNC, INT, CHR, SUCC, PRED, PACK, UNPACK, NEW, CORREC, DISPOSE, etc .. sont définies comme des instructions opérationnelles [3]. Ces fonctions sont utilisées dans la machine classique par les appels de procédures. La conception de systèmes VLSI pour la réalisation d'une fonction spécialisée dans le "pattern matching" [20], les plaques de circuits arithmétiques, les interfaces d'entrée/sortie, et les unités électroniques des blocs de télécommunication, sont des exemples des blocs réalisés par l'approche horizontale.



2.4.2. Approche verticale

2.4.2.1. Espace de données

L'approche verticale de la machine LHN est destinée à améliorer le niveau d'interprétation par le matériel. La macro-définition est un exemple de couche de décomposition verticale. Une couche est totalement réalisée par le matériel.

Dans la structure de programme, les données sont d'abord déclarées (par type de variable) et ensuite elles sont manipulées par les instructions de référence. Afin d'améliorer le niveau du langage, il faut d'abord améliorer le niveau accessible des données.

Le couple (adresse, valeur) est appelé objet. L'objet de l'architecture de von Neumann est très simple. Par l'utilisation directe de la mémoire, l'espace d'adressage constitue une simple séquence linéaire. La ressource offerte aux utilisateurs est la mémoire et les registres : accumulateurs et ports d'entrée/sortie, etc .. La valeur est également très simple : elle est représentée par un mot de l'architecture qui peut être différemment interprété selon l'instruction manipulant cette donnée.

Dans une machine à pile, l'adressage est du type bi-dimensionnel. L'espace d'adressage est donc représenté par le couple (niveau lexique, déplacement). La ressource offerte aux utilisateurs est la pile et les registres spéciaux qui manipulent la pile : par exemple, le "display" de B6700 [9]. Le sommet de pile constitue l'adresse implicite pour les allocations/désallocations.

Malgré le mode d'adressage différent entre la machine de von Neumann et celle à pile, la structure de la valeur est représentée de la même façon par un mot.

Le tableau qui suit résume l'objet par comparaison avec une machine LHN.

Langage / Objet	LV Machine von Neumann	LP Machine à pile	LHN
(1) adressage (binding)	séquence linéaire	couple (niveau lexique, déplacement)	décodage symbolique
(2) valeur	mot mémoire	mot de la pile	donnée structurée

La traduction d'un LHN en LM fait apparaître le temps d'adressage ("binding") : c'est le moment où l'objet de LHN est représenté par les objets de LM. Ceci pose le problème du compromis entre l'efficacité et la flexibilité. L'efficacité est obtenue par le "binding" plus tôt, alors que la flexibilité est obtenue par celui intervenant plus tard.

La traduction d'un LHN en LM fait apparaître la "fragmentation" d'un objet du LHN en objets du LM. Par conséquent, les deux aspects suivants sont observés :
1) l'espace de données (valeur) de LHN est découpé,
2) une instruction du LHN qui manipule une donnée est représentée par plusieurs instructions du LM.

L'exécution d'une couche donnée est donc basée sur la connexion soit un à un (disons "permutation"), soit un à plusieurs (disons fragmentation) entre les objets de haut niveau et ceux de bas niveau. Signalons toutefois que dans la structure de OSI référence, il existe une troisième sorte de connexion : plusieurs à un ("multiplexage").

La fragmentation, pour une opération donnée d'un objet du LHN, cause plusieurs transfers répétés de données qui sont fragmentées en objets du LM et beaucoup d'opérations primitives des données fragmentées. Il est donc recommandé [22] d'utiliser une architecture avec de nombreux registres spécialisés dans la manipulation des opérandes pour que l'on puisse réduire le surcoût dû aux transfers fréquents de données entre les registres et la mémoire.

2.4.2.2. Descripteurs

Les données fragmentées de LM peuvent être regroupées par les descripteurs en décrivant les attributs de la structure. Il y a une place particulière dans la mémoire occupée par les descripteurs qui définissent le mode d'accès, la précision, l'adresse et la structure de données. Un descripteur contient donc l'adresse de l'objet avec les informations nécessaires concernant les données structurées [23].

Dans l'approche de la machine LHN, l'espace de données représenté par la définition de type, peut être réalisé par la référence des descripteurs. Le système de descripteurs se caractérise par :

1) il permet de réduire la taille des instructions en offrant le format fixé et compact [3] ;
2) le mode d'accès des données est contrôlé par le descripteur ainsi que la protection de la mémoire et le test de correction des programmes. Les descripteurs fonctionnent ainsi comme des outils de diagnostic ;
3) la réallocation des données et des programmes peut être simplement réalisée par la modification des pointeurs de descripteurs,sans modification des programmes. Ainsi obtient-on des mécanismes efficaces de gestion de mémoire ;
4) les performances du système sont ainsi améliorées.

La figure 5 visualise l'approche verticale du descripteur et son contexte.

Le point important à considérer dans un système de descripteurs est la caractéristique des instructions non-typées ("langage de type tolérant" [23]). Ceci signifie que pour une opération donnée, il n'existe qu'une instruction correspondante, même s'il y a plusieurs types d'opérandes pour cette opération ("overloading" dans le langage ADA [24]).

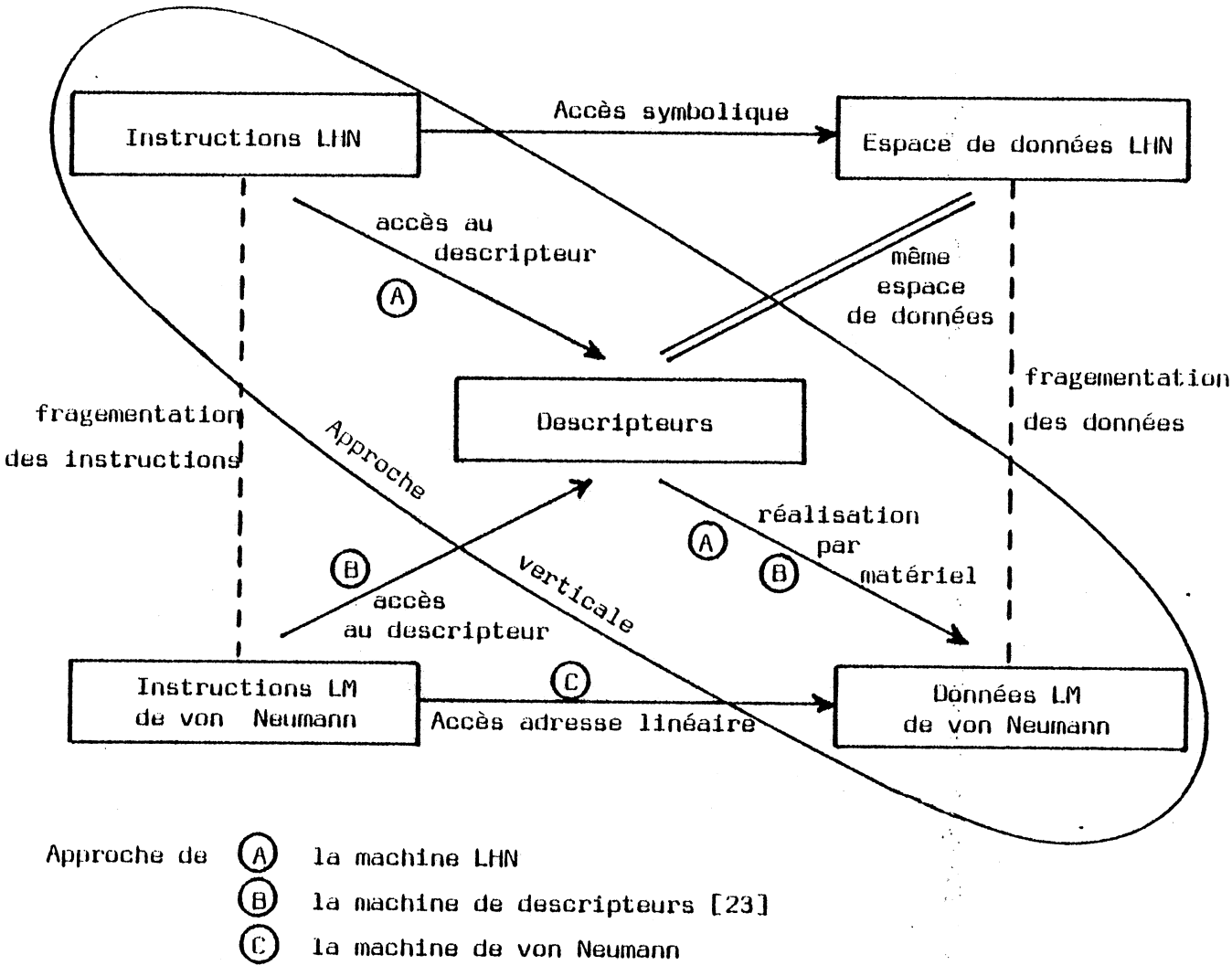


Figure 5 - Approche des descripteurs

Exemple :

Dans la machine HP 3000, l'expression B * C - A + D * E est représentée par les instructions ci-après, où A est réel, B et C sont entiers, D et E sont des données longues [9] :

```
LOAD B       ;  charger l'entier B
LOAD C       ;  charger l'entier C
MPY, FLT     ;  multiplication d'entier et conversion en réel
LDD A        ;  charger le réel A
FSUB, DZRO   ;  soustraction et conversion en donnée longue
LDL D        ;  charger la donnée longue D
LDL E        ;  charger la donnée longue E
EMPY,  EADD  ;  multiplication de données longues et addition de données longu
```

LOAD, LDD et LDL sont des instructions de chargement de données : pour entier, réel et donnée longue respectivement.
MPY et EMPY sont des instructions de multiplication dont les opérandes sont de type entier et donnée longue respectivement.
On voit dans cet exemple que pour une opération donnée il y a plusieurs sortes d'instructions correspondant aux types des données.

Dans cet exemple, les variables A, B, C, D et E sont déclarées par des types simple standard. Si elles étaient déclarées par des types de données structurées, le volume du programme serait multiplié par le facteur du nombre de fragmentation des données. Par exemple, si A et B sont des données structurées qui comportent respectivement les trois éléments des entiers, au lieu de LOAD B, LOAD C, MPY, FLT, on aurait pour chacun trois fois le chargement des éléments B et C et trois fois la multiplication et les conversions. Donc, bien que la machine HP 3000 soit classée dans les machines à pile, il faut fournir plusieurs sortes d'instructions opératives pour une opération donnée, si l'on n'utilise pas le système de descripteurs

Prenons un autre exemple :

dans le langage I-PASCAL, il suffit d'une seule instruction pour chaque opération donnée quel que soit le type des données [3]. L'expression de l'exemple précédent est représentée dans I-PASCAL comme la séquence suivante (où les variables A, B, C, D et E sont des données structurées de n'importe quel type) :

```
REF B ;        accès à B
REF C ;        accès à C
MULT  ;        multiplication
SUB   ;        soustraction
REF D ;        accès à D
REF E ;        accès à E
MULT  ;        multiplication
ADD   ;        addition
```

Bien qu'il y ait des données structurées, la taille du programme ne se multiplie pas. La machine PASC-HLL exécute directement le langage I-PASCAL par la référence des descripteurs des variables [2].

La machine à préfixe est une partie complémentaire de la structure de descripteurs. Dans un système de données à préfixe [18], chaque valeur stockée dans une mémoire est associée à quelques bits qui spécifient l'interprétation des données. Il y a plusieurs moyens d'utiliser des préfixes : le champ de préfixes fait la distinction entre les données et les instructions en assurant une bonne fiabilité d'exécution ; un autre champ de préfixes est celui de la télécommunication des données dans lequel les codes de parité ou Hamming donnent les moyens de détection ou de correction des erreurs ; et enfin le champ de préfixes de clés pour la protection de la mémoire contre les tentatives d'accès non autorisées. Avec l'extension de la notion de machine à préfixes, la conception de la nouvelle architecture, appelée "format de message" est discutée dans [25] et basée sur les unités des informations abstraites. I-PASCAL adopte le champ de préfixes dans une partie du descripteur pour permettre à la machine PASC-HLL de l'utiliser en cours d'interprétation [2].

2.4.2.3. Machine à pile

L'autre structure qui permet une approche verticale est l'architecture à pile. La structure de pile, réalisée soit par logiciel, soit par matériel, est l'outil adéquat pour :

1) l'interprétation du langage bloc structuré,
2) le mécanisme des appels/retours des sous-programmes,
3) le système d'interruption.

Le compilateur du LHN utilise la pile pour l'évaluation de l'expression infixée en la transformant en notation Polonaise renversée. Le "heap storage" est également utilisé dans certains langages, par exemple la pile de PL/1 CONTROLLED, puisque la gestion de l'allocation/désallocation de la mémoire peut être asynchrone par rapport aux entrées/sorties des blocs de procédures.

Un objet du LHN est fragmenté en objets du langage de pile (LP) dans lesquels l'adresse symbolique de LHN est transformée en celle du couple (niveau lexique, déplacement). Cela engendre les instructions de LP qui manipulent la pile dont les fonctions sont allocation/désallocation des données et la gestion d'adressage du couple par les chaînes dynamiques et statiques. La machine à pile manipule le mécanisme d'adressage de pile basé sur le matériel qui aurait été fait par logiciel dans la machine de von Neumann. La figure 6 illustre l'approche de la machine à pile dans le contexte d'une approche verticale.

Cependant, le système de descripteur est une approche qui permet le même espace de données que celle de LHN ; le système de pile s'occupe de la gestion de la manipulation des données. Le programmeur de la machine à pile doit précisément connaître la structure et le fonctionnement de la pile, et il doit choisir les instructions adéquates de la pile. Ceci pose le problème de l'optimisation des codes.

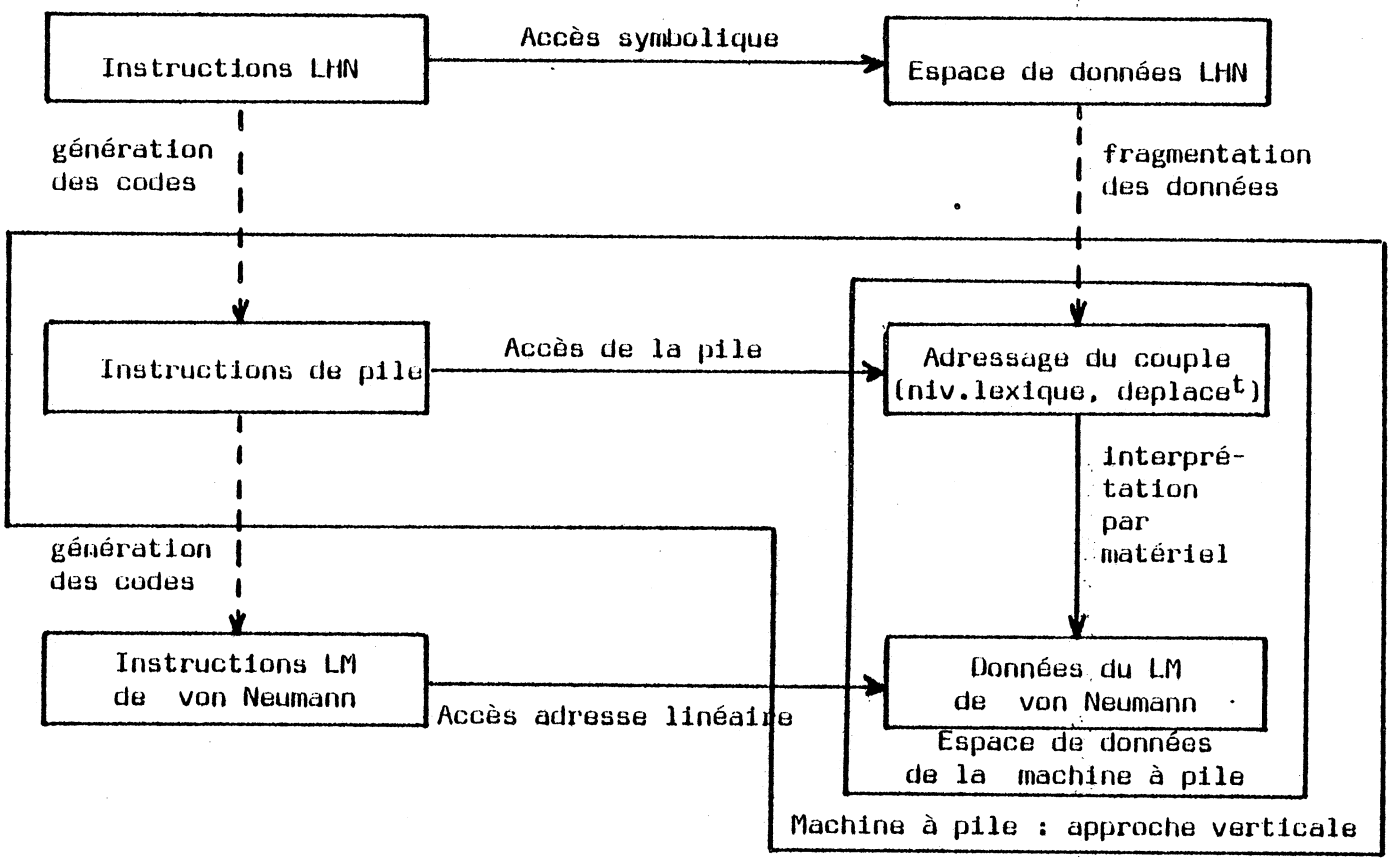


Figure 6 - Approche de la machine à pile

L'exemple suivant montre l'optimisation des instructions de la pile HP 3000. L'expression B * C - A + D * E est la même que précédemment, sauf que toutes les variables sont considérées comme des entiers [9] :

LOAD B		LOAD B
LOAD C		MPYM C
MPY	optimisé	
LOAD A		SUBM A
SUB		
LOAD D		LOAD D
LOAD E		MPYM E
MPY		
ADD		ADD

2.4.3. Réalisation de l'architecture matérielle

Nous venons de présenter l'architecture de la machine LHN basée sur l'approche bi-dimensionnelle. La figure 7 illustre les positions du descripteur et de la pile par rapport aux fonctions standard.

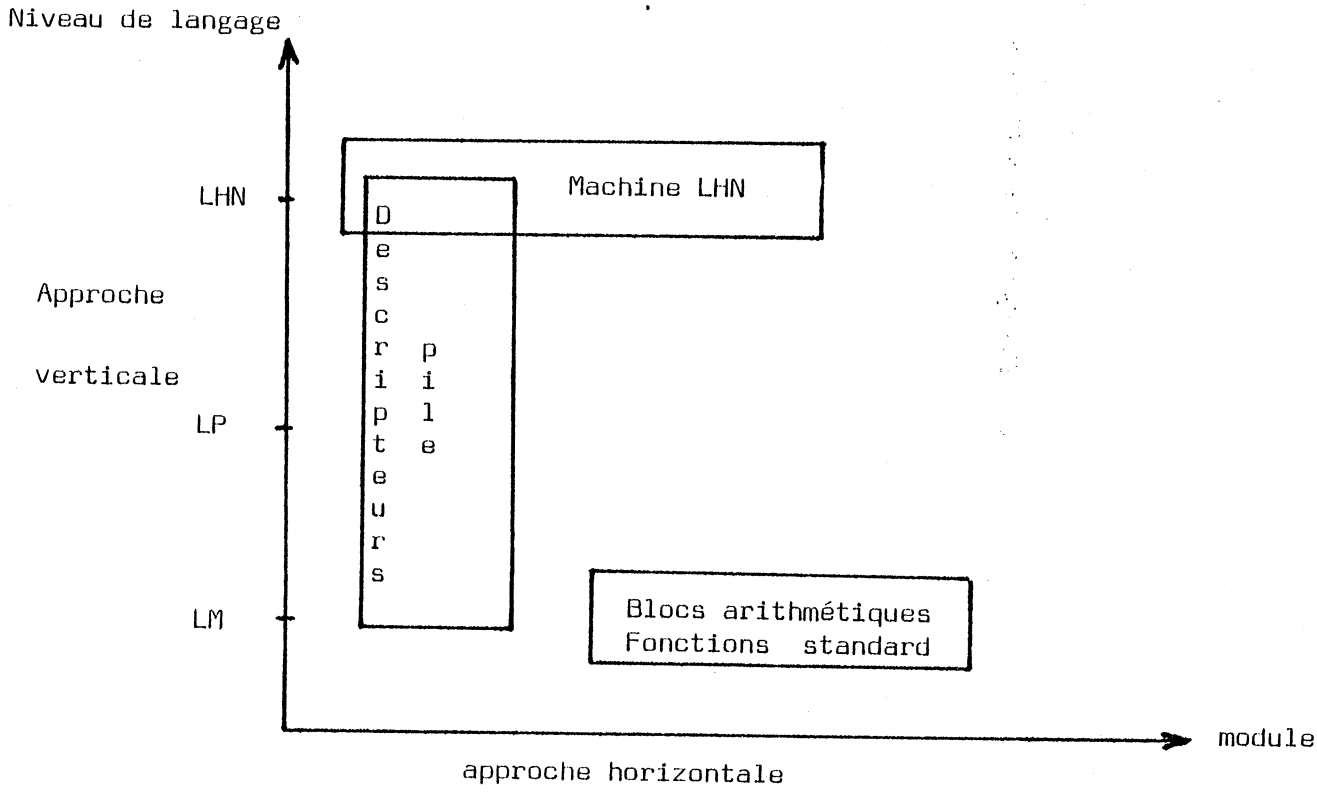


Figure 7 - Comparaison des systèmes par rapport à l'approche bi-dimensionnelle

Cette approche de la machine LHN signifie que la charge de l'interprétation du LHN qui était assurée par le logiciel est effectuée par le matériel.

Les facteurs essentiels à considérer pour l'architecture matérielle de LHN sont donc :
1) les composants (y compris les processeurs),
2) la vitesse globale du système,
3) l'architecture organisant les composants,
à condition que la complexité et le coût du système soient raisonnables et que la vitesse globale du système ne soit pas inférieure à celle de l'architecture conventionnelle.

Les composants disponibles sont les microprocesseurs en tranche, les macro-composants, les tableaux logiques et les blocs de VLSI spécialisés dans la fonction donnée, etc .. Notamment, la technique de la microprogrammation offre un outil flexible et efficace pour la conception de la nouvelle classe d'architecture De plus, la technologie VLSI permet de réaliser cette architecture dans une boîte monolithique après avoir testé le fonctionnement basé sur les composants conventionnels [5].

Parmi les microprocesseurs microprogrammables en tranche, citons : la série INTEL 3000, la série MMI 5700/6700, la série MOTOROLA 10800, TI 74 S 481/74 S 482, et la série AM 2900 [27].

La vitesse globale du système peut être améliorée par les deux approches suivantes:
1) composants,
2) architecture.

La vitesse des composants est de plus en plus améliorée par les technologies de fabrication. Elle approche la vitesse limite électronique caractérisée par le temps de propagation inhérent. Il est cependant difficile d'améliorer la vitesse globale par la seule approche du niveau composants, puisqu'il faut améliorer la vitesse de tous les composants. Un composant plus lent est le facteur critique qui décide de la vitesse globale du système.

Les méthodes courantes au niveau de l'architecture sont :

. pipe-line,

. parallélisme.

Dans un système pipe-line, le déroulement de l'exécution du programme est d'abord analysé au niveau des micro-instructions et ensuite les processeurs fonctionnels exécutent leurs micro-instructions. Les queues des files d'attente sont constituées entre les processeurs contrôlés, par le mécanisme de synchronisation [2]. Cette architecture est fréquemment réalisée par la technique de microprogrammation.

Lorsque l'indépendance opérationnelle est trouvée, elle peut être réalisée par l'architecture parallèle. Pour cela, les blocs homogènes sont construits sous la forme de vecteurs ou de matrices [28]. L'architecture adaptée au langage parallèle est constitué par les multiprocesseurs qui exécutent parallèlement les multiples branches des instructions [29]. La disponibilité des microprocesseurs ainsi que le coût inférieur, permettent d'avoir un système multi-micro-processeurs avec un grand degré de parallélisme.

Dans une approche horizontale, les blocs fonctionnels se situent autour du processeur principal en communiquant sur la base d'entrées/sorties. C'est une organisation maître/esclave. Cependant, dans une approche bi-dimensionnelle, le système est organisé sur les multiprocesseurs autonomes spécialisés dans les fonctions données. L'exemple typique est le processeur SYMBOL qui a huit processeurs autonomes tels que : le processeur central, le traducteur, l'inter-processeur, le contrôleur de mémoire, l'allocateur de mémoire, le processeur du canal de disque, les contrôleurs de canaux et le système superviseur [8]. La figure 8 montre les organisations de ces approches.

