Un système de production automatique de générateur de code

Miguel Santana

▶ To cite this version:

Miguel Santana. Un système de production automatique de générateur de code. Modélisation et simulation. Université Joseph-Fourier - Grenoble I, 1983. Français. NNT : . tel-00308471

HAL Id: tel-00308471
https://theses.hal.science/tel-00308471

Submitted on 30 Jul 2008

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THESE

présentée à

l'Université Scientifique et Médicale de Grenoble

pour obtenir le grade de
DOCTEUR DE 3ème CYCLE
«Informatique»

par

Miguel SANTANA

UN SYSTEME DE PRODUCTION AUTOMATIQUE DE GENERATEURS DE CODE.

Thèse soutenue le 21 décembre 1983 devant la commission d'examen.

J. MOSSIERE	Président
S. KRAKOWIAK	
B. LORHO	Examinateurs
B. ROUGEOT	

UNIVERSITE SCIENTIFIQUE ET MEDICALE DE GRENOBLE

Année universitaire 1982-1983

Président de l'Université : M. TANCHE

MEMBRES DU CORPS ENSEIGNANT DE L'U.S.M.G.

(RANG A)

SAUF ENSEIGNANTS EN MEDECINE ET PHARMACIE

PROFESSEURS DE 1ère CLASSE

ARNAUD Paul	Chimie organique
ARVIEU Robert	Physique nucléaire I.S.N.
AUBERT Guy	Physique C.N.R.S.
AYANT Yves	Physique approfondie
BARBIER Marie-Jeanne	Electrochimie
BARBIER Jean-Claude	Physique expérimentale C.N.R.S. (labo de magnétisme)
BARJON Robert	Physique nucléaire I.S.N.
BARNOUD Fernand	Biosynthèse de la cellulose-Biologie
BARRA Jean-René	Statistiques - Mathématiques appliquées
BELORISKY Elie	Physique
BENZAKEN Claude (M.)	Mathématiques pures
BERNARD Alain	Mathématiques pures
BERTRANDIAS Françoise	Mathématiques pures
BERTRANDIAS Jean-Paul	Mathématiques pures
BILLET Jean	Géographie
BONNIER Jean-Marie	Chimie générale
BOUCHEZ Robert	Physique nucléaire I.S.N.
BRAVARD Yves	Géographie
CARLIER Georges	Biologie végétale
CAUQUIS Georges	Chimie organique
CHIBON Pierre	Biologie animale
COLIN DE VERDIERE Yves	Mathématiques pures
CRABBE Pierre (détaché)	C.E.R.M.O