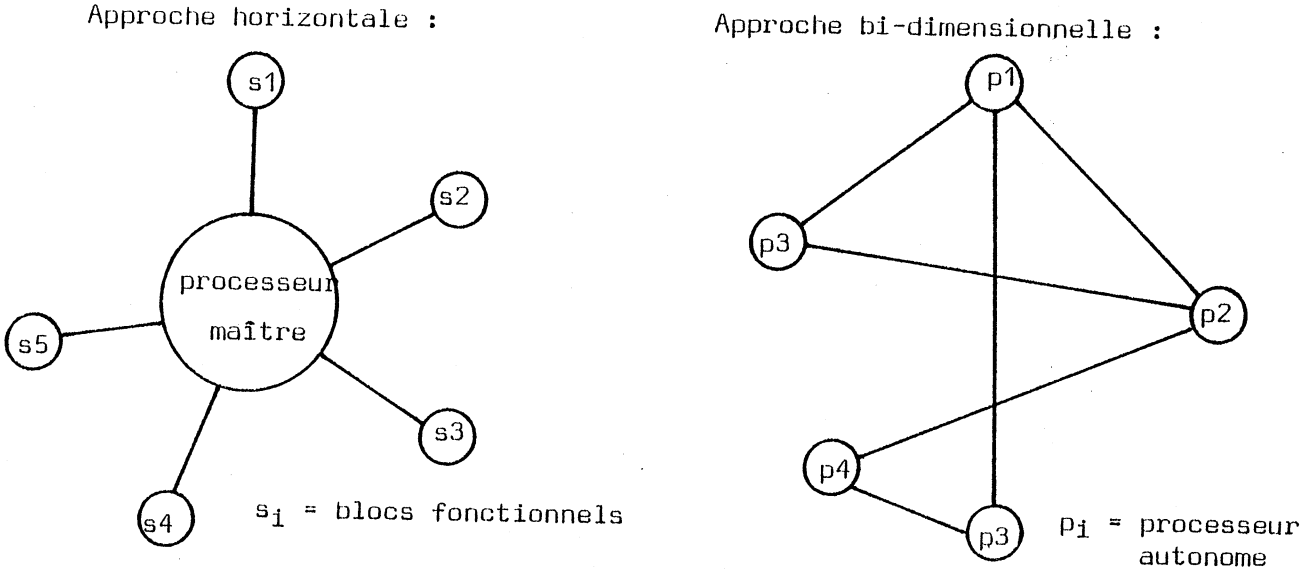


Figure 8 - Organisations des deux approches

2.5. CONCLUSION

Afin d'établir le niveau de LM interprétable par la machine LHN, on a indiqué dans ce chapitre les deux approches qu'il faut éviter :
. celle de l'architecture d'exécution directe de LHN
. et celle de l'architecture traditionnelle.

L'exécution directe ne convient pas pour les raisons suivantes :
1) le traitement du programme source symbolique pour l'exécution directe provoque une occupation volumineuse de la mémoire. Le programme contrôle est basé sur la recherche symbolique de texte lexique.
2) Si le programme est exécuté plusieurs fois sans modification, il est inefficace d'exécuter chaque fois le programme avec l'analyse lexicographique et syntaxique. Cette architecture est donc utile pendant la phase de développement d'un programme.
3) Il est difficile de restaurer les erreurs propagées par l'erreur syntaxique en exécutant directement les fichiers de données par le système d'entrée/sortie (disque, bande magnétique, ...).
4) Il faut décoder, à chaque exécution, les noms symboliques de LHN.

Pourtant cette architecture présente les avantages suivants :
1) le développement du programme avec le mode interactif, grâce à la poursuite de l'exécution au niveau LHN,
2) il n'y a pas à faire de compilation.

L'architecture traditionnelle de von Neumann quant à elle, est caractérisée par : les multiples couches des logiciels complexes, l'exécution séquentielle des instructions primitives, la compilation complexe avec la phase d'optimisation et de génération des instructions, et la difficulté de poursuivre l'exécution du LHN, etc ...

Le LM idéal peut être obtenu par la conception d'un langage qui satisfasse les "critères de Flynn" introduits au § 2.3. Ainsi, la machine LHN peut-elle accepter deux modes d'exécution :

a) sans compilation

b) avec compilation simple réalisée soit par logiciel, soit par matériel, mais si simple que l'on peut considérer comme l'exécution directe de LHN.

La question se pose alors de savoir comment le LM peut être défini ? Le chapitre suivant définira le LM qui satisfait ces conditions.

CHAPITRE 3

ISOMORPHISME DES LANGAGES

3.1. INTRODUCTION

En première approximation, pour fixer les idées dans ce chapitre, on considère deux langages dont les caractéristiques sont presque identiques, si bien que la traduction peut être faite d'une manière simple et systématique.

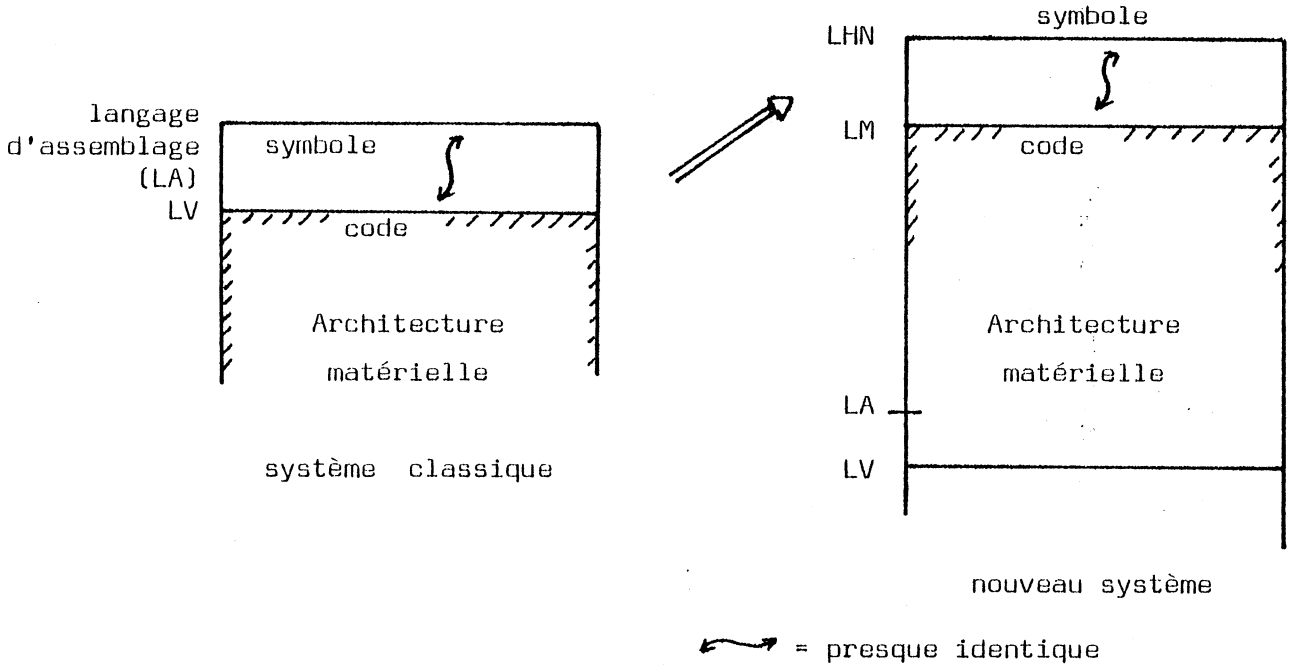


Figure 9 - Le nouveau LM

La figure 9 compare les deux langages. Le langage d'assemblage est la représentation symbolique du LM de von Neumann. Ces deux langages ont la même structure, les mêmes attributs, et les mêmes espaces de données. Ceci implique que la poursuite de l'exécution et la traduction inverse sont simplement réalisables. Pourtant on ne construit pas l'architecture qui exécute directement le lantage d'assemblage puisque LM est considéré dans ce sens très proche du langage d'assemblage.

La même analogie peut être faite dans le cas du LHN. Au lieu de construire l'architecture qui exécute directement le LHN, on peut concevoir un LM qui a le même espace de données et les mêmes instructions que le LHN. La poursuite d'exécution et la traduction inverse peuvent se faire comme dans le cas du langage d'assemblage. Le traducteur est chargé de décoder les identificateurs sans passer par la phase d'optimisation et de génération de codes.

Le paragraphe qui suit propose le LM idéal pour la machine LHN.

3.2. LANGAGE ISOMORPHE

Soit T : L1 → L2 un traducteur qui traduit le programme source L1 en un programme objet L2. Intuitivement, leslangages de programmation L1 et L2 sont isomorphes si toute unité de L1 est bijectivement en correspondance avec une unité de L2 et si les opérateurs des deux langages sont également dans une telle correspondance.

Une définition formelle de cet isomorphisme est la suivante : considérons plus particulièrement deux groupes, G muni de l'opération T, et G' muni de l'opération T', tels qu'il existe entre eux une correspondance biunivoque. Il faut, pour que cette correspondance soit isomorphe, qu'elle soit compatible avec les lois de composition de G et G'. Rassemblons par une double flèche les éléments qui se correspondent ($x \rightleftarrows x'$, par exemple). Nous devons donc avoir :

$$\begin{array}{l} x \rightleftarrows x' \\ y \rightleftarrows y' \end{array} \qquad x \; T \; y \rightleftarrows x' \; T' \; y'$$

Afin d'établir l'isomorphisme des langages, on adopte une grammaire des langages à la place des programmes écrits dans ces langages.

Un langage de programme généré par une grammaire $G = (V_N, V_T, P, S)$, dénoté L(G), est un ensemble de chaînes tel que :

$$L(G) = \{w \mid w \in V_T^* \text{ et } S \overset{*}{\Rightarrow} w\}$$

où V_N, V_T, P, S, sont respectivement les non-terminaux de grammaire, les terminaux, les règles de production et le symbole de départ de la production [15], [31]. Nous restreignons ici l'ensemble des terminaux V_T aux unités lexicales : par exemple, mots réservés, identificateurs, constantes, etc ..

Il s'agit d'abord de construire la structure mathématique de cette grammaire. Soit D un ensemble de structures de données, i.e. l'espace de données généré par :

a) grammaire (la partie de définition du type des données)

b) et les types implicites de données simples (entier, réel, booléen, etc ..).

On modifie cet ensemble D pour qu'il soit un semi-groupe. Soit O le sous-ensemble de V_T contenant les terminaux des opérateurs. Pour que D soit un semi-groupe un opérateur $o \in O$ est défini de la manière suivante :

Etant donnés deux opérandes d1 et d2 qui sont les éléments de D, d1 o d2 signifie un des trois opérateurs suivants : (i) surcharge, (ii) conversion de type, ou (iii) indéfini. La surcharge signifie que l'extension de la signification de l'opérateur est donnée lorsque les mêmes types des opérandes d1 et d2 changent ("non typés" dans [3]) [24]. La conversion de type est apparue lorsque les types d1 et d2 sont différents. Si l'opérateur o n'est pas applicable à d1 ou d2, l'opération binaire de cet opérateur est indéfinie.

On peut alors établir le semi-groupe D en ajoutant un élément redondant d ϵ D qui est défini comme suit :
a) d1 o d2 = d ϵ D lorsque l'opérateur o est indéfini pour d1 ou d2,
b) d1 o d = d o d1 = d pour tout d1 ϵ D.
On obtient donc la relation d1 o d2 ϵ D pour tout d1, d2 ϵ D et tout opérateur o ϵ O, qui conduit au semigroupe D.

Il s'agit ensuite d'établir la notion de proximité pour la comparaison entre les deux langages. Un homomorphisme d'un ensemble X dans un ensemble Y, où X et Y sont de la même catégorie, est une fonction f : X $\rightarrow$ Y dont la loi de décomposition : f(a o b) = f(a) * f(b), est toujours établie pour a, b ϵ X et pour les opérateurs o de X et * de Y [30]. Lorsque f est bijective, on dit que f est un isomorphisme.

Sur la base de ces définitions, l'isomorphisme des langages est défini comme suit. Soit L(G1) et L(G2) les langages de programmation générés respectivement par les grammaires G1 et G2. Soit D1, O1, Vt1 et D2, O2, Vt2 les ensembles correspondant respectivement à L(G1) et L(G2). Vt'1 et Vt'2 sont les ensembles des terminaux groupés selon les constituants structuraux des langages correspondants.

Les langages L1 et L2 sont isomorphes (L1 $\tilde{=}$ L2) si :
1) D1 et D2 sont isomorphes par rapport aux opérateurs de O1 et O2,
2) Vt'1 et Vt'2 sont bijectives.

L'isomorphisme de D1 et D2 signifie qu'il existe une fonction (traduction)

T : D1 $\rightarrow$ D2

qui satisfait les deux conditions :
1) T est un homomorphisme : T(d1 o d2) = T(d1) * T(d2), où d1, d2 ϵ D, o ϵ O1 et * ϵ O2,
2) D1 et D2 sont bijectives.

Malgré la généralité de cette définition, elle peut éventuellement signifier que les deux langages ont les mêmes espaces de données, qui sont manipulés de la même manière par les opérations, et que les instructions exécutables des deux langages se situent sur le même niveau. Donc, la traduction entre les deux langages n'est qu'une transformation permutative des objets.

On verra plus loin que PASCAL et I-PASCAL sont ainsi isomorphes.

3.3. CARACTÉRISTIQUES DES LANGAGES ISOMORPHES

La définition de l'isomorphisme des langages permet au concepteur de choisir le LM de la machine LHN parmi les LHN isomorphes. Si le LM ainsi choisi est un langage codé plutôt qu'un langage symbolique, l'interprétation de ce LM est non seulement libre des charges : de l'analyse lexicale et syntaxique, du décodage des identificateurs et du calcul du programme de contrôle, mais aussi a les mêmes caractéristiques et le même niveau que le LHN en permettant le développement des programmes en mode interactif et la facilité de leur exécution.

La relation d'isomorphisme des langages constitue une relation d'équivalence. De la sorte, l'ensemble des langages peut se partitionner en classes d'équivalence, dénotées $L/\simeq$. Chaque classe est représentée par un langage donné.

Dans le cas des langages isomorphes L1 et L2, le traducteur opère sur le même arbre représentatif. Il n'y a ni génération des instructions de niveau inférieur, ni optimisation des codes.

Les conditions pour concevoir le LM idéal, proposées d'après les critères de Flynn présentés précédemment, sont bien satisfaites par les langages isomorphes.

La couche indépendante décomposée par le critère de l'isomorphisme permet la portabilité entre les langages isomorphes. La connexion de cette couche, entre LHN et LHN isomorphe (LM), est basée sur la correspondance un à un, qui permet la poursuite de l'exécution de LHN en mode interactif ("transparence" pour le critère de Flynn [19] : le même état de transition entre deux langages en laissant le résultat dans la même mémoire).

Parmi les langages dialectaux, par exemple PL/1 : PL/M, PL/1 extension, etc .. la définition de l'isomorphisme ne fonctionne pas, puisqu'il s'agit de "sous-ensembles" ou de "sur-ensembles" qui ne permettent pas la correspondance bijective. Dans la plupart des cas, il est plus pratique de dire qu'un sous-ensemble d'un langage est isomorphe à l'autre langage et vice versa. Il s'agit toujours d'une partie exceptionnelle dépendant de l'implantation sur la machine.

3.4. LA MACHINE LHN ISOMORPHE

On appelle LHN isomorphe un des langages de la classe d'isomorphisme d'un langage LHN.

La machine LHN isomorphe est une machine dont le LM est isomorphe à LHN. Autrement dit, c'est la machine qui exécute directement le LHN isomorphe.

Le traducteur isomorphe est un traducteur dont le langage source et le langage objet sont isomorphes et nous reviendrons sur ce point au chapitre suivant.

CHAPITRE 4

TRADUCTEUR ISOMORPHE

4.1. SÉPARATION PHYSIQUE EN DEUX COUCHES INDÉPENDANTES

Le traducteur isomorphe est défini comme le traducteur dont le langage source et le langage objet sont isomorphes dans le sens de la définition précédemment donnée. Si le langage source est le LHN et le langage objet le LM isomorphe à LHN, ce traducteur est donc le compilateur qui est isomorphe. Nous ne ferons donc pas de distinction entre traducteur et compilateur.

Les fonctions principales du traducteur isomorphe sont :
. analyse lexicale et syntaxique,
. actions sémantiques (conversion de code).

La première fonction inclut l'analyse lexicale, syntaxique et le décodage des identificateurs. Les actions sémantiques sont la génération du langage objet et la conversion de l'unité du langage source en code du langage objet. Puisque les deux langages se situent au même niveau, il n'y a ni fragmentation des données et des instructions, ni étape d'optimisation des codes.

Pour le système d'interprétation du LM, il est plus efficace d'adopter le langage codé (par binaires compacts) plutôt que le langage symbolique (alphanumérique). Les avantages sont [3] :
a) un minimum de volume des instructions de programme,
b) un minimum de bits représentant les valeurs ou structures de LHN,
c) des instructions puissantes avec le maximum de signification sémantique.

Le LHN symbolique est basé sur l'interface avec les utilisateurs, cependant le langage codé est conçu pour la représentation compacte de programmes et pour l'exécution efficace par le système. Si la traduction isomorphe est motivée par l'exécution du LM par l'architecture (compilation isomorphe), la représentation efficace du LM est donc le code binaire compact.

Il y a deux approches possibles pour la construction des compilateurs [32] :
a) approche récursive (approche orientée syntaxe)
b) approche itérative.

La première approche est caractérisée par la récursivité et par la construction systématique à partir de la syntaxe des langages [34]. L'action sémantique est commencée par le parcours descendant et récursif de l'arbre syntaxique depuis le sommet jusqu'aux variables de la grammaire situées sur les feuilles définies par la syntaxe (à partir de maintenant, les variables signifient des non-terminaux de V_N dans la grammaire [15], [31]). L'approche itérative (ou empirique) est une simulation d'un "railyard shunt" (suggéré par Dijkstra) qui joue le rôle de la pile. Cette méthode est basée sur la structure de programme itérative. L'algorithme de l'approche empirique est donc plus efficace que celui de l'approche récursive dont l'inefficacité provient de la récursivité des procédures et de la redondance des variables dupliquées. Pourtant, l'étude de [32] montre que l'algorithme orienté syntaxe peut toujours être transformé en algorithme empirique sans perte de la fonction cohérente.

En supposant que le mécanisme spécial pour appel/retour de procédure soit pris en compte pour améliorer l'efficacité de la récursivité [26], nous adoptons l'approche orientée syntaxe du LHN pour la conception descendante et pour l'approche multilangage [33] en modules matériels, basée sur la décomposition verticale (en n'abordant pas dans cette présentation l'approche itérative dans laquelle l'approche ascendante basée sur la grammaire LR(k) [21], [31], [40], peut être adoptée).

Basé sur l'approche orientée syntaxe du LHN, le traducteur peut être en général décomposé en couches syntaxiques et sémantiques. Afin de permettre la réalisation matérielle, il est souhaitable que la décomposition en couches soit physiquement séparable. Donc, les blocs ainsi décomposés permettent l'implantation par les blocs matériels.

La séparation physique d'un traducteur signifie le détachement des actions sémantiques de la partie syntaxique. Les actions sémantiques étaient implantées sur le module d'analyse syntaxique dans le compilateur classique orienté syntaxe. La figure 10 montre les deux structures des traducteurs.

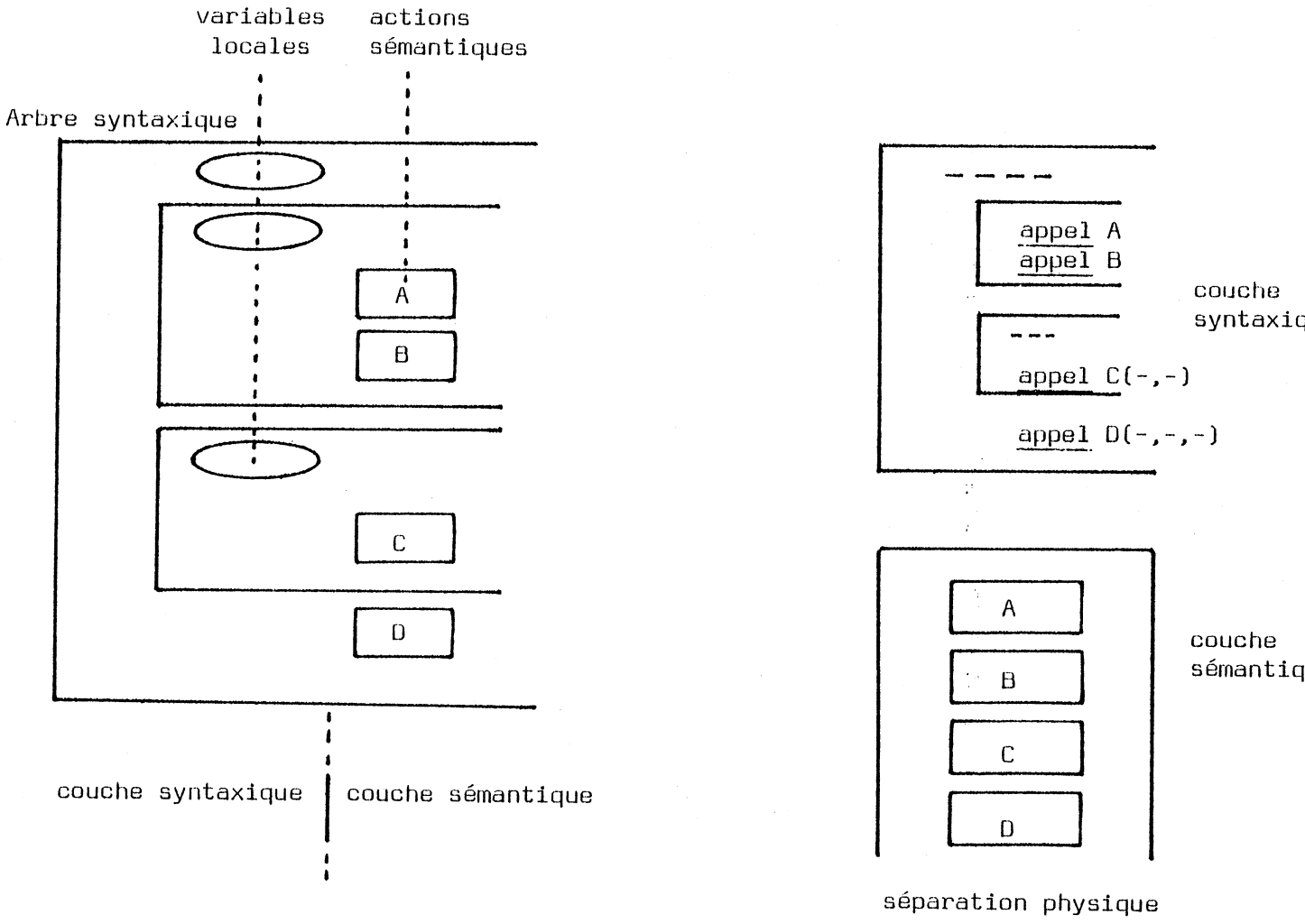


Figure 10 - Compilateur classique et compilateur physiquement séparé

La séparation physique en couches implique la possibilité de remplacement des actions sémantiques par des appels de procédures. Cette possibilité est réalisable si les conditions suivantes sont satisfaites :

a) les variables locales sont limitées et ne sont donc permises que comme les paramètres de procédures sémantiques,

b) les communications entre modules sont basées sur les variables globales.

Ces variables locales ne peuvent pas être annexées dans les variables globales si l'on veut profiter de la récursivité de l'approche orientée syntaxe. La possibilité de la séparation physique est donc dépendante de la nature des variables locales dans le module syntaxique.

Deux sortes de traduction indiquent la nature des variables locales (dans le module syntaxique) :

. traduction isomorphe,

. traduction non-isomorphe.

Dans la traduction non-isomorphe, il y a des fragmentations de données et des instructions du programme source. Cela conduit à avoir de nombreuses variables locales.

Dans la traduction isomorphe, il n'y a pas de fragmentation. Par permutation des objets entre les deux langages, la couche syntaxique n'a qu'un nombre limité de variables locales qui sont admises comme paramètres des procédures sémantiques. Elles apparaissent dans la procédure récursive dont la fonction contient la permutation des objets.

Ces variables locales sont :

a) les pointeurs (pour descripteurs et programme contrôle),

b) les opérateurs (pour transformation de l'expression arithmétique : en notation post-fixée, etc ..).

La figure 11 illustre l'organisation du traducteur isomorphe dans lequel les modules sémantiques sont physiquement séparés (ainsi qu'ils sont indépendants de la "portée des noms" du module syntaxique).

La figure 12 décrit le contexte de la deuxième application des règles de décomposition effectuée à la traduction isomorphe.

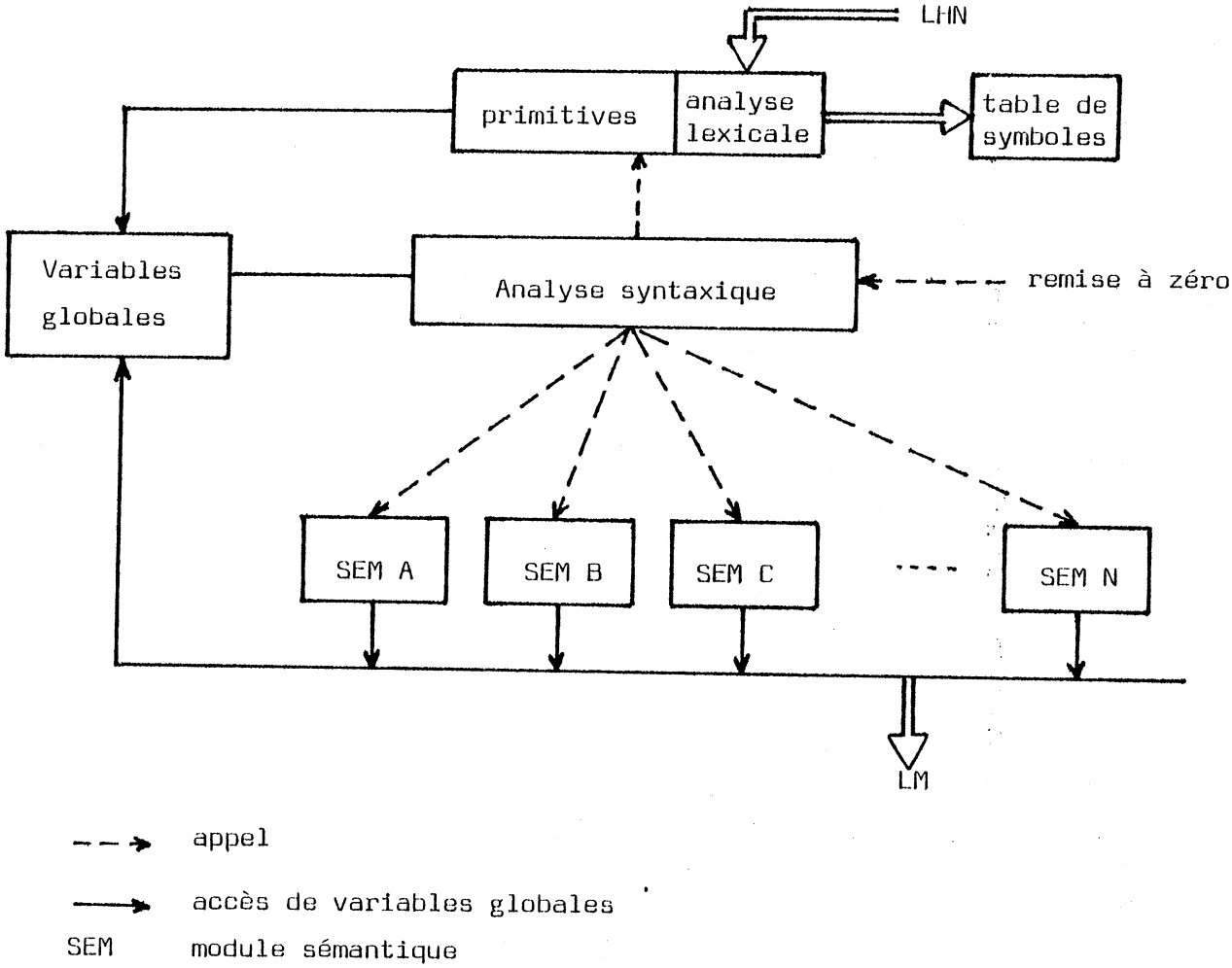


Figure 11 - Traducteur isomorphe physiquement séparé en modules

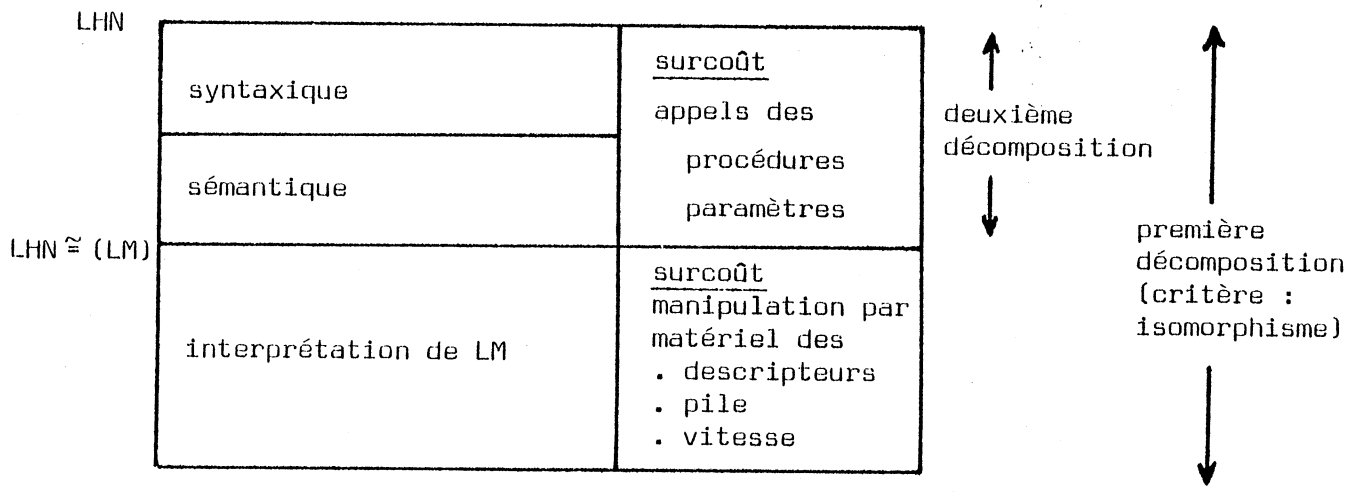


Figure 12 - Décompositions verticales en couches

4.2. TRADUCTEUR ISOMORPHE BASÉ SUR LA SYNTAXE

4.2.1. Introduction

Il existe un certain nombre de compilateurs basés sur l'approche orientée syntaxe de LHN. Etant donnée la syntaxe du contexte libre, la variable (de V_N) à gauche de "::=" constitue le nom de procédure, cependant que la partie droite de la définition constitue le corps de procédure dans lequel les variables des syntaxes sont représentées par des appels d'autres procédures à définir [34], [35].

Cette approche vise à construire le traducteur isomorphe dans un environnement de la séparation physique entre la couche syntaxique et la couche sémantique. La correspondance un à un entre les actions syntaxiques et les actions sémantiques est représentée par les appels de procédures sémantiques dont les modules se situent dans des blocs séparés.

Il existe deux stratégies pour l'approche orientée syntaxe :
1) construire le traducteur basé sur la syntaxe originale de la grammaire du LHN,
2) construire le traducteur basé sur la syntaxe modifiée de la grammaire du LHN.

La première approche semble idéale, puisque une fois que la syntaxe du LHN est conçue, la syntaxe peut être déjà considérée comme la partie de l'analyse syntaxique. La deuxième approche consiste à modifier les règles des syntaxes pour que la grammaire soit déterministe et que les variables intermédiaires ou redondantes de la syntaxe soient supprimées. Le traducteur peut donc être plus efficacement réalisé.

4.2.2. Approche par syntaxe originale

On peut considérer deux méthodes pour l'approche par la syntaxe originale. La première méthode consiste à construire le traducteur basé sur la syntaxe du LHN sans aucune modification et sans aucune addition.

Il y a en général trois sortes de règles de syntaxe dans la grammaire $G = (V_N, V_T, P, S)$ du LHN :

1) La règle dont la définition est la séquence des variables de V_N telle que :
<A> ::= <B> <C> <D> ...
où <A>, <B>, ..., <D> sont les variables des non-terminaux V_N,
peut être représentée par l'algorithme suivant (les programmes ci-dessous ne correspondent pas exactement au LHN donné ; ils expliquent seulement de manière informelle l'algorithme) :

```
procédure A ;
   begin call B ;
         call C ;
         call D ;
           :
   end ;
```

où A est le nom de procédure, et B, C et D sont des procédures définies ailleurs

2) La règle dont la définition est un choix parmi les variables données telle que (appelons les variables <B>, <C>, <D> ... ici les "variables de choix") :
<A> ::= <B> | <C> | <D> ...
correspond à l'algorithme suivant :

```
procédure A
   begin call B ;
         if réussi avec B then return else call C ;
         if réussi avec C then return else call D ;
         :
         if non réussi then return avec l'information d'échec de A ;
   end ;
```

(Cet algorithme peut être représenté par les divers programmes : par exemple, avec la fonction programme de booléen, avec la variable booléenne représentant l'état de réussite d'analyse).

3) La règle qui contient la variable nulle, dénotée par Ø, dans les choix, telle que :
<A> ::= Ø | <B> | <C> | ...
est représentée par le même algorithme que précédemment sauf la dernière ligne remplacée par :

```
return avec l'information de réussite de A
```

qui donne le retour inconditionnel.

Cette méthode avec la syntaxe originale cause pourtant beaucoup de problèmes :

. les règles sont non-déterministes : ainsi toutes les variables possibles dans les cas 2) et 3) doivent être visitées si la variable correspondante n'est pas trouvée. Ceci provoque une complexité énorme et ne garantit pas toujours de réussir à trouver la variable correspondante ;
. la variable nulle (production Ø) provoque la redondance de certaines variables ;
. toute variable intermédiaire est visitée.

L'exécution parallèle par le LM spécialement préparé tel que FORK ... [29], peut être adoptée pour accélérer l'analyse, puisqu'il y a un nombre de parallélisme possible correspondant au choix des règles. Même si la grammaire n'est pas ambigüe, l'inefficacité de cette méthode provient de l'analyse non-déterministe.

La deuxième méthode consiste à ajouter l'élément prévisé, dénoté par la méta-notation [[ ]], à la syntaxe originale devant chaque variable de choix. Ces éléments préfixés permettent l'analyse déterministe.

La règle avec les variables de choix :

<A> ::= <B>| <C> | <D> ...

devient donc :

<A> ::= [[tête 1]] <B> |[[tête 2]] <C> |[[tête 3]] <D> | ...

où [[tête i]] signifie un ensemble d'unités lexicales (terminaux de V_T) dont la i-ème variable de choix commence lors de la dérivation (d'après [31], $FIRST_K$(<B>), etc ..).

Les éléments préfixés n'entrant pas dans les procédures de variables, l'analyse est donc déterministe. Dans le cas de la variable nulle, il n'y a pas de parcours redondant. Pourtant, malgré l'analyse déterministe, les problèmes suivants mettent en évidence une certaine inefficacité :

. les variables intermédiaires restent et provoquent une certaine complexité ;
. les jetons dans les têtes de l'élément préfixé ne sont pas toujours uniques, si bien qu'il faut revoir l'unité lexicale suivante pour la décision de dérivation ; il s'agit en fait de l'analyse de LL(k) où $k \geq 2$;
. la récursivité à gauche ou à droite doit être modifiée selon l'analyse donnée.

L'approche par la syntaxe originale n'est donc pas pratique si la syntaxe de la grammaire n'est pas simple.

4.2.3. Approche par syntaxe modifiée

4.2.3.1. Première étape et grammaire LL(1)

En modifiant la syntaxe du LHN, on peut obtenir la syntaxe modifiée telle que chaque variable de choix commence toujours par un jeton lexical (unité lexicale) de V_T. Ainsi, l'analyseur syntaxique avance-t-il le pointeur d'entrée du LHN sur le jeton suivant avant d'entrer dans la procédure de variable de choix. (Note : dans le système de l'élément préfixé, dénoté par [[ ]], ceci fonctionne seulement comme le déterminateur de choix sans avancement du jeton suivant). Afin d'avoir la syntaxe modifiée déterministe avec un jeton au maximum, il faut donc que la grammaire modifiée soit LL(1) (ou LR(1)). (Dans cette présentation du traducteur orienté syntaxe, c'est la grammaire LL(1) qui est choisie).

La grammaire LL(k) est définie comme suit [31]. Soit $G = (V_N, V_T, P, S)$ la grammaire contexte libre. G est LL(k) pour un entier fixé k, s'il existe toujours la dérivation gauche extrême (lm) :

1) $S \overset{*}{\underset{lm}{\Rightarrow}} wA\alpha \underset{lm}{\Rightarrow} w\beta\alpha \overset{*}{\underset{lm}{\Rightarrow}} wx$

2) $S \overset{*}{\underset{lm}{\Rightarrow}} wA\alpha \underset{lm}{\Rightarrow} w\gamma\alpha \overset{*}{\underset{lm}{\Rightarrow}} wy$

tel que FIRSTk(x) = FIRSTk(y), signifie que $\beta = \gamma$. (Ici, A est variable de V_N, α, β, γ, w, x, et y sont les chaînes de V_N et V_T et $\overset{*}{\underset{lm}{\Rightarrow}}$ signifie dérivation multiple).

La définition de FIRSTk(α) est la suitante :

$$FIRSTk(\alpha) = \{x \mid \alpha \overset{*}{\underset{lm}{\Rightarrow}} x\beta \text{ et } |x| = k \text{ ou } \alpha \overset{*}{\underset{lm}{\Rightarrow}} x \text{ et } |x| < k\}$$

Les critères de modification sont donc les suivants :

1) pour chaque choix de règle, un jeton lexical précède la variable de choix ;
2) la syntaxe est LL(1)
3) les règles de choix sont ordinales (l'analyse se passe de la variable de choix gauche vers la variable de choix droite).

Dans la plupart des cas, il est difficile d'avoir une grammaire LL(1) sans décodage des identificateurs. Sans décoder les identificateurs, la grammaire devient soit LL(k), avec k ≥ 2, soit ambigüe. Il faut donc modifier la syntaxe sous forme de LL(1) par :

1) préparer la procédure spéciale dont la fonction est le décodage des identificateurs, ainsi que les jetons des identificateurs apparus distincts dans la syntaxe (il y a aussi possibilité de décoder avec la préparation des mots réservés correspondant aux identificatuers distincts (cf. § 6.3.1.1.)) ;
2) modifier la syntaxe de la forme récurrente gauche en une forme récurrente droite.

Un exemple de modification de la syntaxe de PASCAL sous une forme LL(1) est donné au § 6.3.1., où l'on adopte la procédure spéciale, appelée IDKINDS, pour le décodage des identificateurs.

En supposant que la syntaxe soit modifiée en LL(1), la première étape de la modification de syntaxe est la suivante :

1) la règle avec les variables consécutives
   <A> ::= <B> <C> α
   n'est pas modifiée, où α est une chaîne (des variables V_N et des terminaux V_T). L'algorithme est le même que le 1) du § 4.2.2.

2) La règle avec les variables de choix :
   <A> ::= <B> β | <C> γ | ... | <D> δ
   où β, γ, δ sont les chaînes, devient :
   <A> ::= b <B'> β | c <C'> γ|...| <D> δ ou d <D'> δ
   où b, c et d sont les jetons de V_T qui sont dérivés à gauche par <B>, <C> et <D> respectivement. Donc,
   b = FIRST1(<B>), c = FIRST1(<C>), d = FIRST1(<D>).
   Les nouvelles variables <B'>, <C'>, et <D'> sont les chaînes apparues par extraction du jeton. Notons dans cette règle que :
   - les jetons b, c et d sont distincts par la modification de LL(1) ;
   - le dernier choix n'est que visité en cas d'échec de l'analyse avec les variables de choix précédentes (donc, <D>δ est admissible).

3) La règle qui contient la variable nulle dans un premier choix tel que :

<A> ::= Ø | <B> β | ... | <D>δ

est modifiée en déplaçant Ø sur le dernier choix tel que :

<A> ::= b <B'> β|... | d <D'> δ | Ø

Dans ce cas, la variable <D> n'est plus le dernier choix, elle devient d <D'>.

4) La règle définissant la variable itérative, notée par

<A> ::= {<B>}

est modifiée ainsi :

<A> ::= b <B'> { b <B'>} | Ø

pour être déterministe.

L'algorithme correspondant au cas 2) est le suivant, où w signifie un terminal de jeton à analyser et NEXTTOKEN est l'avancement du pointeur d'entrée du LHN sur le jeton suivant (ici le dernier choix est d <D'>δ) :

```
procédure A ;
   begin if w not in (b, c, ..., d) then ERREUR                    (1)
         else case w of b : begin NEXTTOKEN ; call B' ; β end ;
                        c : begin NEXTTOKEN ; call C' ; γ end ;
                        :
                        d : begin NEXTTOKEN ; call D' ; δ end ;    (2)
         end
end ;
```

Si le dernier choix est <D>δ , la ligne (1) devient :

if w not in (b, c, ...) then begin call D ; δ end ;

en retardant l'analyse par appel de la procédure D. (La ligne (2) est supprimée dans ce cas).

Si le dernier choix est la variable nulle (cas 3), la ligne (1) devient l'action nulle telle que :

if w not in (b, c, ...) then begin end.

La répétition de la variable du cas 4) est programmée par :

while w = b do begin NEXTTOKEN ; call B' end.

L'exemple qui suit illustre la modification de la syntaxe de PASCAL dans le cas de la partie de déclaration de procédure et fonction, dont la modification complète est donnée dans l'annexe A.3. :

<proc. & fct. dec. p> ::= procedure <proc.d> ; <proc. & fct. dec. p> |
function <fct.d> ; <proc. & fct. dec. p> | ∅ (3)

<proc.d> ::= ident <proc. heading> <block>

<proc. heading> ::= ; | (<formal para. sec.>{<formal para. sec>}) ;

<fct.d> ::= ident <fct. heading> <block>

<fct. heading> ::= : <result type> ; | (<formal para. sec> {;
<formal para. sec>}) : <result type> ;

La modification de (3) correspond à la syntaxe originale telle que :

<proc. & fct. dec. p> ::= {<proc. or fct. dec> ;}

<proc. or fct. dec> ::= <proc. dec.> | <fct. dec.>

La justification est simple. Supposons que A soit <proc. & fct. dec. p>, B soit <proc. or fct. dec>, C soit <proc. dec.> et D soit <fct. dec.>. Alors,

A ::= {B}, B ::= C | D

deviennent

A ::= B{B} | ∅

→ A ::= BA | ∅

→ A ::= (C | D) A | ∅

→ A ::= CA | DA | ∅

4.2.3.2. Deuxième étape

La deuxième étape de la modification syntaxique consiste à fusionner les règles de production redondantes et à optimiser la syntaxe.

1) La syntaxe avec production nulle telle que :

<A> ::= <B> <C> α

<B> ::= b <B'> | ∅

<C> ::= c <C'> | ∅

est modifiée en supprimant la production nulle et devient :

<A> ::= b <B'> 11 | 12

11 ::= c <C'> 12 | 12 ...

où 11 et 12 sont les étiquettes syntaxiques qui indiquent la production suivante. Dans la programmation, les étiquettes syntaxiques ont disparu puisqu'elles fonctionnent seulement dans la grammaire pour la définition temporaire.

L'algorithme correspondant est donc :

```
procedure A ;
    begin if w = b then begin NEXTTOKEN ; call B' end ;
          if x = c then begin NEXTTOKEN ; call C' end ;
          :
end ;
```

Par exemple, la définition de block dans PASCAL est optimisée par la suppression des productions nulles (cf. annexe A.4) :

<block> ::= label <label d.> 11 | 11

11 ::= const <const d.> 12 | 12

12 ::= type <type d.> 13 | 13

13 ::= var <var. d.> 14 | 14

14 ::= procedure <proc. d.> ; 14 | function <fct. d> ; 14 | 15

15 ::= <compound st>.

2) Pour les règles redondantes telles que :

<A> ::= <B>, <B> ::= <C>, <C> ::= <D>, etc ..

elles sont simplifiées par une production :

<A> ::= <D>.

4.2.3.3. Troisième étape

La modification de la syntaxe dans les deux étapes précédentes fait parfois apparaître des syntaxes modifiées inefficaces. On verra dans les annexes A.3 et A.4 que la partie <simple type> est doublement définie à cause de la modification correspondant au cas 2 de la première étape.

Afin d'éviter ce phénomène, on adopte l'élément préfixé, dénoté par [[ ]], de la même manière que pour la deuxième méthode du § 4.2.2. (approche par syntaxe originale), si cette nécessité est exigée par l'optimisation de la syntaxe modifiée.

La partie double de la définition <type> peut donc être éliminée par l'élément préfixé suivant. (Ici, IDKINDS est la procédure de décodage des identificateurs pour permettre l'analyse LL(1), où les méta-parenthèses (( )) signifient le domaine où IDKINDS est effectué)

```
<type> ::= [[ (ident) ]] <simple type> | ↑ ident | packed 18 | 18
    18 ::= array[<simple type> {, <simple type>}] of <type> |
           record <file list> end | set of <simple type> | file of <type>
<simple type> ::= ( ident {, ident }) | IDKINDS (( ident (CONST)..
                  ident )) | ident (TYPES)
```

4.3. CONSTRUCTION DU TRADUCTEUR ISOMORPHE

Par la décomposition de la traduction isomorphe, on fait apparaître les deux couches et un surcoût :

- la couche syntaxique,
- les actions sémantiques,
- le surcoût (variable globale, primitive, initialisation, etc ..).

Les étapes de la construction de ces blocs sont constituées par les séquences suivantes :

1) Etablissement de la syntaxe originale :
étant donné un LHN, la syntaxe est définie d'une manière non-procédurale sous la forme d'un contexte libre.

2) Modification de la syntaxe en LL(1) :
la syntaxe de la grammaire est modifiée pour la rendre déterministe.

3) Modification de la syntaxe en trois étapes :
étant donnée une règle de choix représentée par le type non-procédural, il devient procédural par ordinalité.

4) Rangement de la syntaxe globale dans le type procédural :
c'est une étape qui dépend de l'outil-langage de développement. Si c'est le langage PASCAL, toutes les procédures doivent être déclarées avant le corps de procédure. Ainsi la syntaxe doit-elle être rangée selon la relation de blocs-imbrication (FORWARD aussi, si nécessaire). La syntaxe globale devient donc de type procédural si le langage utilisé pour le développement du traducteur est de type procédural.

5) Construction d'une couche syntaxique :
dans le cadre des algorithmes expliqués au § 4.2.3., la syntaxe modifiée est directement programmable. Chaque fois que l'on a besoin de l'action sémantique, l'appel de la procédure sémantique est implanté dans l'analyse syntaxique. La programmation basée sur la syntaxe permet d'adopter la méthode de la programmation structurée sans instruction "go to". Les variables locales sont établies lorsqu'il est nécessaire de sauvegarder les pointeurs ou les opérateurs dans un environnement de récurrence syntaxique.

6) Etablissement des variables locales et des structures de données :
les données nécessaires pour les actions sémantiques sont préparées (données pour le décodage des identificateurs, structures de descripteurs, code de LM, structures intermédiaires, etc ..).

7) Construction des procédures primitives :

. primitives pour les structures de la programmation de l'analyse syntaxique,

. analyse lexicale, test d'erreurs, conversion de constante, etc ..

. création des données initialisées.

8) Construction des actions sémantiques :

les appels de procédures sémantiques implantées dans l'analyse syntaxique sont définis en fonction des étapes 6 et 7. La création des descripteurs et des conversions des unités lexicales en LM sont les actions principales de cette étape.

9) Initialisation :

avant de traduire, les structures de données sont initialisées pour mettre l'exécution en état initial.

Tous les détails de ces étapes se retrouvent dans les exemples du chapitre 6 dans lequel la construction du traducteur isomorphe de PASCAL est présentée.

DEUXIEME PARTIE

CHAPITRE 5

PASCAL ISOMORPHE ET I-PASCAL

5.1. INTRODUCTION

Dans ce chapitre, nous comparons le langage PASCAL au langage I-PASCAL dans le sens de l'isomorphisme des langages.

Pour un LHN donné, il y a plusieurs langages intermédiaires possibles dans la hiérarchie définie précédemment. Pour les structures de ces langages intermédiaires, il y a cinq candidats possibles [37] :

- format 3-opérandes (y compris N-opérandes où N > 3) ;
- format 2-opérandes ;
- format 1-opérande ;
- notation Polonaise ;
- arbre programme.

Parmi ces candidats, seuls la notation Polonaise et l'arbre programme peuvent être adoptés comme format du LHN isomorphe. Par la définition de l'isomorphisme des langages, chaque jeton lexical du LHN doit être en correspondance biunivoque avec les instructions du langage objet. Dans le cas des formats N-opérandes (où $N \geq 1$), il y a plus de deux jetons dans une instruction. Cela ne permet pas la correspondance biunivoque.

Pour les langages intermédiaires de PASCAL, la notation Polonaise est adoptée pour la manipulation de la pile. Il y a pour les langages intermédiaires de PASCAL : P-code de ETH, P-code de Microengine [10], les variations de P-code (U-code) [14], LM de la machine à pile, etc .. [8], [38]. Par contre, en [37] le format 3-opérandes est adopté pour représenter le LHN intermédiaire dans un environnement de microprogrammation.

Le langage I-PASCAL est le LM de la machine PASC-HLL qui a été construite à Grenoble, basée sur la conception de multiprocesseurs fonctionnels spécialisés dans l'exécution du langage PASCAL [2], [3]. Le format de l'instruction de I-PASCAL est la notation Polonaise. Il existe un champ opérande pour des instructions de référence, mais les instructions opératives n'ont pas de champ d'opérandes (cf. Annexe B.2.).

Un langage sous forme d'arbre programme est bien adapté à la représentation du LHN isomorphe. La machine de la structure de liste peut être conçue pour l'interprétation efficace du LHN [39]. Cette forme permet aussi la structure interne de la traduction de PASCAL.

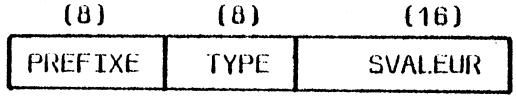
Dans ce chapitre la parenté entre PASCAL et I-PASCAL sera mesurée par le critère de l'isomorphisme.

5.2. DÉMONSTRATION DE L'ISOMORPHISME ENTRE PASCAL ET I-PASCAL

5.2.1. Espace de données

I-PASCAL adopte un système de descripteurs pour représenter l'espace de données [3]. L'accès des variables est effectué par la référence au descripteur dans lequel les informations concernant la structure et le mode d'accès des variables sont décrits.

Deux sortes de descripteurs sont présentés : descripteur de variables et descripteur de type. Le format du descripteur de variables a la structure suivante [3] :



Le PREFIXE est le champ qui décrit la catégorie de la variable.
Le TYPE est le pointeur destiné au descripteur de type.
Le SVALEUR contient soit la valeur, soit l'adresse de la valeur.

L'espace des données de PASCAL est constitué des deux ensembles suivants :
1) type implicite de données,
2) type explicite de données.

Le type implicite de donnée est la donnée simple non-structurée. Ainsi, le programmeur déclare-t-il les variables sans définition de ce type, puisqu'il est déjà implicitement reconnu par le traducteur. L'entier, réel, Booléen, etc .. sont les exemples de ce genre.
Le type explicite de donnée de PASCAL est créé par la syntaxe <type>. Il est défini par le programmeur et est basé sur la dérivation à partir de la syntaxe <type>.

Afin de démontrer l'isomorphisme de l'espace des données entre PASCAL et I-PASCAL, il faut montrer :

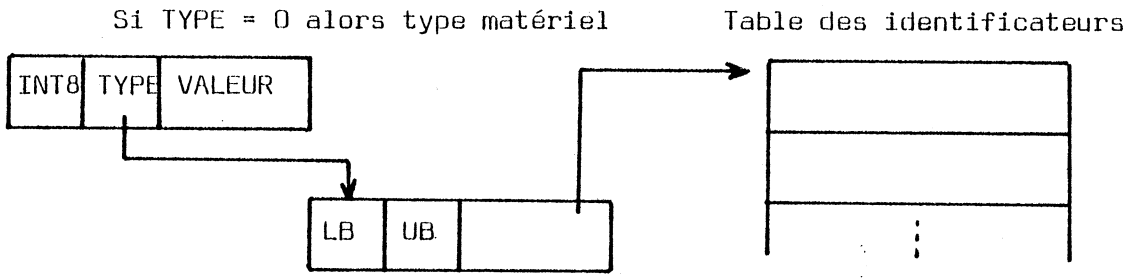
1) la correspondance biunivoque entre la syntaxe <type> de PASCAL et les descripteurs de I-PASCAL (y compris les types implicites),
2) l'homomorphisme entre deux espaces de données.

La syntaxe de PASCAL pour <type> est définie soit par <simple type>, soit par <structured type>, soit par <pointer type> [36].

Le <simple type> est redéfini par <scalar type>, <subrange type>, et/ou <type identifier>. Le <type identifier> est le type qui est défini par un autre <type>, il est donc absorbé dans une autre structure de type.
Le <scalar type> est constitué par la séquence d'identificateurs :

<scalar type> ::= (ident {, ident})

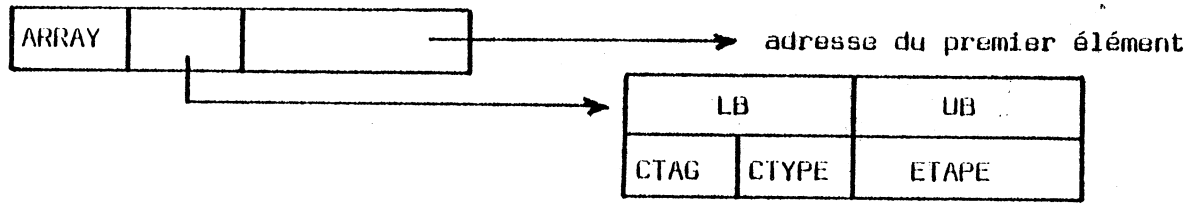
Le descripteur correspondant à ce <scalar type> a la structure suivante :



Les éléments scalaires représentés par les identificateurs, sont codés en entier avec la borne exprimée par LB (minimum) et UB (maximum). La table des identificateurs est préparée en option pour la référence symbolique. Le type <scalaire> peut donc être représenté par le descripteur avec la correspondance biunivoque.

<subrange type> ::= <constant> ... <constant> est exprimé dans un descripteur par les deux bornes valeur supérieure et valeur inférieure. Il y a plusieurs descripteurs possibles pour ce type, selon la taille de la valeur. Par exemple, pour un type implicite d'entier, quatre représentations de descripteurs existent dans I-PASCAL : 8, 16, 32 et 64 bits d'entier. Ils apparaissent comme conséquence de la partie représentative dépendant de la machine. En négligeant cette partie (en traitant 4 sortes d'entiers comme un type d'entier), les descripteurs avec deux valeurs (UB et LB) représentent donc le type <subrange type .

Pour <structured type> on a quatre types : <array type>, <record type>, <set type> et <file type>.
La configuration du descripteur qui représente la structure du <type matrice> est la suivante :



Le <index type>de la syntaxe originale est simplifié dans I-PASCAL par le <subrange type>. La syntaxe correspondante à ce descripteur est donc :

<array type> ::= array [<subrange type>]of <component type>.

La récursivité de <component type> peut être réalisée dans le descripteur par le pointeur CTYPE qui définit récursivement un autre descripteur de type. La structure de <array type> est donc en correspondance biunivoque avec celle du descripteur, à condition que le <index type> soit simplifié par le <subrange type>.

Le <record type> est constitué d'éléments qui sont de types différents. (cf. Annexe A.1. pour la syntaxe de <record type>. Le descripteur de <record type> est organisé par la combinaison des descripteurs :
1) tête d'enregistrement,
2) variant descripteur,
3) champ du descripteur [3].
La configuration détaillée se trouve en [3]. L'utilisation combinée de ces descripteurs permet le même espace de données dans <record type> de PASCAL que dans celui de I-PASCAL. La répétition de <record section> et la récursivité de <type> peuvent être construites par la répétition de champ de descripteurs dans lequel F.TAG représente la récursivité de <type> . La répétition des <variant part> est répalisé par les descripteurs variant. Les parties <field list> et <variant part> qui sont récursivement utilisées pour la définition de <variant part> sont représentés par les champ-descripteurs.

Les types, <set type>, , <file type> et <pointer type> sont respectivement représentés par les descripteurs : indicateur de puissance, fichier et pointeur, dont la récursivité est représentée par un champ donné du descripteur [3] (cf. annexe B.1).

La correspondance biunivoque est donc obtenue par :

a) la structure de syntaxe PASCAL est représentée par le format du descripteur I-PASCAL ;

b) la récursivité syntaxique est réalisée dans le descripteur par le pointeur indiquant un autre descripteur.

Le format et le pointeur du descripteur permettent la création de l'espace des données de I-PASCAL qui est en correspondance biunivoque avec celui de PASCAL.

Pour les types implicites de I-PASCAL, il y a plusieurs sortes de descripteurs, en raison de la nature limitée de la mémoire. L'utilisation de descripteurs est donc dépendante de la taille des données à manipuler. Le type entier a 8, 16, 32, 64 bits de descripteur, le réel en a 32, 64 bits, enfin le Booléen et la chaîne de caractères ont respectivement deux sortes de descripteurs. Pourtant, l'opérateur de I-PASCAL, caractérisé par "non typé" ("surcharge" dans § 3.2), est appliqué non seulement aux différents types, mais aux différents descripteurs pour un type implicite donné. Autrement dit, il ne distingue pas entre plusieurs sortes de descripteurs d'entiers.

L'espace de données de Pascal et celui de I-PASCAL sont donc biunivoques si on néglige :

. la simplification de <index type> dans le type <array type>,

. plusieurs descripteurs pour un type implicite.

L'homomorphisme entre PASCAL et I-PASCAL signifie qu'ils ont la même interprétation sémantique des opérateurs. L'opération de I-PASCAL est conçue pour avoir la même signification que celle de PASCAL en satisfaisant la surcharge, la conversion ou l'opération indéfinie. La traduction T entre deux langages constitue donc l'homomorphisme de cette façon :

$$T(d_1 \circ d_2) = T(d_1) * T(d_2)$$

où d_1 et d_2 sont des structures de données de PASCAL ($T(d_1)$ et $T(d_2)$ sont les descripteurs correspondants de I-PASCAL), et où $\circ$ et $*$ sont respectivement des opérateurs correspondants de PASCAL et de I-PASCAL.

5.2.2. Ensemble de terminaux

L'ensemble de terminaux V'_T contient les opérateurs et les terminaux groupés de V_T. On va donc montrer la correspondance biunivoque entre V'_T de PASCAL et V'_T de I-PASCAL.

Les opérateurs de I-PASCAL correspondant à ceux de PASCAL sont les suivants :

PASCAL :	+	-	*	/	div	mod	=	<	>	≥	≤	not	or	and	in
I-PASCAL :	ADD	SUB	MULT	DIV	IDIV	MOD	EQ	NEQ	GT	LT	GE	LE	NOT	OR	AND	IN
	ensemble															
	arithmétique						relationnel						logique			ensemble

La correspondance biunivoque est établie entre deux ensembles d'opérateurs.

I-PASCAL offre des opérateurs supplémentaires qui ne sont pas standard dans PASCAL. Ces opérateurs sont prévus pour l'optimisation des expressions simples dans lesquelles zéro et un sont des opérandes. Ces opérateurs supplémentaires de I-PASCAL sont groupés dans trois catégories :

a) affectation avec incrémentation ou décrémentation de 1, 0, ou n :
   INCV(V), DECV(V), CLR(V), SET(V), INCA, DECA, CLRA, SETA, ASSI(N), INCAI(N), DECAI(N) ;

b) opérateurs relationnels avec les opérandes de valeur 1 ou 0 :
   EQ_0, LT_0, LE_0, NEQ_0, GE_0, GT_0,
   EQ_1, LT_1, LE_1, NEQ_1, GE_1, GT_1,

c) opérateurs logiques :
   XOR, XAND, NAND, NOR, XNOR, XNAND.

Les catégories a) et b) signifient que les opérateurs ont des instructions de format 1-opérande. L'isomorphisme ne peut donc pas être établi avec les opérateurs supplémentaires.

Afin d'établir la bijection entre les ensembles d'opérateurs, il y a deux stratégies possibles :

1) négliger les opérateurs supplémentaires de I-PASCAL en n'adoptant que les opérateurs standard,
2) étendre le langage PASCAL en adoptant les opérateurs supplémentaires comme des fonctions standard.

La première méthode consiste à prendre le sous-ensemble des opérateurs de I-PASCAL pour que les deux ensembles des opérateurs soient bijectifs. Dans la deuxième méthode, les opérateurs supplémentaires sont considérés comme des fonctions standard de PASCAL, qui sont réalisées par l'architecture matérielle. I-PASCAL adopte les fonctions standard de PASCAL telles que : ABS, SQR, ODD, TRUNC, INT, CMR, SUCC, PRED, EOF, NEW, CORREC, DISPOSE, PACK, UNPACK, etc .. ainsi que les instructions interprétables par la machine plutôt que les sous-programmes de la bibliothèque [3]. Cela rejoint l'approche horizontale de la machine LHN. La deuxième stratégie implique donc que les opérateurs supplémentaires sont considérés de la même façon que l'approche horizontale.

En groupant les terminaux de contrôle de programme, on peut ranger les structures correspondantes entre PASCAL et I-PASCAL. La table suivante indique ce regroupemen

PASCAL :	if then else	while do	loop exitif end
I-PASCAL :	EXIT (THEN / NEHT) ELSE	WHILE (LOOP,ENDLOOP)	LOOP EXITIF ENDLOOP

— non standard —

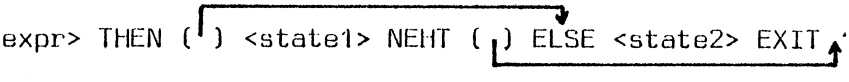
for to downto do	repeat until	case of end	with do
FORUP ROFUP FORDOWN ROFDOWN	LOOP UNTIL	CASE OF FO	WITH(+n) WITH(-n)

go to
(GOTO,EXITFOR)

La structure de contrôle du programme I-PASCAL est pareille à la notation post-fixée. Ce contrôle de programme est structuré par les pointeurs (le calcul des positions de structure est donc fait lors de la compilation). Par exemple, le contrôle du programme :

if <expr> then <state1> else <state2>

est représenté dans PASCAL par :



Les instructions THEN et NEHT fonctionnent comme l'instruction "go to". Le temps de fixation de structure doit donc être établi pendant la traduction. L'imbrication structurelle de contrôle de programme (par la récursivité <statement>) fait apparaître la structure des pointeurs imbriqués. Par conséquent, les variables locales pour pointeurs doivent être préparées dans la couche syntaxique du traducteur.

Dans la définition de l'isomorphisme des langages, on ne distingue pas le temps de fixation de structure. Il s'agit seulement de la correspondance biunivoque entre terminaux groupés de PASCAL et ceux de I-PASCAL. La décision du temps de fixation est un problème de compromis entre l'efficacité et la flexibilité.

Les terminaux de I-PASCAL concernant l'accès aux variables sont groupés par la correspondance bijective avec PASCAL. Voici un exemple d'accès et la table correspondante :

PASCAL : T[I] . A ↑[J] . B
I-PASCAL : REF T, REF I, INDEX, FIELD A, POINT, REF J, INDEX, FIELD B

PASCAL :	[ ] 1	.	↑	,	..	[ ] 2	:=
I-PASCAL :	INDEX	FIELD	POINT	ELEM	INTER	ZERO-SET	ASSA
	'tableau'			ensemble			

La table correspondante pour l'accès de variables, procédures, et paramètres, est la suivante pour les terminaux groupés :

PASCAL :	variable identificateur	appel procédure	appel function	paramètre ,
I-PASCAL :	REF	CALL, ENTER	CALLF, ENTER	PARAM

Pour les terminaux concernant la structure de programme, la correspondance est établie comme suit :

PASCAL :	procedure begin end	function begin end	program const type var
I-PASCAL :	PROCEDURE (descripteur) RETURN	FUNCTION (descripteur) RETURNF	module exécutable (table des segments)

On peut donc obtenir la correspondance biunivoque entre V'_T de PASCAL et V'_T de I-PASCAL.

5.3. SOUS-ENSEMBLE DE I-PASCAL ISOMORPHE

L'isomorphisme exact entre PASCAL et I-PASCAL peut être établi par les deux options suivantes :

(1) les restrictions imposées sur I-PASCAL :

a) modification de <index type> et l'omission de plusieurs descripteurs pour un type implicite donné, sont négligées ;

b) les opérateurs et instructions supplémentaires sont supprimés en constituant un sous-ensemble de I-PASCAL ;

(2) extension de PASCAL (super-PASCAL) : les opérateurs et instructions supplémentaires sont inclus dans le langage source PASCAL.

La figure 13 illustre les trois cas possibles des relations entre PASCAL et I-PASC

Le troisième cas est constitué par le fait que les opérateurs de I-PASCAL sont applicables aux <record type> qui ne sont pas permis dans le langage PASCAL. Par exemple, T1 := T2 + T3 ou T1 := T2 * T3, où T1, T2 et T3 sont des tableaux, peuvent être définis par les microprogrammes correspondants aux opérateurs matriciels + et * , dans la machine PASC-HLL [2]. L'extension de I-PASCAL peut donc s'étendre aux langages de très haut niveau (LTHN) comme ADA, etc .., grâce à l'interprétation des descripteurs pendant l'exécution du LM.

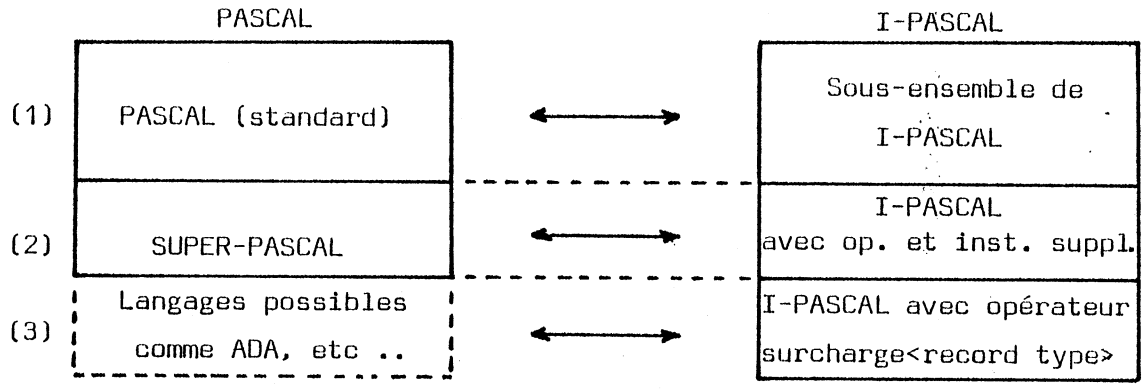


Figure 13 - Correspondance avec les sous-ensembles de I-PASCAL

5.4. CONCLUSION

I-PASCAL est isomorphe à PASCAL (selon sous-ensembles donnés). En effet, I-PASCAL est le langage symbolique représentant les codes du LM de PASC-HLL. En notant l'isomorphisme par $\cong$, on voit les relations suivantes :

PASCAL $\cong$ I-PASCAL $\cong$ LM de PASC-HLL

La machine PASC-HLL exécute directement le langage qui est isomorphe à PASCAL (par transitivité, PASCAL $\cong$ LM de PASC-HLL). La machine PASC-HLL est donc une machine PASCAL et elle est classée dans la catégorie des machines LHN.

CHAPITRE 6

TRADUCTEUR ISOMORPHE DE PASCAL

6.1. INTRODUCTION

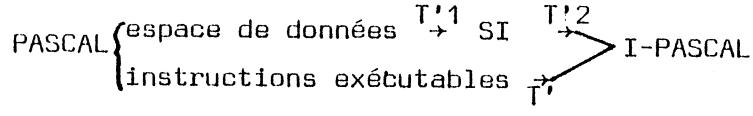
Ce chapitre présente le traducteur de PASCAL qui génère le code de I-PASCAL. Ce traducteur est isomorphe de PASCAL d'après le chapitre précédent. Les principes et les algorithmes présentés au chapitre 4 sont donc appliqués aux langages PASCAL et I-PASCAL.

Le traducteur isomorphe de PASCAL peut se décomposer en deux couches physiquement indépendantes : l'analyse syntaxique et les actions sémantiques. La couche de l'analyse syntaxique est construite sur la syntaxe modifiée de PASCAL par implantation des appels des actions sémantiques dans l'ordre syntaxique. Le décodage des identificateurs et les tests d'erreurs syntaxiques sont effectués dans cette couche. La couche syntaxique est donc une partie indépendante de la configuration de la machine réelle.

La couche des actions sémantiques est physiquement séparée de la précédente, et la communication entre les deux couches est réalisée par les paramètres des procédures sémantiques et par les variables globales. La fonction principale de cette couche est de générer la structure de I-PASCAL. Elle ne contient donc ni la phase d'optimisation des codes, ni la fragmentation des données et instructions.

La couche des actions sémantiques contient la partie qui dépend de la machine, puisqu'elle englobe la représentation des valeurs par les mots de la machine. Il y a donc trois procédures possibles pour la traduction, dans la mesure où la partie dépendante de la machine est isolée dans la couche des actions sémantiques.

1) traduction directe : PASCAL $\xrightarrow{T}$ I-PASCAL ;
2) traduction indirecte (par les structures intermédiaires (SI)) :
   PASCAL $\xrightarrow{T1}$ SI $\xrightarrow{T2}$ I-PASCAL
3) traduction mixte :



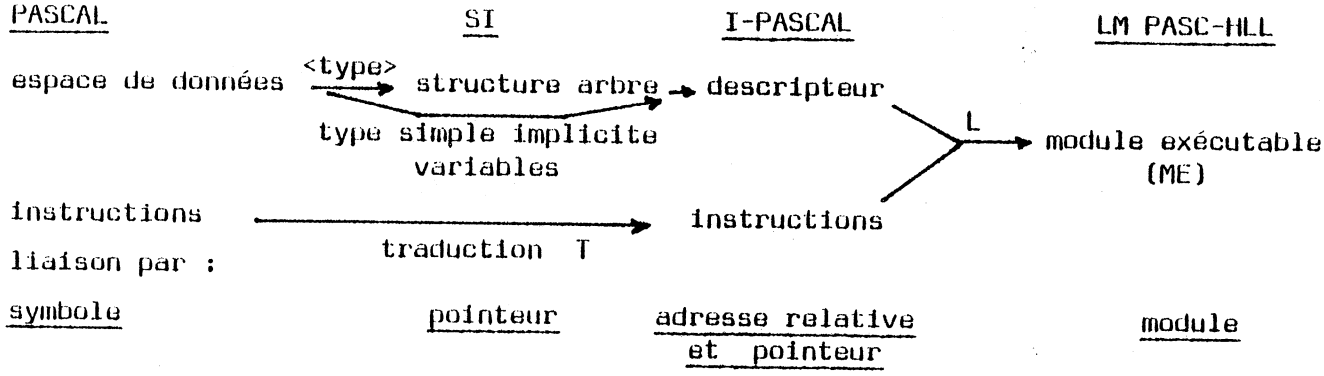
Dans la traduction directe, la couche des actions sémantiques constitue la partie dépendante de la machine. Dans le cas de la traduction indirecte, on a deux parties : l'une est indépendante et l'autre dépendante de la machine. Les structures intermédiaires (SI) sont construites d'une manière indépendante de la machine en satisfaisant les relations PASCAL $\cong$ SI $\cong$ I-PASCAL (SI et I-PASCAL sont presque isomorphes). La traduction mixte, enfin, adopte la structure intermédiaire avant de créer le descripteur, cependant, il traduit les instructions de PASCAL directement en instructions de I-PASCAL.

Les structures intermédiaires (SI) sont en général représentées par des structures de listes (liaison par pointeur). Si la traduction T2 de la traduction indirecte est réalisée par l'architecture matérielle, par exemple la machine LISP [39], ce processus sera acceptable pour représenter LHN sous la forme d'un arbre de programme (qui est une des façons de représenter le LHN isomorphe (cf. § 5.1.)).

On adoptera la traduction mixte pour la construction du traducteur de PASCAL.

6.2. LES STRUCTURES INTERMÉDIAIRES ET TEMPORAIRES

Le traducteur de PASCAL (traduction mixte) contient aussi la phase de liaison des modules comme suit :



L'espace de données de PASCAL représenté par la syntaxe <type> est d'abord transformé en une structure d'arbre. Ensuite, la structure d'arbre est traduite en descripteurs. Les types implicites (entier, réel, booléen, ...) et les variables déclarées, sont directement transformés en descripteurs. Basées sur ces descripteurs, les instructions de PASCAL sont directement traduites en instructions de I-PASCAL dont les références des variables sont effectuées par l'adresse relative et par les pointeurs.

Les structures de I-PASCAL sont finalement groupées dans le module exécutable (ME) dont l'unité de bloc est une procédure [2]. Ces blocs de procédures sont liés par les pointeurs qui sont groupés dans la table des segments. Le module exécutable est la structure que la machine PASC-HLL exécute. la figure 14 représente la structure du module exécutable [2].

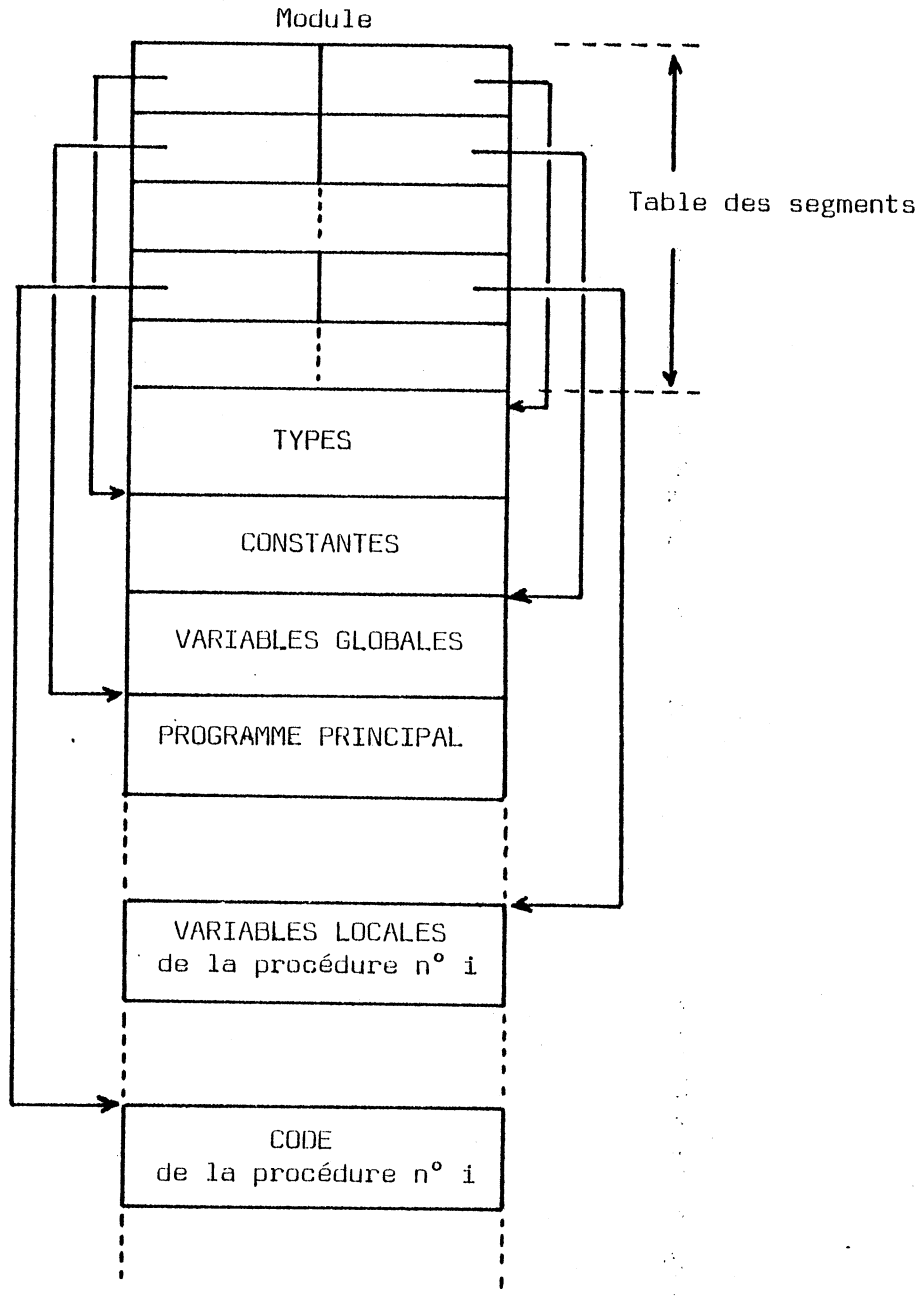


Figure 14 - Module exécutable de PASC-HLL

La structure de I-PASCAL (éventuellement les descripteurs et les instructions) contient donc les informations supplémentaires qui seront utilisées pour la liaison du module exécutable. Les trois types de structures sont :

1) la table des segments (TS),
2) les descripteurs (DES),
3) le code I-PASCAL (ICODE).

TS contient les informations sur les blocs de procédure : la position du code (CODEP), le descripteur (DVARI), les paramètres (DPARA), et leurs volumes (CODLG, NV, NTZVV, NP, TZVP), etc ..
DES est la structure du descripteur dans laquelle les descripteurs de 32 bits (DESVAL) ont une adresse relative (DESNO), un nom symbolique (DESNAME).
Il existe six sortes d'instructions dans I-PASCAL (ICODE) dont les longueurs sont 8, 16, ou 24 bits.
Le schéma de la figure 15 décrit les structures qui contiennent I-PASCAL (cf. également l'annexe B.3.).

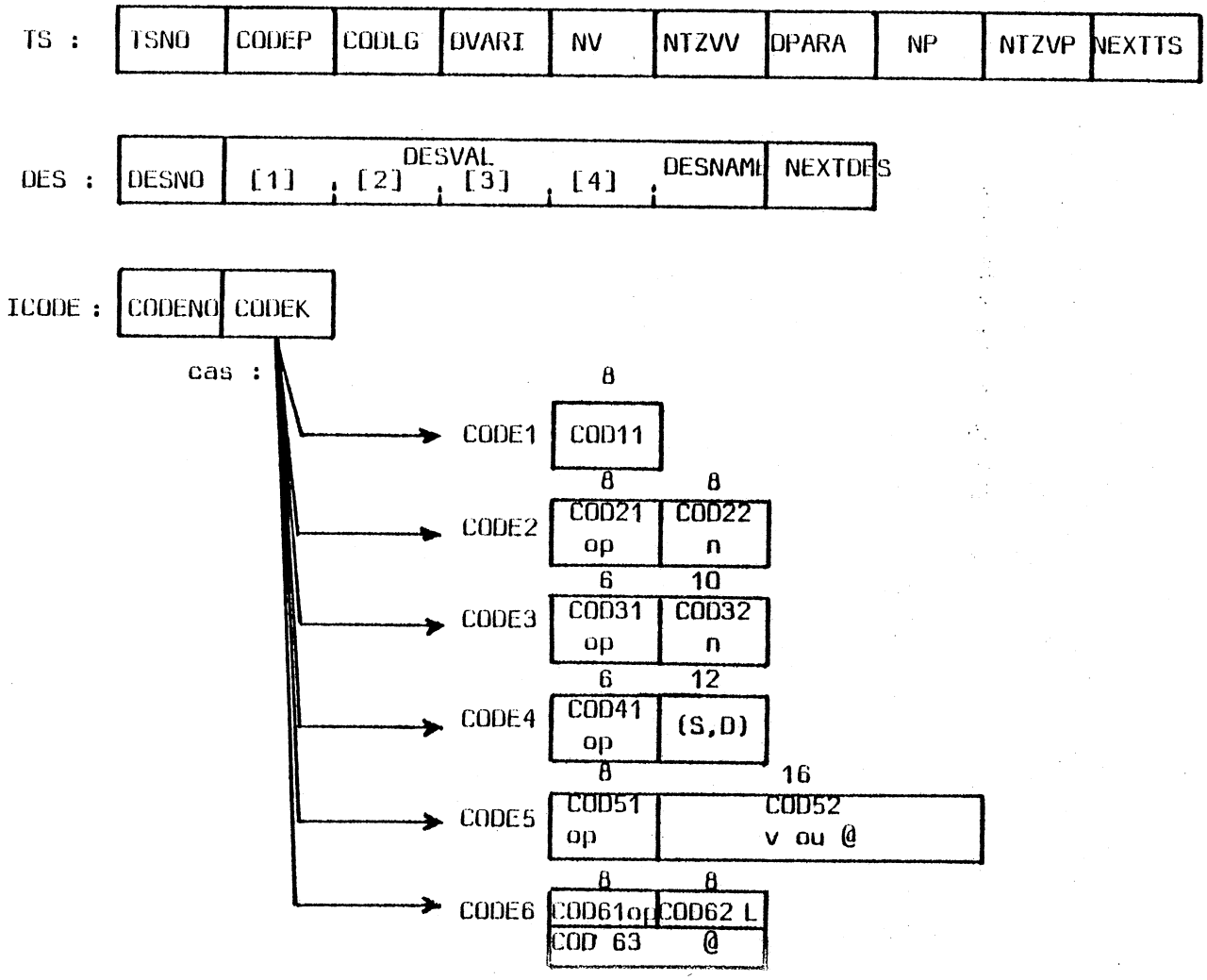


Figure 15 - Structure de I-PASCAL

Basé sur ces trois types (TS, DES, ICODE), PASCAL est traduit en I-PASCAL dont les structures sont déclarées par les variables FTYPE pour les types, FCON pour les constantes et FTS pour la table des segments qui contient les pointeurs vers les segments. Ces structures constituent donc le programme objet du traducteur PASCAL. La figure 16 illustre ces structures.

Le programme objet de T : PASCAL → I-PASCAL peut être temporairement résident soit dans la mémoire principale, soit dans la mémoire secondaire, si le volume du programme est trop grand. L'éditeur de liens regroupe donc les structures de I-PASCAL dans le module exécutable.

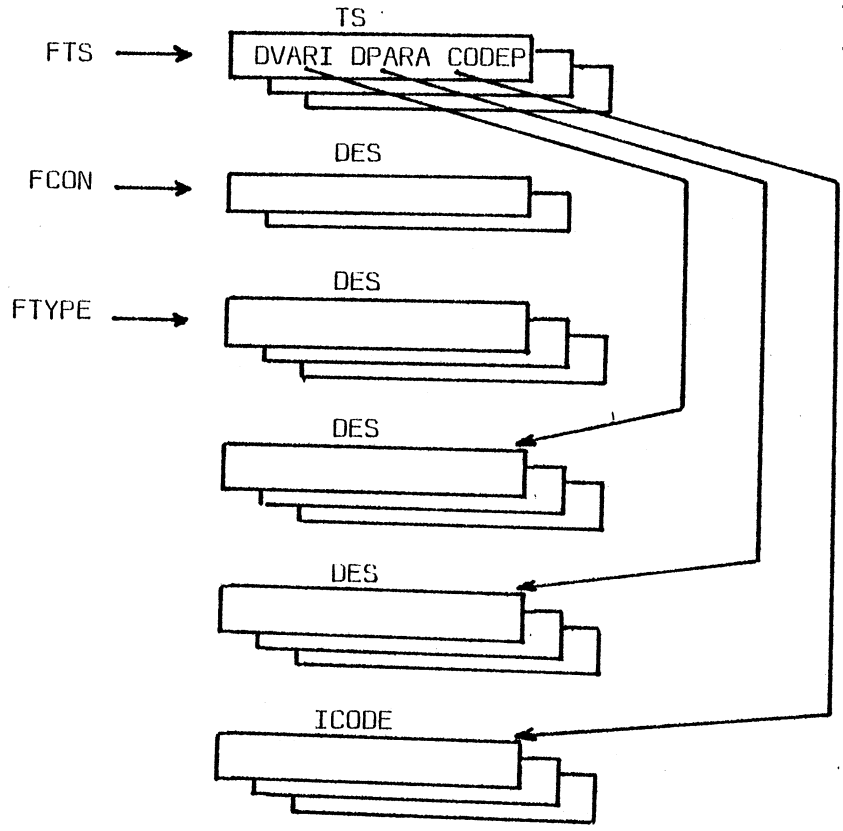


Figure 16 - Programme objet (I-PASCAL)

Pendant la traduction isomorphe, des structures temporaires sont utilisées pour le décodage des identificateurs et pour les transformer en descripteurs. La structure IDTAB constitue l'arbre des identificateurs qui contient les informations sur les variables déclarées. Le décodage des identificateurs est basé sur l'imbrication des blocs. CONSTA est la structure qui contient la valeur de la constante définie par le programmeur. ITYPE est la structure intermédiaire qui représente la syntaxe <type>. Cette structure est basée sur la récursivité en formant l'arbre. Ces structures (IDTAB, CONSTA, ITYPE) sont décrites en annexe B.3.

6.3. CONSTRUCTION DU TRADUCTEUR

6.3.1. Modification de la grammaire PASCAL

6.3.1.1. LL(1) PASCAL

La syntaxe de PASCAL est "contexte libre", ce qui s'écrit généralement sous la forme de Backus-Naur (BNF) [36]. L'ordre des définitions de la syntaxe n'est pas important, ce qui signifie qu'elle est du type non-procédural.

Afin de construire le traducteur isomorphe de PASCAL basé sur les principes définis au chapitre 4, les trois étapes de la modification syntaxique sous la forme LL(1) doivent être définies.

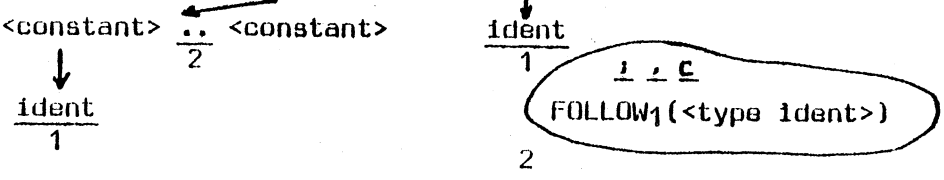
D'après la définition de LL(k), on peut démontrer (théorème classique) que la grammaire ambigüe n'est pas LL(k) et que la grammaire qui contient la récursivité gauche n'est pas LL(k) [31]. (Les définitions de LL(k) et de FIRSTk se trouvent au § 4.2.3.1. et dans [31]).

La grammaire de PASCAL contient une partie de la syntaxe ambigüe, une partie de la récursivité gauche et trois parties de LL(2). Le reste est LL(1). La cause de LL(2) et de l'ambiguité est l'identificateur non-décodé.

(1) Les parties de LL(2) sont les suivantes, où FOLLOWk(α) est défini comme dans [31] :

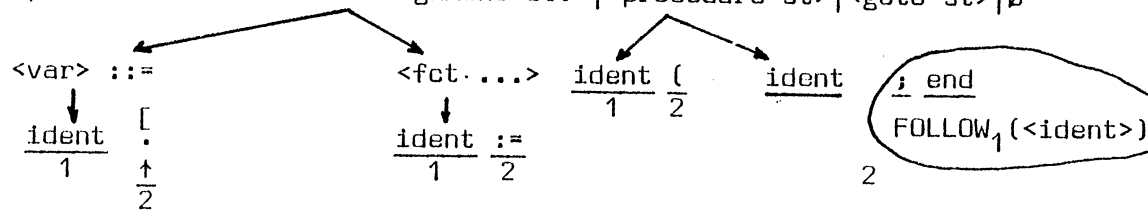
$$FOLLOWk(\beta) = \{w \mid S \overset{*}{\Rightarrow} \alpha\beta\gamma \text{ et } w \text{ est dans } FIRSTk(\gamma)\}$$

(a) <simple type> ::= <scalar type> | <subrang.type> | <identifier type>

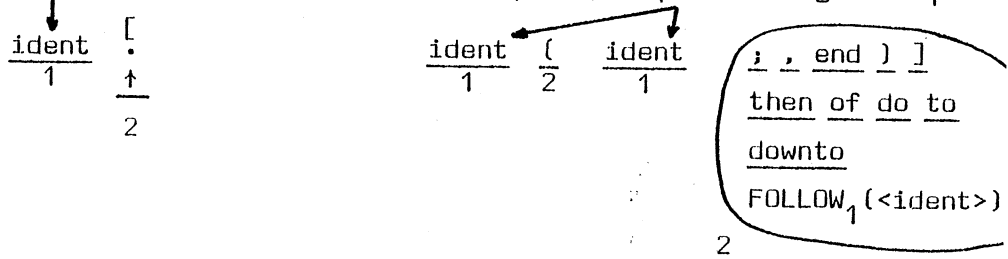


où il faut avoir les deux unités lexicales pour chaque variable de choix : ident .. et ident $FOLLOW_1$(<type ident>), pour que la dérivation distincte soit obtenue entre <subrange type> et <identifier type> . (L'explication est la même pour les exemples suivants).

(b) <simple statement> ::= <assignment st.>|<procedure st>|<goto st>|Ø

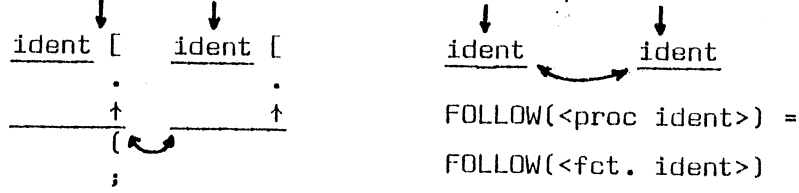


(c) <factor> ::= <variable>|< unsigned const.>|(<expr>)|<fct designator>|



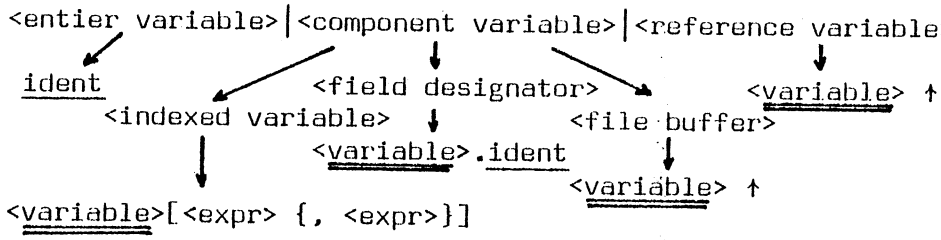
(2) Dans la grammaire ambigüe, il y a plus de deux dérivations distinctes pour une chaîne w donnée. Dans PASCAL <actual parameter> est une syntaxe ambigüe

<actual parameter> ::= <expr>|<variable>|<proc. ident.>|<fct. ident>



(3) La partie de la syntaxe <variable> constitue la dérivation récursive à gauche.

<variable> ::= <entier variable>|<component variable>|<reference variable>



Il y a deux options possibles pour que la grammaire PASCAL soit modifiée en LL(1)

1) la préparation des mots réservés préfixés : devant chaque identificateur qui génère une syntaxe non-LL(1), on préfixe le mot réservé distinct : par exemple, varident, fctident, etc .. La syntaxe devient donc :
<...> ::= varident ident ...| fctident ident ...| typeident ident ... etc ..
Il faut donc modifier la syntaxe originale de PASCAL en ajoutant des mots réservés non-standard. De plus, le décodage des identificateurs par les mots réservé est sous la responsabilité du programmeur.

2) La procédure spéciale : une procédure spéciale pour classer les identificateurs est introduite dans le traducteur. Devant les identificateurs provoquant une syntaxe non-LL(1), cette procédure s'applique.

Nous avons choisi la deuxième option. La procédure spéciale, appelée IDKINDS, indique la classe de l'identificateur à analyser. La classe lexicale des identificateurs est : IDCLASS = (TYPES, KONST, VARS, FIELD, PROC, FUNC). Le format syntaxique de IDKINDS est le suivant :

[[ident]] IDKINDS ((ident (TYPES) ... | ... | ... | ident (VARS) ... etc))

Les méta-parenthèses [[ ]] signifient l'élément préfixé tel que défini aux § 4.2.2. et 4.2.3.3. Si l'analyseur prend une unité lexicale qui est l'identificateur, la procédure IDKINDS est donc choisie sans avancer le pointeur. Cette unité est ensuite décodée par procédure suivant la classe lexicale, et le choix suivant peut donc être déterminé par la classe de l'identificateur (dans ce cas, le pointeur avance à l'unité suivante).

6.3.1.2. Les trois étapes de la modification

En choisissant IDKINDS, la modification de la grammaire en LL(1) se fait par étapes. Pendant la première et la deuxième étape de la modification, les variables syntaxiques identifier et constant sont traitées comme des unités lexicales. Elles sont finalement décodées pendant la troisième étape de la construction de la grammaire LL(1). Techniquement, la modification de la syntaxe en LL(1) se fait donc de la manière suivante (les annexes A.3 et A.4 indiquent le déroulement de la première et de la deuxième étape conformément au § 4.2.3. ; l'annexe A.5 illustre la modification finale en LL(1) qui sera utilisée pour la construction du traducteur).

(1) LL(2) → LL(1) :

(a) <simple type> ::= (ident, {ident}) |
[[ident]] IDKINDS ((ident (TYPES) | <const> .. <const>))

(b) <unlabelled st.> ::= [[ident]] IDKINDS ((([[ident(VARS)]] <assgt.st.> |
[[ident(FUNC)]] <assgt.fct.st> | [[ident(PROC)]] <proc.st>))
| goto <goto st.> | begin <begin st.> | ... etc

<simple statement> est modifié en <unlabelled st.> pendant la première étape.

(c) <factor> ::= [[ident]] IDKINDS ((([[ident(VARS)]] <variable>|[[ident(FUNC)]]
<fct designator>)) | <unsigned const.>|... etc ..

(2) G ambigüe → LL(1) :

<actual parameter> ::= [[ident]] IDKINDS ((ident(PROC) | ident(FUNC))) | <expr>

<var> est modifié en <expr>.

(3) Récursivité gauche → récursivité droite avec LL(1) :

<variable> ::= ident ℓ9

ℓ9 ::= [<expr>{,<expr>}] ℓ9 | . ident ℓ9 | ↑ ℓ9 | ∅

Par exemple, l'interprétation de <simple type> est la suivante : si l'unité lexicale à analyser est "(", alors le premier choix est pris, sinon si l'unité n'est pas l'identificateur, on a une erreur, sinon la procédure IDKINDS est activée. Etant donné que l'identificateur est décodé par IDKINDS, si l'identificateur est une classe de type, le premier choix dans la parenthèse de IDKINDS est choisi, sinon le deuxième choix <const> .. <const> est inconditionnellement choisi.

De même, <unlabelled st.> peut être expliqué par l'algorithme suivant :

```
procedure <unlabelled st> ;
  begin if SY not in (ident, goto, begin, ... etc ...) then erreur
  |   else case SY of
  |   |   ident : begin call IDKINDS ;
  |   |           |     if IDCLASS not in (VARS, FUNC, PROC) then erreur
  |   |           |     else case IDCLASS of
  |   |           |          |     VARS : <assgt. st.> ;
  |   |           |          |     FUNC : <assgt. fct. st.> ;
  |   |           |          |     PROC : <proc. st.> ;
  |   |           |          end ;
  |   |           end ;
  |   |   goto : begin NEXTTOKEN ; <goto st.> end ;
  |   |   begin : begin NEXTTOKEN ; <begin st.> end ;
  |   |   etc ..
  |   end
  end ;
```

Les unités lexicales soulignées sont examinées par l'analyseur avec le pointeur SY pour décider du choix de dérivation. La procédure NEXTTOKEN signifie l'avancement du pointeur vers l'unité suivante. Ceci est appliqué lorsque l'unité soulignée n'est pas l'élément préfixé entre [[et]].

La grammaire LL(1) peut être analysée par l'algorithme de "parser" prédictif [31]. La figure 17 représente l'algorithme "parser" K-prédictif qui analyse la grammaire LL(k).

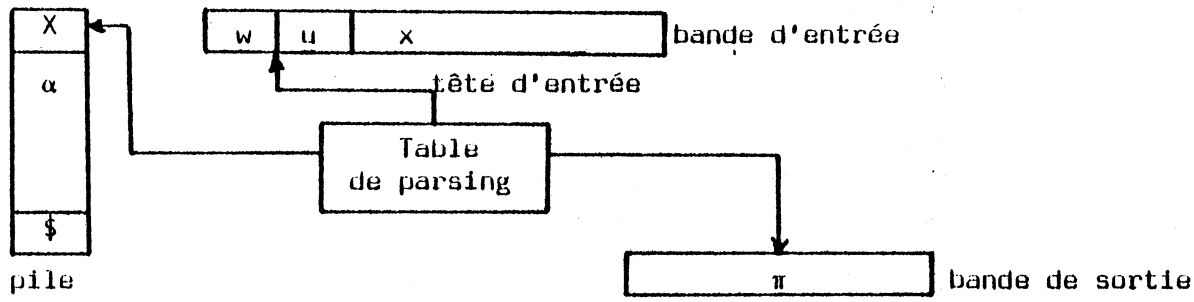


Figure 17 - Algorithme de parser prédictif

La figure 18 représente une partie de la table de parsing correspondant à LL(1) PASCAL.

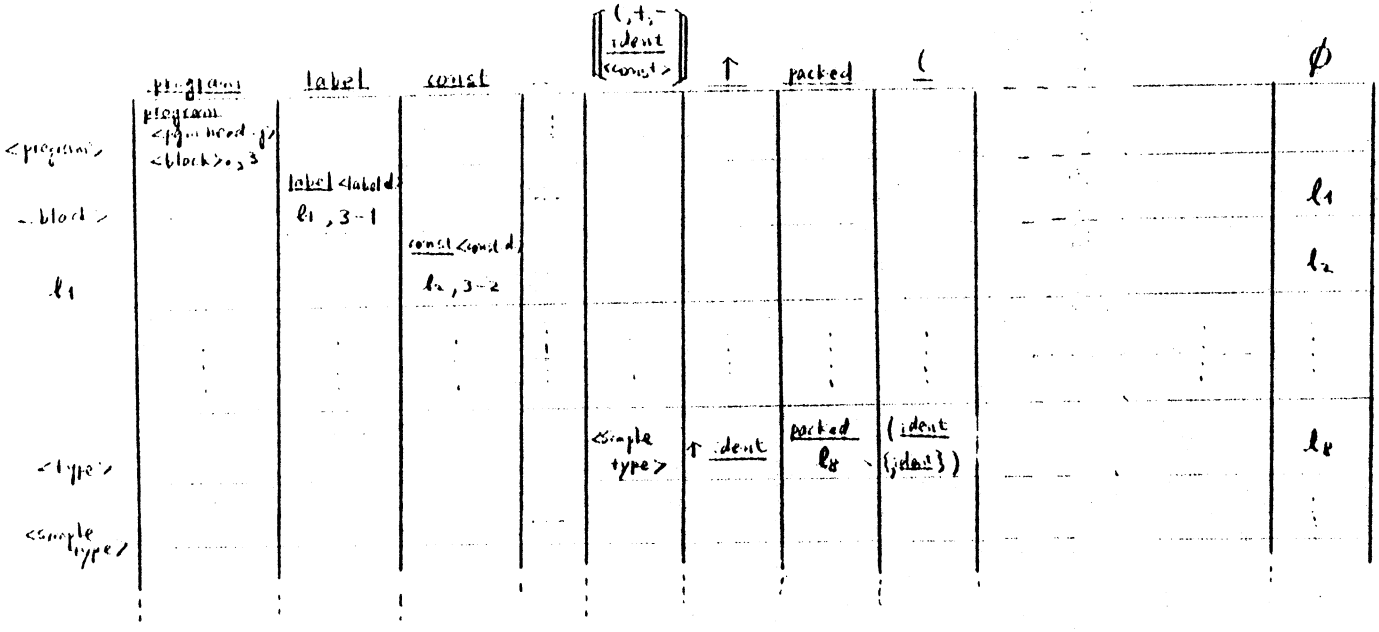


Figure 18 - Une partie de la table de parsing : LL(1) PASCAL

Dans le traducteur orienté syntaxe, cet algorithme de "parser" prédictif est donc modifié de la manière suivante :

a) la table de parsing correspondant à LL(1) PASCAL modifiée, est représentée soit par les unités lexicales précédant les variables de choix, soit par l'élément préfixé dans la méta-parenthèse [[ ]] ;
b) la pile de l'algorithme est représentée par la structure de blocs récursifs dans laquelle les variables locales fonctionnent comme les données de la pile.

Le traducteur orienté syntaxe modifié de LL(1) PASCAL est donc équivalent à l'algorithme de "parser" prédictif.

6.3.2. Grammaire de type procédural

Avant toute programmation basée sur la syntaxe modifiée, il faut que la syntaxe soit de type procédural si le langage du traducteur est de type procédural. Il est aussi possible, dans le cas où le langage du traducteur est un langage de très haut niveau (LTHN) dont le type est non-procédural, que cette étape puisse être éliminée. Il s'agit ici du choix du langage outil de traduction pour décider de la nécessité de cette étape.

Dans cette présentation, le langage PASCAL lui-même est utilisé pour la construction du traducteur isomorphe de PASCAL. Pour ce langage outil de développement, la version du PASCAL Microentine a été choisie [10]. Ceci constitue le compilateur croisé (de même que le Microengine constitue la machine croisée) par rapport à la machine PASCAL PASC-HLL. La syntaxe modifiée doit donc être basée sur l'imbrication des blocs de PASCAL.

Les parties de la syntaxe à considérer sont les suivantes :

a) la syntaxe de <type> est utilisée pour la définition de <type def. part> et de <variable declaration part>. <type> doit donc se situer au même niveau lexical que celui de <type def. part> et de <variable declaration part> . A l'intérieur de <type>, <simple type> et <field list> sont rangés au même niveau lexical. De même, <formal parameter sec.> est utilisé par <proc. declaration> et par <fct. declaration>. Ils se situent donc au même niveau lexical.

b) Les expressions syntaxiques <expr>, <variable> et <actual parameter> sont définies l'une par rapport à l'autre, et on utilise aussi FORWARD au même niveau lexical.

La relation d'imbrication peut donc être exprimée par la figure 19 dans laquelle les lignes verticales de (1) à (6) signifient les bornes des blocs correspondant aux variables syntaxiques et où une même position verticale correspond au même niveau lexical. Les variables syntaxiques sont ici les parties gauches de ::= (qui deviennent des noms de procédures), en omettant les parties droites pour la démonstration lisible de la relation d'imbrication. Les variables intermédiaires, notées ℓ1, ℓ2, ..., ℓi, de la deuxième étape, sont éliminées dans le schéma 19, en supposant qu'elles sont introduites dans le bloc. Ce schéma correspond donc au rangement de la grammaire modifiée de PASCAL (cf. annexe A.5) qui sera acceptée par le compilateur croisé (PASCAL Microengine).

6.3.3. Programmation en deux couches séparables

La syntaxe modifiée et rangée est directement programmable. En résumé, étant donnée une définition de syntaxe, elle est programmée sur la base des règles suivantes :

a) la variable V_N de la partie gauche de ::= devient le nom de procédure ;

b) la partie droite de ::= devient le corps de procédure dans lequel :
   - les variables syntaxiques V_N sont remplacées par les appels de procédures syntaxiques correspondants,
   - les contrôles de programme sont appliqués conformément au § 4.2.3.,
   - l'action sémantique est réalisée soit par l'appel de procédure sémantique, soit par l'implantation directe de module sémantique dans le module syntaxique ;

c) les variables locales sont générées si nécessaire.

D'après les caractéristiques du traducteur isomorphe, la couche syntaxique et les actions sémantiques sont séparables (§ 4.1.). Dans le module syntaxique du traducteur isomorphe de PASCAL, les actions sémantiques sont donc réalisées par des appels de procédures sémantiques, dont les modules sont regroupés dans un module physiquement séparé de la couche syntaxique.

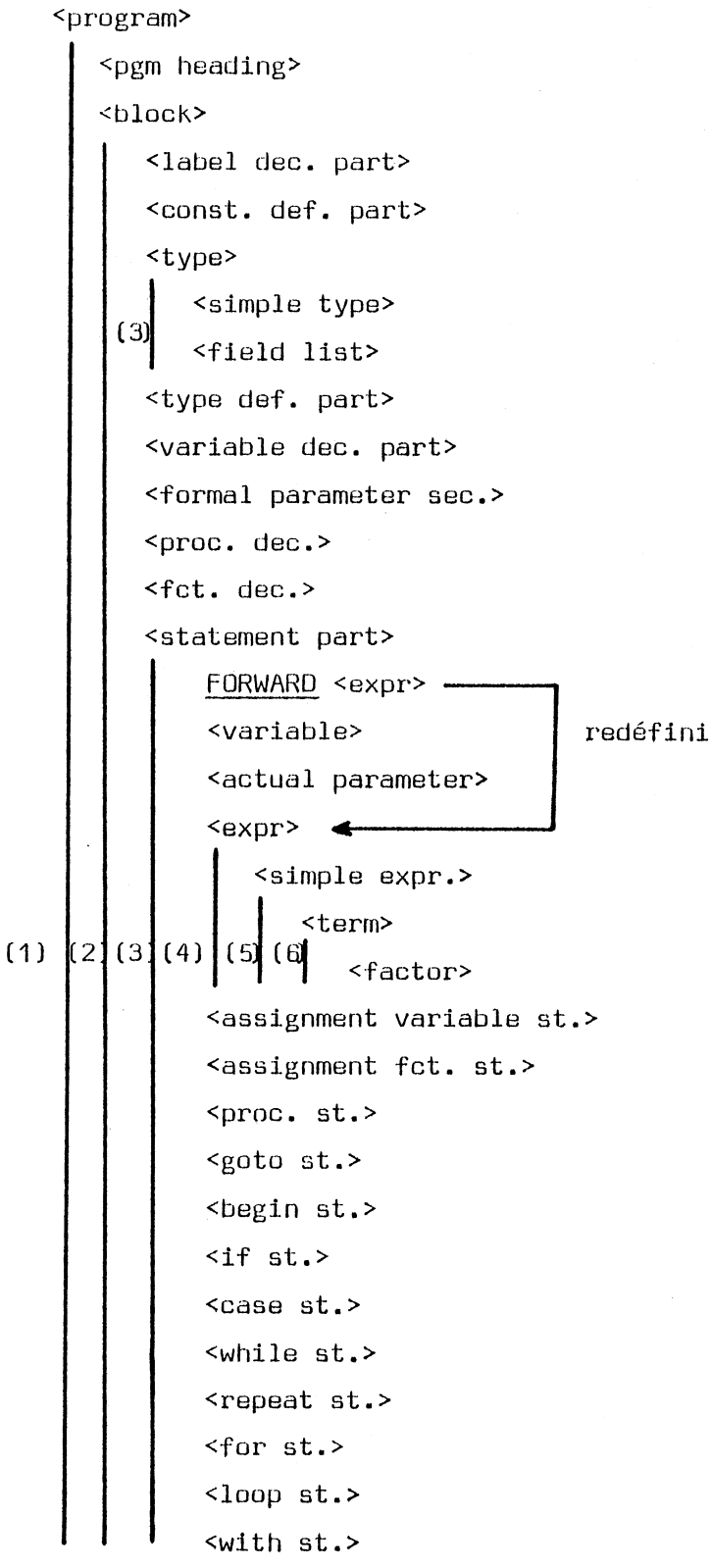


Figure 19 - Le type procédural de la syntaxe modifiée de PASCAL

Par la séparation physique entre les deux couches syntaxique et sémantique, la portée des noms de la couche syntaxique ne peut plus être appliquée à la couche sémantique. La communication entre les deux couches est donc assurée par les variables globales ou par les paramètres des procédures sémantiques. Les variables locales de la couche syntaxique doivent être passées comme paramètres des procédures sémantiques lorsqu'elles sont référencées par les actions sémantiques.

Il est inévitable, pour sauvegarder l'information de structure, d'établir des variables locales de la couche syntaxique, lorsque la syntaxe est récursive et que la traduction des objets est commutative. Les variables locales du traducteur isomorphe de PASCAL sont limitées à :

- pointeurs de données structurées (descripteurs, programmes de contrôle),
- opérateurs (notation post-fixée dans <expr>)
- étiquettes de <goto>.

Pour la construction de la couche sémantique, les blocs suivants sont nécessaires :

- établissement des structures globales (variables globales),
- actions sémantiques correspondant aux procédures sémantiques,
- initialisation et blocs de primitives.

6.3.4. Environnement de programmation

6.3.4.1. Variables globales

Les variables globales sont des structures qui sont utilisées par le module sémantique et par les primitives. Elles sont mises dans l'état initial par le module d'initialisation.

Trois catégories de structures constituent les variables globales :

a) les structures intermédiaires,

b) les structures temporaires,

c) les structures pour l'analyse lexicale.

On retrouvera ces structures en annexe B.3.

Le module exécutable (ME) est représenté dans I-PASCAL par les structures intermédiaires. Les descripteurs de I-PASCAL sont construits par les structures temporaires. Les tables sont préparées pour l'analyse lexicale dont la fonction consiste à produire les unités lexicales qui sont acceptables pour l'analyseur syntaxique.

6.3.4.2. Variables locales

La figure 20 présente les variables locales qui sont implantées dans la syntaxe modifiée et rangées dans PASCAL.

La variable TSTEMP est un pointeur qui sauvegarde la position de la table des segments dans le module exécutable (ME). Elle est nécessaire en raison de la récursivité de <block>. LABTAB est le tableau local utilisé pour la manipulation des étiquettes locales de <label dec. part>. Les variables TYPE1, T1 et T2 sont des pointeurs nécessaires en raison de la récursivité de <type> et de la sauvegarde des pointeurs en cours de construction des descripteurs. Les variables JMP1, JMP2 sont les pointeurs qui sauvegardent les positions des instructions du programme de contrôle. La récursivité de <compound statement> fait apparaître ces variables locales. Les variables EXOP, SIMPOP et TERMOP sont les opérateurs utilisés pendant la transformation de la syntaxe <expression> de PASCAL en syntaxe de I-PASCAL Autrement dit, c'est la traduction d'une expression in-fixée dans une expression post-fixée, d'après la syntaxe récursive de <expression>. Le nombre maximum de paramètres pour la procédure sémantique ne dépasse donc pas trois paramètres dans le cas de la séparation physique en deux couches.

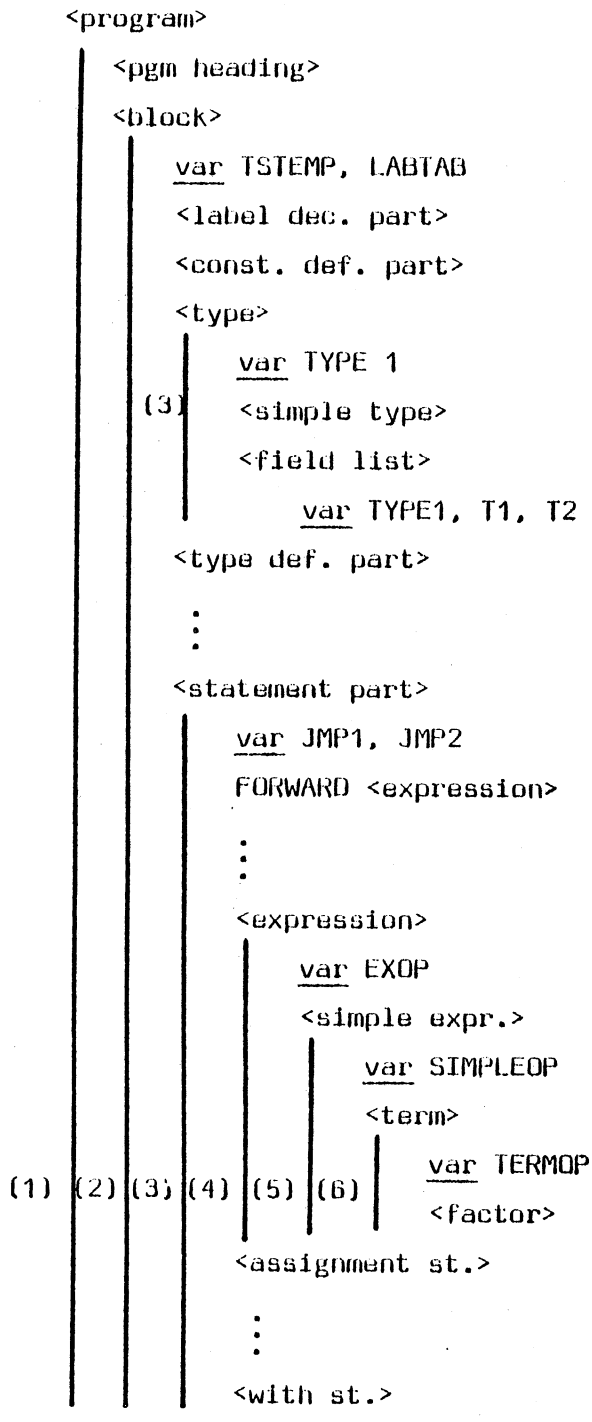


Figure 20 - Variables locales de la couche syntaxique

6.3.4.3. Primitives

Il y a trois catégories de procédures dans le module des primitives :

a) la création de structures intermédiaires et temporaires, et l'insertion de la structure dans la liste linéaire,

b) l'analyse lexicale et les tests d'erreurs syntaxiques,

c) les primitives pour la traduction de la syntaxe.

La première catégorie consiste à préparer les procédures qui traiteront de la création des structures de données (TS, DES, ICODE, ITYPE, IDTAB, CONSTA, cf. annexe B.3). Cette création est effectuée par la procédure intrinsèque NEW, avec les paramètres de pointeur de type. Avant d'utiliser les structures créées, elles sont initialisées (par les primitives NEWIDTYPE, NEWRECSTR, NEWIDTAB, NEWTS). La liste linéaire de ces structures est construite par les primitives NEXTDES, NEWTCONS, NEXTTAB, ...

La deuxième catégorie consiste à analyser les alphanumériques afin de préparer les unités lexicales et les messages d'erreurs (IDKINDS, ERROR, NEXTTOKEN, SEARCHID, ...).

La troisième catégorie concerne les primitives qui sont utilisées pour la traduction de la syntaxe modifiée.

Soit <A> ::= a α une règle de syntaxe où A est une variable de V_N, a est un terminal de V_T et α est une chaîne. Les trois cas suivants sont à considérer pour la programmation :

```
(1) if SY = "a" then NEXTTOKEN else ERROR (NO) ;
(2) if SY = "a" then begin SEMANT (  ) ; NEXTTOKEN end
                else ERROR (NO) ;
(3) if SY = "a" then begin SEMANT (  ) ; NEXTTOKEN end
                else begin ERROR (NU) ; SKIP (  ) end ;
```

Le premier cas n'a pas d'action sémantique.
Le deuxième cas fait l'action sémantique en appelant une procédure sémantique.
Le troisième cas récupère l'unité à analyser.
Ces primitives peuvent être exprimées par une seule procédure, si l'appel par nom est permis pour le passage des paramètres. A défaut de cette facilité dans le langage PASCAL, ces primitives sont distinctes (de plus, PASCAL de Microengine ne permet pas de noms de procédures et de fonctions comme les paramètres actuels de procédures).

Soit <A>{,< A >}; une règle de< syntaxe>. Si le pointeur d'analyseur est sur le début de <A> (c'est-à-dire FIRST1(<A>)), cette syntaxe est programmée par l'instruction non standard loop exitif end :

```
loop <A> ; NEXTTOKEN exitif SY = ';' ;                                    (1)
                    if SY = ',' then NEXTTOKEN else ERROR end
```

Si le pointeur d'analyseur indique l'unité lexicale précédente de <A>, cette syntaxe peut être représentée par :

```
repeat NEXTTOKEN ; <A> ; NEXTTOKEN until SY = ';'
```

Pourtant, il est rare que le pointeur d'analyseur soit sur unité lexicale située avant <A> lorsque l'analyse est commencée avec cette syntaxe.
Si le pointeur d'analyseur est sur le début de <A>, et si l'on n'adopte pas l'instruction non standard (la machine Microengine ne permet pas l'instruction loop exitif end), le programme (1) devient :

```
var : CONT : boolean ;                                                    }
      repeat <A> ; NEXTTOKEN ;                                            }
            if SY = ';' then CONT := VRAI                                 } (4)   } (5)
                        else début CONT := faux ; NEXTTOKEN end           }       }
      until CONT ;                                                        }
```

La partie (4) est réalisée par une primitive (CONT1). Si l'appel par nom est permis, la partie totale de (5) peut être exprimée par une seule primitive. Pourtant, si la boucle est imbriquée n fois, n variables booléennes (comme CONT) doivent être utilisées dans cette technique.

6.4. ORGANISATION DES PROGRAMMES

Dans le langage PASCAL, les déclarations de procédures doivent précéder le programme qui appelle ces procédures. L'organisation des modules du traducteur est donc conforme au schéma de la figure 21.

La couche sémantique est physiquement séparée de la couche syntaxique. Cette dernière est un module de blocs imbriqués selon la syntaxe modifiée de PASCAL. Cependant, les actions sémantiques sont constituées par les procédures sémantiques simples qui ne contiennent plus d'autres procédures locales (pas de blocs imbriqués). Les variables locales du module syntaxique sont communiquées avec les actions sémantiques par les paramètres des procédures sémantiques. On remarque qu'il n'est pas nécessaire d'utiliser d'instructions de branchement (go to).

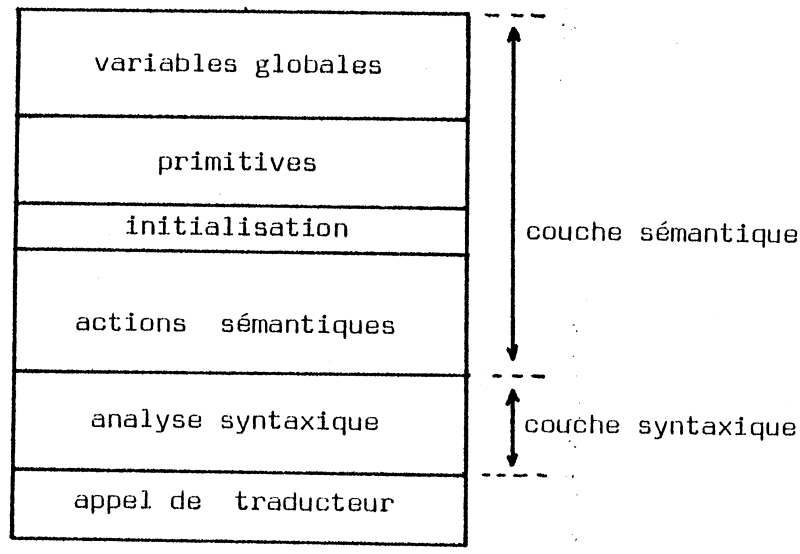


Figure 21 - Organisation du traducteur

6.5. LANGAGE D'ÉCRITURE DU TRADUCTEUR

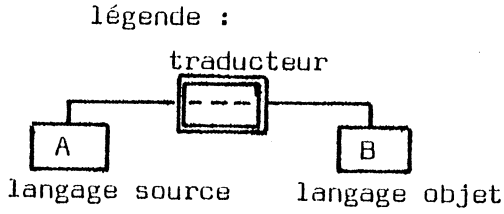
6.5.1. Auto-compilation

Afin de construire le traducteur isomorphe de PASCAL dont le langage objet est exécutable par la machine PASC-HLL, il faut fournir les outils de développement : le compilateur et la machine croisée qui fournit ce compilateur. Le programme du traducteur (le langage source LS) de PASCAL est compilé par le compilateur de la machine croisée qui crée le langage machine (LM) exécutable par elle-même. Après exécution de LM sur la machine croisée, les programmes écrits en PASCAL sont traduits en programme I-PASCAL exécutable par la machine PASC-HLL (appelons le LM un "traducteur croisé").

Si le traducteur de PASCAL est construit dans le même langage que PASCAL, le traducteur résident (qui se trouve dans la machine PASC-HLL) peut être obtenu par auto-compilation de LS. La machine Microengine est dans ce cas la machine croisée dont le compilateur accepte le langage PASCAL. La figure 22 illustre les étapes de cette auto-compilation.

Le traducteur de PASCAL est d'abord construit par le langage PASCAL (étape (1)). Ce traducteur est compilé par le compilateur de la machine croisée (compilateur PASCAL de Microengine : étape (2)). Le traducteur compilé est écrit en LM de la machine croisée (P-code). Ceci constitue le traducteur croisé. Ensuite, le traducteur écrit en PASCAL (LS) est traduit par lui-même (LM)(auto-compilation) par les codes LM de la machine croisée (étape (3)). Le résultat de l'étape (3) est le traducteur de PASCAL en codes I-PASCAL. Il est donc exécutable par la machine PASC-HLL et il peut s'installer dans la machine PASC-HLL comme traducteur résident.

Une fois que le traducteur résident est obtenu par les étapes (1) à (3), les programmes PASCAL sont traduits en I-PASCAL et exécutés par PASC-HLL (étape (4)). Les étapes (1) à (3) constituent donc la partie amorçage qui est supprimée après avoir obtenu le traducteur résident. On peut aussi adopter le traducteur croisé à chaque traduction des programmes PASCAL (étape (3)). Dans ce cas, la machine croisée doit toujours être installée autour de PASC-HLL pour la traduction croisée.



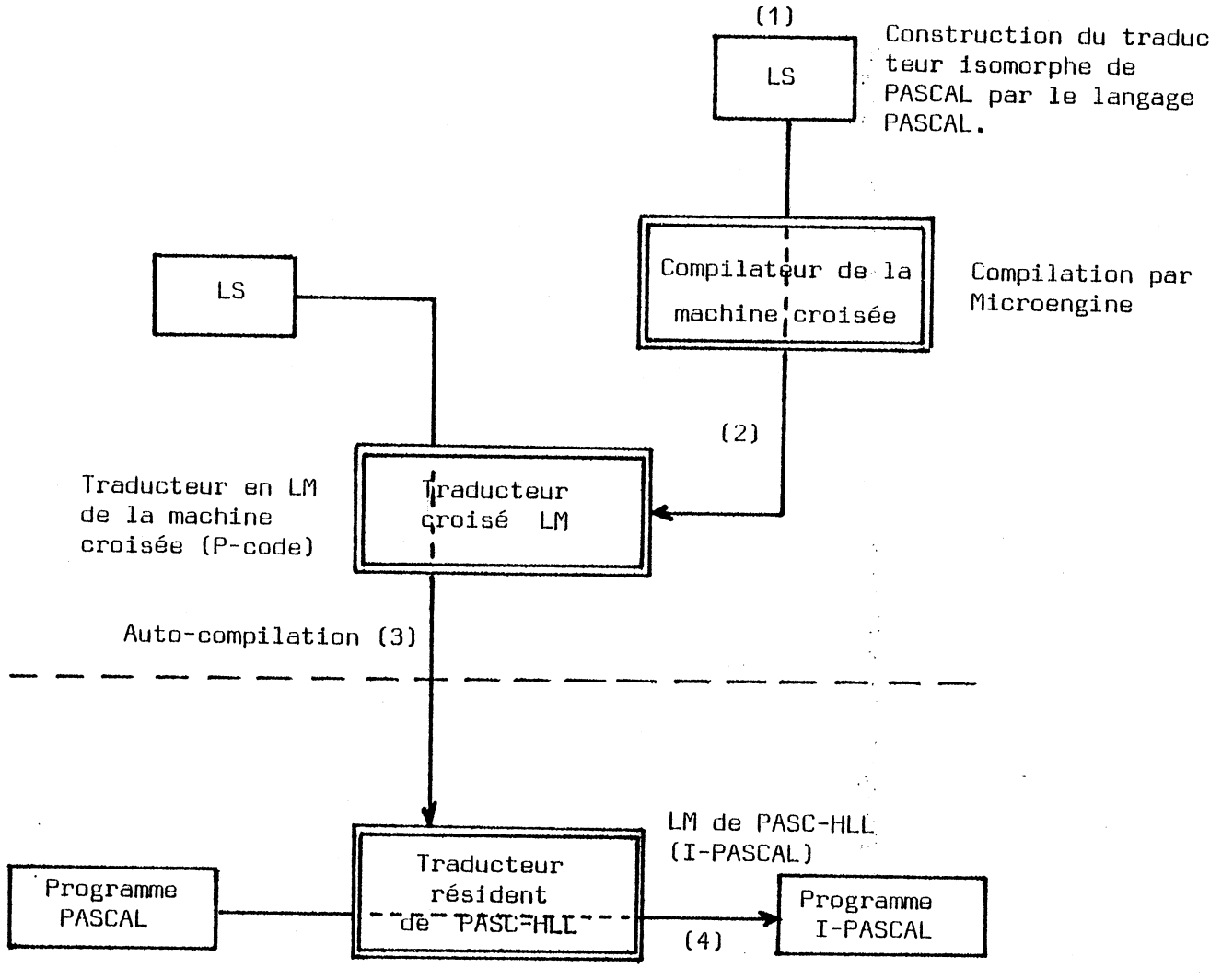


Figure 22 - Les étapes de l'auto-compilation

En réalité, il y a une différence entre le langage PASCAL d'écriture du traducteur (ici, langage PASCAL de Microengine) et le langage PASCAL acceptable par le traducteur (ici langage PASCAL acceptable par le traducteur isomorphe en I-PASCAL). On trouve deux cas :

a) certaines limites de I-PASCAL limitent la programmation de PASCAL ; en celà, I-PASCAL est un sous-ensemble du PASCAL de la machine croisée ;
b) le PASCAL de la machine croisée ne contient pas tout : ceci est dû au fait que le PASCAL de la machine croisée est un sous-ensemble de I-PASCAL.

Le premier cas provoque des problèmes lors de l'auto-compilation. Ces problèmes seront discutés au § 6.5.2.. Dans le deuxième cas, il n'y a pas de problème pendant l'auto-compilation. Mais l'efficacité du traducteur résident n'est pas bonne. Le § 6.5.3. revient sur ce deuxième cas.

6.5.2. Restrictions de I-PASCAL pour l'auto-compilation de PASCAL

L'auto-compilation (étape (3) de la figure 22) est réalisable à condition que le programme du traducteur (LS) puisse être compilé sous la structure de I-PASCAL. Autrement dit, il y a des parties qui dépendent de la nature de la machine et qui imposent des contraintes de programmation en PASCAL.

La profondeur du niveau lexical (NL) est limitée à six dans un programme I-PASCAL. Le maximum d'imbrication des blocs est donc de six niveaux. Le nombre de variables globales doit être inférieur ou égal à 256. Les variables locales et les paramètres sont permis respectivement au maximum à 64 et 62. L'espace d'adressage de I-PASCAL est indiqué par les parties d'adressage des instructions suivantes [3].

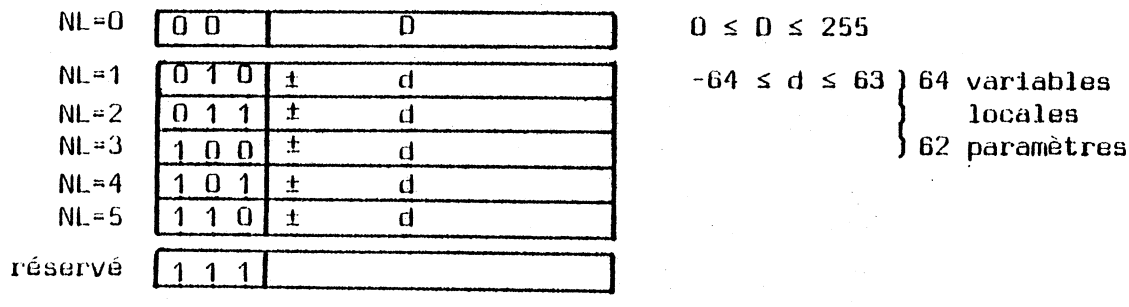


Figure 23 - Adressage de I-PASCAL

On peut implanter le test de cette limite au cours de la construction du traducteur (étape (1)). Le traducteur résident de PASCAL peut ainsi traduire les programmes PASCAL en I-PASCAL en testant le niveau lexical (NL), le nombre de variables et les paramètres. Pourtant, lors de l'étape d'amorçage, il faut que le programme du traducteur (LS) soit construit en fonction de la limitation de NL et des variables.

Dans la construction systématique du traducteur basé sur la syntaxe modifiée, il est inévitable d'adopter beaucoup de niveaux lexicaux selon la structure de la syntaxe. La figure 19 montre que NL est déjà au nombre de 6 dans la couche syntaxique. Il n'y a pas de problème pour les variables locales puisqu'il y a au maximum trois variables dans le traducteur isomorphe de PASCAL. (Même cas que pour les paramètres). Pourtant, le volume des variables globales peut risquer de dépasser la limite.

Afin de réaliser l'auto-compilation, le traducteur doit donc être construit en observant les restrictions suivantes :

a) les blocs imbriqués dans lesquels la récursivité ne se trouve pas, doivent être éliminés en les remplaçant par le programme sans procédure. (Si NL dépasse la limite malgré cette opération, l'algorithme récursif doit être modifié en algorithme itératif. Cette approche sera tout à fait contraire à l'approche systématique que nous avons présentée).
b) La réduction du nombre de variables (de paramètres également) doit être effectuée en les regroupant sous un nom de la structure de type (array, record, ...)

Les deux restrictions qui précèdent sont pourtant les parties qui détruisent la construction du traducteur isomorphe.

Les constantes doivent être précisées selon les formats de I-PASCAL pour que l'auto-compilation soit possible. Dans l'étape (2) de la figure 22, les constantes sont acceptées par le langage PASCAL de la machine croisée. Ces constantes, dans l'étape (3), doivent être également acceptables par le traducteur croisé, ce qui signifie qu'elles doivent être exprimées sous la forme de constantes de I-PASCAL. Par exemple, l'entier de PASCAL Microengine est simplement déclaré par un type entier. Cependant, il faut préciser la sorte d'entier parmi entier-8, entier-16, entier-32 ou entier-64. Si, par défaut, entier de PASCAL-Microengine correspond à entier-16 de I-PASCAL, les valeurs maximales des deux entiers doivent être équivalentes.

6.5.3. PASCAL de la machine croisée

La machine Microengine est choisie comme machine croisée pour la construction du traducteur isomorphe [10]. Il existe trois sortes de facilités de I-PASCAL que le langage PASCAL ne peut accepter :

a) le programme de contrôle non standard : loop exitif end ;
b) le passage des paramètres par procédure ou par fonction ;
c) les opérateurs supplémentaires (§ 5.2.2.) qui n'existent pas dans le langage PASCAL.

Le traducteur est donc construit sans ces facilités. Pour l'auto-compilation, il n'y a pas de problème. Mais il s'agit de l'efficacité du traducteur. Dans ce cas, les algorithmes du traducteur doivent être réalisés par les instructions inefficaces de PASCAL Microengine à cause du manque des instructions efficaces de I-PASCAL.

L'efficacité du traducteur est prise en compte de la manière suivante :

a) le traducteur croisé (étape (2)) ou résident (étape (3)) sont construits de telle manière qu'ils peuvent accepter le programme PASCAL qui contient les facilités précédentes, mais sans utiliser ces facilités dans le programme traducteur ;
b) le programme du traducteur est modifié en utilisant les facilités de I-PASCAL ;
c) le programme modifié du traducteur est compilé par le traducteur croisé ou le traducteur résident.

CONCLUSION

Le système d'interprétation d'un LHN est systématiquement analysé par les règles de décomposition. La modularité et les couches indépendantes de la machine LHN sont donc obtenues par ces règles. Les trois couches indépendantes sont progressivement décomposées par les critères définis dans cette présentation : l'interprétation du LM par le matériel, les actions sémantiques et l'analyse syntaxique.

En appliquant l'isomorphisme des langages comme premier critère de décomposition, on obtient le LM idéal pour la conception de nouvelles architectures. Ce LM satisfait les besoins de la machine LHN : traduction réversible, compilation simple, objets bijectifs entre LM et LHN, même transition d'état entre LM et LHN pendant l'exécution, décodage des identificateurs comme tâche principale du compilateur, et correspondance bijective entre les instructions du LM et les unités sémantiques du LHN.

L'approche de cette étude évite l'approche architecture d'exécution directe (interprétation totale du LHN par le matériel) en laissant une place à la compilation simple dans la machine LHN. Il est donc raisonnable de faire des efforts sur les problèmes du compilateur seulement si la réduction du coût des compilateurs (écriture, exécution du compilateur et exécution des codes compilés) compense l'augmentation des coûts de conception (du compilateur, de l'architecture matérielle et de l'implémentation matérielle).

Pour la nouvelle architecture de la machine LHN, les LM classiques isomorphes aux langages d'assemblage doivent donc être remplacés par les LM isomorphes aux LHN. Cette définition de l'isomorphisme des langages permet également de formuler les critères formels des classes des machines LHN.

L'approche de la machine LHN proposée dans notre présentation suit les étapes :
1) préparation du LM qui est isomorphe à un LHN,
2) réalisation de l'architecture qui exécute directement ce LM (LHN isomorophe),
3) construction d'un traducteur isomorphe.

Le traducteur isomorphe peut être construit par deux couches physiquement séparables : syntaxique et sémantique, en appliquant un deuxième critère de décomposition. La modularité et la régularité du traducteur isomorphe sont obtenues par la séparation physique basée sur la syntaxe modifiée. Cela donne la possibilité d'une réalisation par des systèmes VLSI.

La réalisation du traducteur isomorphe PASCAL montre qu'il y a des variables locales limitées dans une couche syntaxique (actuellement, trois variables au maximum dans un bloc). Cela permet la séparation physique entre deux couches avec une complexité négligeable. La modification syntaxique du langage PASCAL en LL(1) est réalisée par le décodage de certains identificateurs dans une classe syntaxique.

Il est montré que le langage I-PASCAL est isomorphe au langage PASCAL. Par conséquent, la machine PASC-HLL est classée dans une catégorie de machines LHN (machines PASCAL). (Les autres machines exécutant les langages intermédiaires : P-code, U-code, ..., ne peuvent pas être classés comme machines LHN).

De plus, l'extensibilité des opérateurs de I-PASCAL à n'importe quel type de données (par le microprogramme de l'opération des descripteurs), permet à I-PASCAL d'être isomorphe aux langages de très haut niveau (LTHN) comme ADA, ...

Une suite au travail que nous avons exposé pourrait comporter entre autres des études sur :

. les machines LTHN, machines bases de données, machines systèmes d'exploitation ;
. une machine multi-LHN qui accepte des langages différents et des générateurs automatiques de compilateurs isomorphes ;
. un module VLSI d'analyseur syntaxique basé sur l'unité de contrôle programmable [41], sous forme de PLA ; les attributs de ce module sont :
  - absence de partie opérationnelle (contrôle de séquence),
  - liberté des actions sémantiques remplacées par des appels.

ANNEXES

A. MODIFICATIONS SYNTAXIQUES

A.1. Syntaxe originale de PASCAL

A.2. Syntaxe déterministe avec le langage PASCAL original

A.3. Première étape de modification

A.4. Deuxième étape de modification

A.5. Troisième étape de modification

B. STRUCTURE DE I-PASCAL

B.1. Descripteurs

B.2. Instructions

B.3. Structures intermédiaire et temporaire du traducteur

ANNEXE A - MODIFICATIONS SYNTAXIQUES

Légende : mots soulignés = unité lexicale, V_T

[[ ]] = élément préfixé

« » = récurrence

$\hat{m}$ = sans modification (identique à la colonne de gauche)

A.1. Syntaxe originale de PASCAL

① <pgm> ::= <pgm heading> <block>.

② <pgm heading> ::= program <ident>
(<file ident> {, <file ident>});
<file ident> ::= <ident>

③ <block> ::= <label dec.p> <const def.p>
<type def.p> <var dec.p> <proc & fct d.p> <st.p>

④ <label dec.p> ::= ϕ | label <label> {, <label>} ;
<label> ::= <unsigned integer>

⑤ <const def.p> ::= ϕ | const <const d.> {; <const d.>} ;
<const d> ::= <ident> = <const>

⑥ <type def.p> ::= ϕ | type <type def.> {; <type def.>} ;
<type def> ::= <ident> = <type>

⑦ <var dec.p> ::= ϕ | var <var dec> {; <var dec>} ;
<var dec> ::= <ident> {, <ident>} : <type>

⑧ <proc & fct d.p> ::= {<proc or fct dec>;}

⑨ <proc or fct dec> ::= <proc dec> | <fct dec>

⑩ <proc dec> ::= <proc heading> « block »
<proc heading> ::= procedure <ident>; |
procedure <ident> (<formal para.s>{;<formal para.s>});

⑪ <fct dec> ::= <fct heading> « block »
<fct heading> ::= function <ident> : <result type>; |
function <ident> (<formal para.s>
{;<formal para.s>}) : <result type>;
<result type> ::= <type ident>

⑫ <st.p> ::= <compound st>

⑬ <type> ::= <simple type> | <struc.type> | <pointer type>

A.2. Syntaxe déterministe avec PASCAL original

① $\hat{m}$

② $\hat{m}$

③ $\hat{m}$

④ <label dec.p> ::= label <label> {, <label>} ; | ϕ
$\hat{m}$

⑤ <const def.p> ::= const <const d.> {; <const d.>} ; | ϕ
$\hat{m}$

⑥ <type def.p> ::= type <type def.> {; <type def.>} ; | ϕ
$\hat{m}$

⑦ <var dec.p> ::= var <var dec> {; <var dec>} ; | ϕ
$\hat{m}$

⑧ $\hat{m}$

⑨ <proc or fct dec> ::= [[procedure]] <proc dec> | [[function]] <fct dec>

⑩ $\hat{m}$
<proc heading> ::= procedure <ident> <T_4>
<T_4> ::= ; | (<formal para.s> {; <formal para.s>});

⑪ $\hat{m}$
<fct heading> ::= function <ident> <T_5>
<T_5> ::= : <result type>; | (<formal para.s>
{; <formal para.s>}) : <result type>;
$\hat{m}$

⑫ $\hat{m}$

⑬ <type> ::= [[1 <const> <ident>]] <simple type> | [[packed array record set file]] <struc.type> |
[[1]] <pointer type>

<simple type> ::= <scalar type> | <subrange type> | <type ident>
<scalar type> ::= (<ident> { , <ident> })
<subrange type> ::= <const> .. <const>
<type ident> ::= <ident>
<struc. type> ::= <unpacked struc. type> | packed <unpacked struc. type>
<unpacked struc. type> ::= <array type> | <record type> | <set type> | <file type>
<array type> ::= array [<index type> { , <index type> }] of <component type>
<index type> ::= <simple type>
<component type> ::= ≪ type ≫
<record type> ::= record <file list> end
<set type> ::= set of <base type>
<base type> ::= <simple type>
<file type> ::= file of ≪ type ≫
<pointer type> ::= ↑ <type ident>

⑭ <field list> ::= <fixed p.> | <fixed p.> ; <variant p> | <variant p>
<fixed p.> ::= <record sec.> { ; <record sec.> }
<record sec.> ::= <field ident> { , <field ident> } : ≪ type ≫ | ϕ
<variant p> ::= case <tag field> <type ident> of <variant> { ; <variant> }
<tag field> ::= <field ident> : | ϕ
<field ident> ::= <ident>
<variant> ::= <case label list> : (<field list>) | ϕ
<case label list> ::= <case label> { , <case label> }
<case label> ::= <const>

⑮ <formal para. s> ::= <para gr.> | var <para gr.> | function <para gr.> | procedure <ident> { , <ident> }
<para gr.> ::= <ident> { , <ident> } : <type ident>

⑯ <compound st.> ::= begin <statement> { ; <statement> } end

⑰ <statement> ::= <unlabeled st.> | <label> : <unlabeled st.>

⑱ <unlabeled st.> ::= <simple st> | <structured st>

<simple st> ::= <assignment st> | <proc. st> | <goto st> | ϕ

<simple type> ::= [( ] <scalar type> | [<const>] <subrange type> | [<ident>] <type ident>
<scalar type> ::= m̂
m̂
m̂
m̂

<unpacked struc. type> ::= [array] <array type> | [record] <record type> | [set] <set type> | [file] <file type>
m̂
m̂
m̂
m̂
m̂
m̂
m̂
m̂
m̂

⑭ <field list> ::= [case] <variant p.> | <fixed p.>

<fixed p.> ::= <record sec.> <T1>
<T1> ::= ; ≪ field list ≫ | ϕ
<record sec.> ::= m̂
<variant p.> ::= <T2> of <variant> { ; <variant> }
<T2> ::= <ident> <T3>
<T3> ::= : <type ident> | ϕ

<variant> ::= [<const>] <case label list> : (<field list>) | ϕ
m̂
m̂

⑮ <formal para. s> ::= [<ident>] <para gr.> | var <para gr.> |
m̂
m̂

⑯ m̂

⑰ <statement> ::= [<unsigned integer>] <label> : <unlabeled st.> | <unlabeled st.>

⑱ <unlabeled st.> ::= [<ident> / goto / begin / if / case / while / repeat / for / with] <simple st> | [begin / if / case / while / repeat / for / with] <structured st>

<simple st> ::= [<ident> (var) / <ident> (fct)] <assignment st.> | [<ident> (proc)]
<proc. st> | [goto] <goto st> | ϕ

<assignment st> ::= <variable> = <expr> |
<fct ident> = <expr>

⑲ <variable> ::= <entire var> | <comp. var> | <ref. var>
<entire var> ::= <var ident.>
<var ident> ::= <ident>
<comp. var> ::= <index var.> | <field desig.> | <file buffer>
<index var> ::= <array var.> [<expr> {, <expr>}]
<array var> ::= ≪ variable ≫
<field desig> ::= <record var.>. <field ident>
<record var.> ::= ≪ variable ≫
<field ident> ::= <ident>
<file buffer> ::= <file var> ↑
<file var> ::= ≪ variable ≫
<ref. var> ::= ≪ variable ≫

⑳ <proc st> ::= <proc ident> | <proc ident>
(<actual para> {, <actual para>})
<proc ident> ::= <ident>
<goto st> ::= go to <label>

㉑ <structured st> ::= ≪ compound st ≫ | <cond. st> |
<repetive st.> | <with st.>

<cond st> ::= <if st> | <case st>

㉒ <if st> ::= if <expr> then ≪ statement ≫ |
if <expr> then ≪ statement ≫ else ≪ statement ≫

㉓ <case st> ::= case <expr> of <case list elt>
{; <case list elt>} end
<case list elt> ::= <case label list> : ≪ statement ≫ | ϕ
<case label list> ::= <case label> {, <case label>}

<repetive st> ::= <while st> | <repeat st> | <for st>

㉔ <while st> ::= while <expr> do ≪ statement ≫

㉕ <repeat st> ::= repeat ≪ statement ≫ {; ≪ statement ≫}
until <expr>

㉖ <for st> ::= for <control var> := <for list> do
≪ statement ≫
<control var> ::= <ident>
<for list> ::= <initial value> to <final value> |
<initial value> downto <final value>
<initial value> ::= <expr>
<final value> ::= <expr>

<assignment st> ::= [[<ident> (var)]] <variable> = <expr> |
[[<ident> (fct)]] <fct ident> = <expr>

⑲ <variable> ::= <ident> < T6 >
< T6 > ::= [<expr> {, <expr> }] ≪ T6 ≫ | <field ident> ≪ T6 ≫ |
↑ ≪ T6 ≫ | ϕ
<field ident> ::= <ident>

⑳ <proc st> ::= <proc ident> < T12 >
< T12 > ::= [[( ]] (<actual para> {, <actual para> }) | ϕ
m̂
m̂

㉑ <structured st> ::= [[begin]] ≪ compound st ≫ | [[if / case]] <cond.st> |
[[while / repeat / for]] <repetive st.> | [[with]] <with st.>
<cond.st> ::= [[if]] <if st.> | [[case]] <case st.>

㉒ <if st> ::= if <expr> then ≪ statement ≫ < T18 >
< T18 > ::= else ≪ statement ≫ | ϕ

㉓ m̂
<case list elt> ::= [[<const>]] <case label list> : ≪ statement ≫ | ϕ
m̂
<repetive st> ::= [[while]] <while st> | [[repeat]] <repeat st> |
[[for]] <for st>

㉔ m̂

㉕ m̂

㉖ m̂
m̂
<for list> ::= <initial value> < T14 > <final value>
< T14 > ::= to | downto
m̂
m̂

(27) <with st> ::= with <rec. var. list> do «statement»

<rec. var. list> ::= <record var> {, <record var>}

(28) <expr> ::= <simple expr> | <simple expr> <rel. op> <simple expr>

<rel. op> ::= = | <> | < | <= | >= | > | in

<simple expr> ::= <term> | <sign> «term» |
<term> <add op.> «simple expr»

<sign> ::= + | −

<add op.> ::= + | − | or

<term> ::= <factor> | <factor> <mult op> «terme»

<mult op> ::= * | / | div | mod | and

<factor> ::= «variable» | <unsigned const> |
(«expr») | <fct. desig> | <set> | not «factor»

(29) <fct. desig> ::= <fct ident> | <fct ident>
(<actual para> {, <actual para>})

(30) <actual para> ::= «expr» | <variable> |
<proc. ident> | <fct. ident>

(31) <set> ::= [<elt list>]

<elt list> ::= <elt> {, <elt>} | ∅

<elt> ::= «expr» | «expr» .. «expr»

(27) $\hat{m}$

$\hat{m}$

(28) <expr> ::= <simple expr> <T_7>

<T_7> ::= [[<rel. op>]] <rel. op> <simple expr> | ϕ

<rel. op> ::= $\hat{m}$

<simple expr> ::= [[+ | −]] <sign> <term> <T_8> | <term> <T_8>

<T_8> ::= [[+ | − | or]] <add. op> «simple expr» | ϕ

<add. op> ::= $\hat{m}$

<term> ::= <factor> <T_9>

<T_9> ::= [[<mult op>]] <mult op> «term»

<mult op> ::= $\hat{m}$

<factor> ::= [[<unsigned const>]] <unsigned const> | [[( ]]
(«expr») | [[<ident> (fct)]] <fct. desig> | [[[ ]] <set> |
[[not]] not <factor> | «variable»

(29) <fct. desig> ::= <fct ident> <T_{10}>

<T_{10}> ::= [[( ]] (<actual para> {, <actual para>}) | ϕ

(30) <actual para> ::= [[<ident> (proc)]] <proc. ident> | [[<ident> (fct)]]
<fct ident> | «expr»

(31) $\hat{m}$

<elt list> ::= [[ ]]] ϕ | <elt> {, <elt>}

<elt> ::= «expr» <T_{11}>

<T_{11}> ::= .. «expr» | ϕ

A.3. Première étape de modification

① $\hat{m}$

② <pgm heading>::= program <ident>(<file ident>{,<file ident>});
<file ident>::= <ident>

③ <block>::= <label dec p><const def p><type def p>
<var dec p><proc & fct d p><statement p>

④ <label dec p>::= label <label>{,<label>};| ϕ
<label>::= <unsigned integer>

⑤ <const def p>::= const <const def>;{<const def>;}| ϕ
<const def>::= <ident>=<const>

⑥ <type def p>::= type <type def>;{<type def>;}| ϕ
<type def>::= <ident>=<type>

⑦ <var dec p>::= var <var dec>;{<var dec>;}| ϕ
<var dec>::= <ident>{,<ident>}:<type>

⑧⑨ <proc & fct dec p>::= procedure <proc d>;
≪proc & fct dec p≫| function <fct d>;
≪proc & fct dec p≫| ϕ

⑩′ <proc d>::= <ident> <p.h.D> ≪block≫
<p.h.D>::=;| (<formal para s>{;<formal para s>});

⑪′ <fct d>::= <ident> <p.h.D> ≪block≫
<p.h.D>::= :<result type>;| (<formal para s>
{;<formal para s>}):<result type>;
<result type>::= <type ident>

⑫ <statement p>::= <compound st>

⑬ <type>::= (<ident>{,<ident>})|<const>..<const>|<ident>|
↑<type ident>|packed <unpacked str type>|
<unpacked str type>

A.4. Deuxième étape de modification

④ $\hat{m}$

② <pgm heading>::= program <ident>(<ident>{,<ident>});

③ <block>::= label <label d> l_1 | l_1
l_1::= const <const d> l_2 | l_2
l_2::= type <type d> l_3 | l_3
l_3::= var <var d> l_4 | l_4
⑧⑨ l_4::= procedure <proc d>; l_4 | function <fct d>; l_4 | l_5
③⑫ l_5::= <compound st>

④ <label d>::= <unsigned integer>{,<unsigned integer>};

⑤ <const d>::= <ident>=<const>{;<ident>=<const>};

⑥ <type d>::= <ident>=<type>{;<ident>=<type>};

⑦ <var d>::=<ident>{,<ident>}:<type>{;<ident>{,<ident>}
:<type>};

⑩′ <proc d>::=<ident> l_6 ≪block≫
l_6::=;| (<formal para s.>{;<formal para s.>});

⑪′ <fct d>::=<ident> l_7 ≪block≫
l_7::=:<ident>;| (<formal para. s>{;<formal para.s>})
:<ident>

⑬ <type>::= (<ident>{,<ident>})|<const>..<const>|<ident>|
↑<ident>|packed l_8 | l_8

<unpacked str.type> ::= array [<index type>{,<index type>}]
of <comp.type> | record <field list> end |
set of <base type> | file of ≪type≫

<type ident> ::= <ident>

<index type> ::= <simple type>

<comp. type> ::= ≪type≫

<base type> ::= <simple type>

<simple type> ::= (<ident>{,<ident>}) | <const>..<const> | <ident>

⑭ <field list> ::= <ident> <T1> ≪field list≫ | ;≪field list≫ |
case <ident> <T2> <T3> | ∅

<T1> ::= , <ident>{,<ident>} ≪type≫ | : ≪type≫

<T2> ::= : <ident> of | of

<T3> ::= <const>{,<const>}: (≪field list≫) ≪T3≫ |
; ≪T3≫ | ∅

⑮ <formal para. s> ::= var <para gr> | function <para gr> |
procedure <ident>{,<ident>} | <para gr>

<para gr> ::= <ident>{,<ident>}: <type ident>

⑯ m̂

⑰ <statement> ::= <label>: <unlabeled st> | <unlabeled st>

⑱㉑ <unlabeled st> ::= [[<ident>]]IDKINDS([[ident(VARS)]]
<assgt. st> | [[ident(FUNC)]] <assgt. fct. st> | [[ident(PROC)]]
<ident> <proc. st>) | goto <goto st> | begin
≪statement≫ {;≪statement≫} end | if <if. st> |
case <case st> | while <while st> | repeat <repeat st> |
for <for st> | with <with st> | ∅

<assgt. st> ::= <variable> = <expr>

<assgt. fct. st> ::= <fct. ident> = <expr>

⑲ <variable> ::= [<expr>{,<expr>}] ≪variable≫ |
. <ident> ≪variable≫ | ↑ ≪variable≫ | ∅

⑳ <proc. st> ::= (<actual para>{;<actual para>}) | ∅

<goto st> ::= <label>

㉒ <if st> ::= <expr> then ≪statement≫ <if T>

<if T> ::= else ≪statement≫ | ∅

㉓ <case st> ::= <expr> of <case list elt>{;<case list elt>} end

<case list elt> ::= <const>{,<const>}: ≪statement≫ | ∅

㉔ <while st> ::= <expr> do ≪statement≫

ℓ8 ::= array [<simple type>{,<simple type>}] of ≪type≫ |
record <file list> end | set of <simple type> |
file of ≪type≫

<simple type> ::= (<ident>{,<ident>}) | <const>..<const> | <ident>

⑭ m̂
m̂
m̂
m̂
m̂
m̂

⑮ m̂
m̂

<para gr> ::= <ident>{,<ident>}: <ident>

⑯ annexé dans ⑱

⑰ <statement> ::= <unsigned integer>: <unlabeled st> |
<unlabeled st>

⑱㉑ m̂
m̂
m̂
m̂
m̂
m̂
m̂

<assgt. fct. st> ::= <ident> = <expr>

⑲ m̂
m̂

⑳ m̂
m̂

㉒ m̂
m̂

㉓ m̂
m̂

㉔ m̂

(25) <repeat st>::= ≪statement≫ {, ≪statement≫} until <expr>

(26) <for st>::= <control var> := <for list> do ≪statement≫

<control var>::= <ident>

<for list>::= <initial value> <for T>

<for T>::= to <final value> | downto <final value>

<initial value>::= <expr>

<final value>::= <expr>

(27) <with st>::= with <rec.var.list> do ≪statement≫

<rec.var.list>::= <rec.var> {<rec.var>}

<rec.var>::= <variable>

(28) <expr>::= <simple expr> <simple T>

<simple T>::= <rel.op> <simple expr> | ∅

<simple expr>::= ± <term> <term T> | <term> <term T>

<term T>::= <add.op> ≪simple expr≫

<term>::= <factor> <factor T>

<factor T>::= <mult.op> ≪term≫ | ∅

<factor>::= [[<ident>]] IDKINDS ([[ident(VARS)]] <variable> | [[ident(FUNC)]] <fct desig>) | <unsigned integer> | (≪expr≫) | <elt list> | not ≪factor≫

(29) <fct desig>::= (<actual para> {, <actual para>}) | ∅

(30) <actual para>::= [[<ident>]] IDKINDS (ident(PROC) | ident(FUNC)) | ≪expr≫

(31) <elt list>::=] | <elt> {, <elt>}]

<elt>::= ≪expr≫ <elt T>

<elt T>::= .. <expr> | ∅

(25) m̂

(26) <for st>::= <ident> := <expr> <for T> <expr> do ≪statement≫

<for T>::= to | downto

(27) <with st>::= with <ident> <variable> {, <ident> <variable>} do ≪statement≫

(28) m̂ m̂ m̂ m̂ m̂ m̂ m̂ m̂ m̂ m̂ m̂

(29) m̂

(30) m̂ m̂

(31) m̂ m̂ m̂

A.5. Troisième étape de modification

de ① à ⑪ , même que A.4

⑬ <type> ::= [(<ident> | + | − <unsigned no>)] <simple type> | ↑<ident> | packed l_8 | l_8

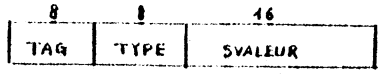
l_8 ::= array [<simple type> {, <simple type>}] of ≪type≫ | record <field list> end |
set of <simple type> | file of ≪type≫

<simple type> ::= (<ident> {, <ident>}) | [[<ident>]] IDKINDS((ident(TYPES) | <const>..<const>))

de ⑭ à ㉑ , même que A.4

ANNEXE B - STRUCTURE DE I-PASCAL

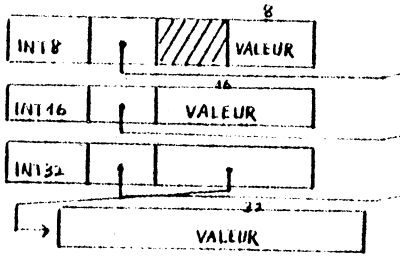
B.1. Descripteurs



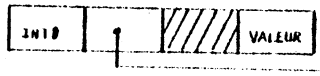
*: si TYPE=0 alors
type matériel
sinon <subrange>

Descripteur de variable

1. entier

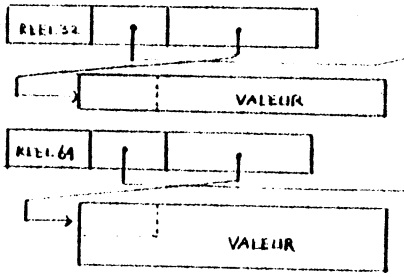


2. scalar

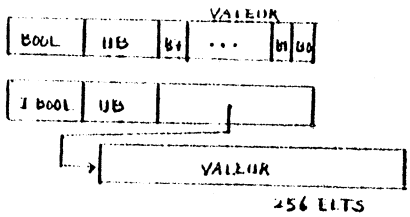


ex) A : JOUR

3. réel

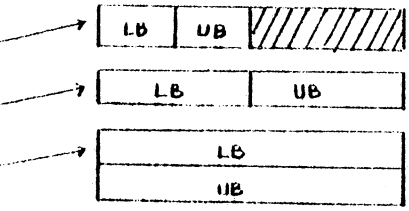


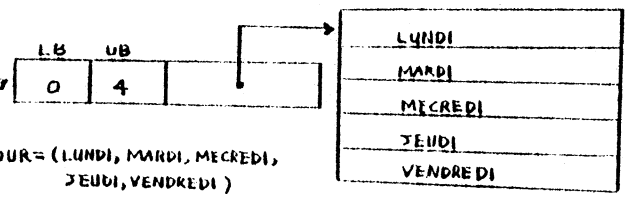
4. booléen



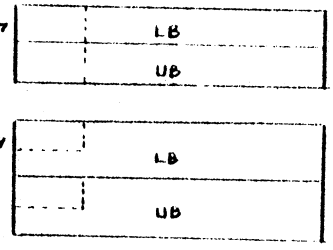
256 ELTS

Descripteur de type

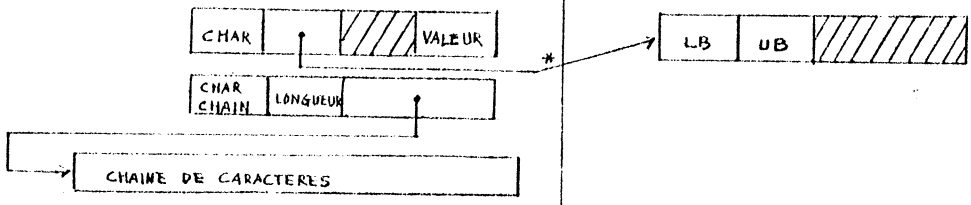




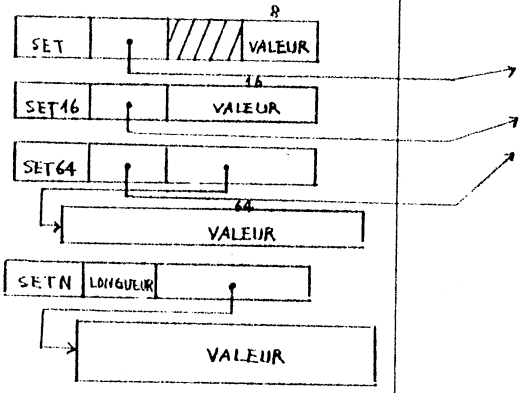
JOUR = (LUNDI, MARDI, MECREDI, JEUDI, VENDREDI)



5. caractère

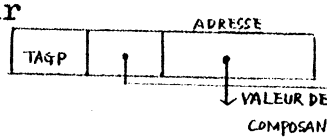


6. ensemble



descripteur de type

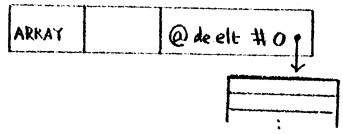
7. pointeur



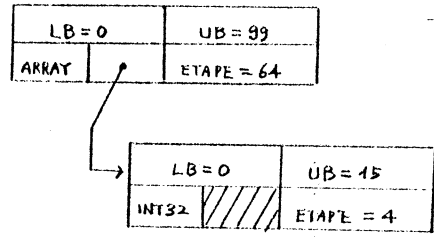
descripteur de structure

STRUCTURE

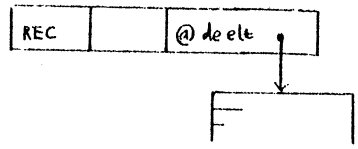
1. array

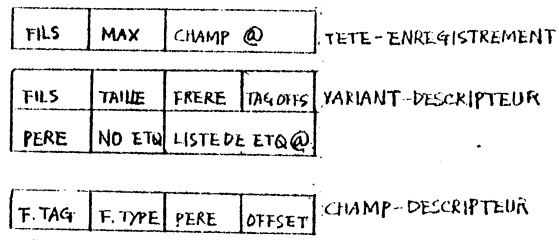


ex) A : array [0..99] of array [0..15] of integer

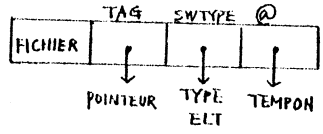


2. record

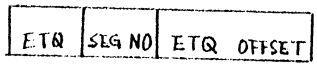




3. fichier



4. procédure



B.2. Instructions

if <expr> then <stat>	<expr> THEN (ℓ) <stat> EXIT ↑	THEN (m) 000001 m
if <expr> then <stat1> else <stat2>	<expr> THEN(ℓ) <stat1> NEHT(ℓ) ELSE <stat2> EXIT ↓	NEHT(m) 110101 m
		ELSE 1101 1000
		EXIT 1010 1100
for i:=<exp1> to <exp2> do <stat>	<exp1> <exp2> FORUP(ℓ) ASS(i) <stat> ROFUP ↑	FORUP(m) 010100 m
for i:=<exp1> downto <exp2> do <stat>	<exp1> <exp2> FORDOWN(ℓ) ASS(i) <stat> ROFDOWN ↑	FORDOWN(m) 010101 m
while <expr> do <stat>	LOOP(ℓ) <exp> WHILE <stat> ENDLOOP ↑	ROFUP 0001 0100
repeat <stat> until <expr>	LOOP(ℓ) <stat> <exp> UNTIL ↑	ROFDOWN 0001 0110
loop <stat1> exitif(<exp>) <stat2> end	LOOP(ℓ) <stat1> <exp> EXITIF <stat2> ENDLOOP ↑	LOOP(m) 111 110 m
		ENDLOOP 1100 1000
case <exp> of $V_1^1, \dots, V_1^{n_1}$: <stat1> ;	<exp> CASE(ℓ) IS(V_1^1) ··· IS($V_1^{n_1}$) OF(ℓ) <stat1> FO	EXIT IF 0000 1000
$V_2^1, \dots, V_2^{n_2}$: <stat2> ;	IS(V_2^1) ··· IS($V_2^{n_2}$) OF(ℓ) <stat2> FO	WHILE 0000 1010
⋮ ⋮	⋮ ⋮	UNTIL 0000 1100
$V_p^1, \dots, V_p^{n_p}$: <statp> ;	IS(V_p^1) ··· IS($V_p^{n_p}$) OF(ℓ) <statp> FO	ASS(s,d) 100 000 (s,d)
end	ESAC ↑	CASE(m) 000 100 m
		OF(m) 101 000 m
		FO 0100 0100
		ESAC 0100 1000
goto ℓ	EXITFOR(NFOR, NSEG) , GOTO(dep.)	EXITFOR(m,n) 1001 0100 n m
		GOTO(m) 111 101 m
proc ;	CALL(proc) , CALLF(fct)	CALL(s,d) 101 010 (s,d)
fct ;	ENTER	CALLF(s,d) 101 100 (s,d)
proc(<exp1>, ..., <expn>) ;	CALL(proc) <exp1> PARAM(sp, -3)	ENTER 1001 1000
	⋮ ⋮	PARAM(s,d) 100 001 1111 d
	<expn> PARAM(sp, -(2+n))	RETURN 1001 1100
	ENTER	RETURNF 1001 1110
	RETURN , RETURNF	WITH(n) 1010 0100 n
with <rec.var> do <stat>	REF(rec.var.) ASS(w,0) WITH(+1)	ASS(w,i) 100 000 1110 i
with <R1> <R2> ··· <Rn> do <stat>	<stat> WITH(-1)	REF(w,i) 110 000 1110 i
	REF(R1) ASS(w,0)	
	REF(R2) ASS(w,1)	
	⋮	
	REF(Rn) ASS(w,n-1)	
	WITH(+n)	
	<stat>	
	WITH(-n)	
v := <expr>	<expr> ASS(V)	ASS(s,d) 100 000 (s,d)
<adr.exp> := <valeur.exp>	<adr.exp> <valeur exp> ASSA	ASSA 0101 1000
v := v+1	INCV(V)	INCV(s,d) 100 010 (s,d)
v := v-1	DECV(V)	DECV(s,d) 100 011 (s,d)
v := 0	CLRV(V)	CLRV(s,d) 101 110 (s,d)
v := 1	SETV(V)	SETV(s,d) 101 111 (s,d)
<EA> := <EA> + 1	<EA> INCA	INCA 0001 1100
<EA> := <EA> - 1	<EA> DECA	DECA 0001 1101
<EA> := 0	<EA> CLRA	CLRA 0001 1110
<EA> := 1	<EA> SETA	SETA 0001 1111
<EA> := n	<EA> ASSI(n)	ASSI(n) 0001 1011 n
<EA> := <EA> + n	<EA> INCAI(n)	INCAI(n) 0001 1000 n
<EA> := <EA> - n	<EA> DECAI(n)	DECAI(n) 0001 1001 n
<EA>[n] := <exp>	<EA> <exp> ASSXI(n)	ASSXI(n) 0001 1010 n
	REF(s,d)	REF(s,d) 110 000 (s,d)
	INDD(s,d)	INDD(s,d) 110 001 (s,d)

	HEX	
ZERO-INT	D 0	
ZERO-SET	D 1	
ONE-INT	E 0	
INT8(v)	E 0	v
SET8(v)	E 1	v
CHAR(v)	E 2	CHAR-CODE
INT16(v)	E 4	v
SET16(v)	E 5	v
INT32(@)	E 8	@
SET32(@)	E 9	@
REAL32(@)	E B	@
INT64(@)	E C	@
REAL64(@)	E F	@
SETV(L,@)	E D	L
	@	
CS(L,@)	E F	L
	@	

ADD	6 0
SUB	6 1
MULT	6 2
DIV	6 3
IDIV	6 4
MOD	6 6
ODD	2 8
INC	2 0
DEC	2 1
INCI(n)	2 4 n
DECI(n)	2 6 n
NEG	2 8
ABS	2 9
TRUNC	2 A
INT	2 2
CHAR	2 3

INDX	4 0	
INXI(P)	0 0	P
FIELD(P)	3 C	P

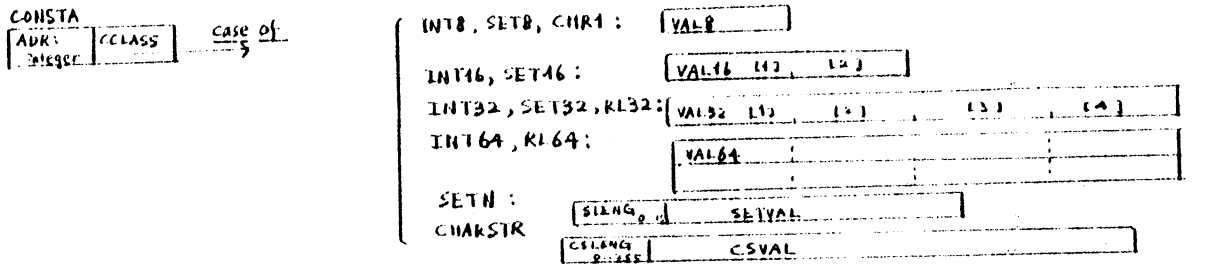
EQ	7 0
NEQ	7 1
LT	7 4
LE	7 5
GT	7 6
GE	7 7
EQ0	3 0
LT0	3 5
LE0	3 4
NEQ0	3 1
GE0	3 6
GT0	3 7
EQ1	3 2
LT1	3 9
LE1	3 8
NEQ1	3 3
GE1	3 C
GT1	3 D

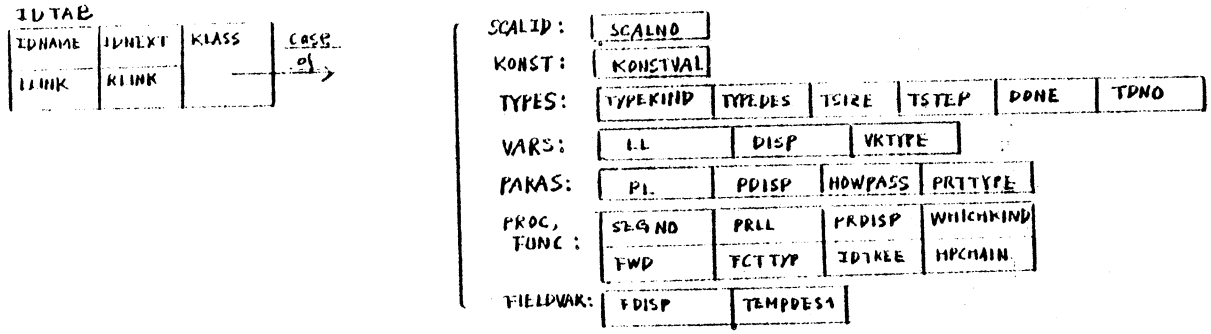
AND	6 8
OR	6 9
XOR	6 A
XAND	6 B
NAND	6 C
NOR	6 D
XNOR	6 E
XNAND	6 F
NOT	2 C

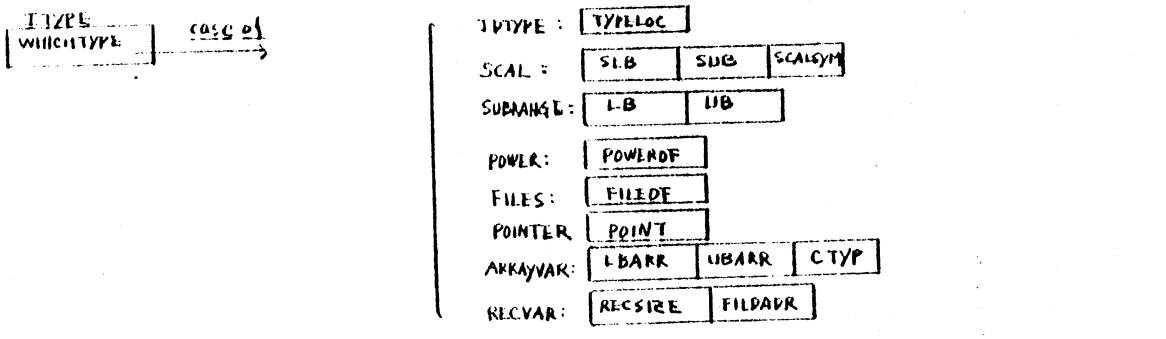
IN	7 C
ELEM	7 8
INTER	4 C

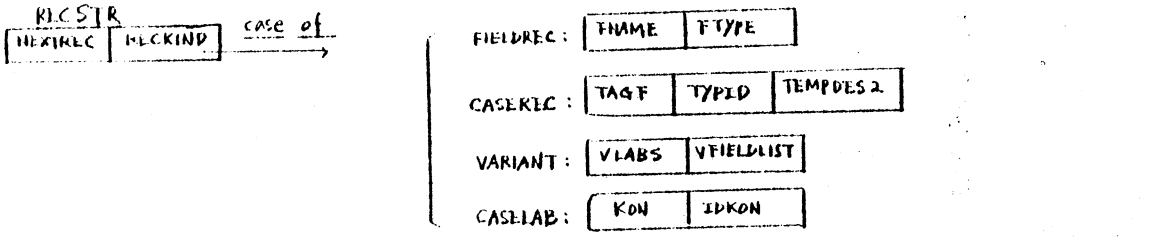
POINTER	2 E

B.3. Structures intermédiaire et temporaire du traducteur









BIBLIOGRAPHIE

[1] E.W. DIJKSTRA, *A discipline of programming*, Prentice Hall Inc., 1976.

[2] J.P. SCHOELLKOPF, *Machine PASC-HLL : définition d'une architecture pipeline pour une unité centrale adaptée au langage PASCAL*, Thèse de Doctorat de 3ème Cycle, INPG, Grenoble, Juin 1977.

[3] J.P. SCHOELLKOPF, *A tutorial on the high-level language machine design for PASCAL*, ENSIMAG, Rapport de Recherche n° 131, Octobre 1978.

[4] H. ZIMMERMANN, *OSI reference model*, IEEE Transactions on Communications, vol COM.28, n° 4, Avril 1980.

[5] C. MEAD & L. CONWAY, *Introduction to VLSI systems*, Addison Wesley, 1980.

[6] G. GORDON, *System simulation*, 2ème Edition, Prentice Hall Inc., 1978.

[7] IRIA, *SIRIUS*, Bulletin de Liaison de la Recherche en Informatique et Automatique, n° 57, Octobre 1979.

[8] C.R. CARLSON, *A survey of high-level language computer architecture*, in High-Level Language Computer Architecture, Academic Press, 1975.

[9] D.M. BULMAN, *Stack Computers*, IEEE Computers, Mai 1977.

[10] Microengine Corp., *Pascal Microengine computer operational manual*, Mars 197

[11] Y. CHU, *Architecture of hardware data interpreter*, IEEE Transactions on Computers, vol C.28, n° 2, Février 1979.

[12] Y. CHU, *A LSI modular direct execution computer organization*, IEEE Computers, Juillet 1978.

[13] T.H. KEHL, *BASIL architecture. A high-level language architecture*, ACM, SIGARCH, vol 4, n° 4, Janvier 1976.

[14] P.A. NELSON, *A comparison of Pascal intermediate language*, Proceedings ACM, SIGPLAG, Symposium on Compiler Construction, Denver, 1979.

[15] J.E. HOPCROFT & J.D. ULLMAN, *Formal languages and their relation to automata*, Addison Wesley, 1969.

[16] Y. CHU, *Concept of high-level language computer architecture*, in High-Level Language Computer Architecture, Academic Press, 1975.

[17] Y. CHU, *High-level computer architecture*, IEEE Computers, Juillet 1981.

[18] D.B. WORTMAN, *Language directed computer design*, International Workshop on Computer Architecture, Grenoble, Juin 1973.

[19] M.J. FLYNN, *Direction and issues in architecture and language*, IEEE Comput., Octobre 1980.

[20] M.J. FOSTER & H.T. KUNG, *The design of special purpose VLSI chips*, IEEE Comput., Janvier 1980.

[21] A.V. AHO & S.C. JHONSON, *LR parsing*, Comput. Survey, vol 6, n° 2, Juin 1974.

[22] R.D. DANNENBERG, *An architecture with many operand registers to efficiently execute block-oriented language*, ACM SIGARCH Newsletters, vol 7, n° 6, Avril 1979.

[23] T.A. WELCH, *An investigation of descriptor oriented architecture*, ACM SIGARCH, vol 4, n° 4, Janvier 1976.

[24] P. WEGNER, *Programming with ADA*, Technical contribution.

[25] E.A. FEUSTAL, *Tagged architecture and semantics of programming language*, ACM SIGARCH, vol 4, n° 4, Janvier 1976.

[26] A.P. BATSON, *Design data for ALGOL 60 machine*, ACM SIGARCH, vol 4, n° 4, Janvier 1976.

[27] T.G. RAUCHER & P.N. ADAMS, *Microprogramming : a tutorial and survey of recent developments*, IEEE Transactions on Computers, vol C.29, n° 1, Janvier 1980.

[28] M.B. VINEBERG & A. VIZIENIS, *Implementation of high-level language on an array machine*, International Workshop on Computing Architecture, Grenoble, Juin 1973.

[29] E. de MASSAS, *Etude, réalisation et utilisation d'outils logiciels adaptés à l'écriture parallèle des algorithmes*, Thèse de Doctorat de 3ème Cycle, INPG, Grenoble, Septembre 1978.

[30] S.T. FU, *Elements of modern algebra*, Holden day, Inc., 1965.

[31] A.V. AHO & J.D. ULLMAN, *The theory of parsing, translation and compiling*, Vol 1 : Parsing. Prentice Hall Inc., 1972.

[32] J. COHEN & R. SITVER, *A case study in program translation : translation into Polish*, IEEE Transactions on Software Engineering, vol SE.5, n° 6, Novembre 1979.

[33] S. FOURNIER & M.T. LIU, *System design of grammar programmable high-level language machine*, ACM SIGARCH, vol 4, n° 4, Janvier 1976.

[34] E.D. KNUTH, *Top-down syntax analysis*, Acta Informatica, vol 1, Mars 1971.

[35] J. COHEN, R. SITVER & D. AUTY, *Evaluation and improving recursive descent parsers,* IEEE Transactions on Software Engineering, vol SE.5, n° 5, Septembre 1979.

[36] K. JENSEN & N. WIRTH, *Pascal users manual and report,* Springer Verlag, 1974

[37] P.R. MA & L. LEWIS, *On the design of a microcode compiler for a machine-independent high-level language,* IEEE Transactions on Software Engineering, vol SE.7, n° 3, Mai 1981.

[38] D. PERKINS & R.L. SITES, *Machine independent Pascal code optimization,* ACM Proceedings SIGPLAG, Symposium on Compiler Construction, Denver, 1979.

[39] G.J. SUSSMAN, J. HOLLOWAY, G.L. STEELE & A. BELL, *SCHEME-79 Lisp on a chip,* IEEE Computer, Juillet 1981.

[40] C. WETHERELL & A. SHANNON, *LR automatic parser generator and LR(1) parser,* IEEE Transactions on Software Engineering, vol SE.7, n° 3, Mai 1981.

[41] B.I. DERVISOGLU & P.J. CRISCIONE, *A hard programmable control unit design using VLSI technology,* IEEE Trans. Comput., vol C 30, n° 10, Octobre 1981.

dernière page de la thèse

AUTORISATION DE SOUTENANCE

VU les dispositions de l'article 3 de l'arrêté du 16 avril 1974,

VU les rapports de présentation de Messieurs

. L. BOLLIET, Professeur

. GUIBOUD RIBAUD, Ingénieur

Monsieur PARK Seung Kyu

est autorisé à présenter une thèse en soutenance pour l'obtention du diplôme de DOCTEUR-INGENIEUR, Spécialité "Informatique".

Fait à Grenoble, le 22 mars 1982

Le Président de l'I.N.P.-G.

D. BLOCH
Président
de l'Institut National Polytechnique
de Grenoble
P.O. le Vice-Président,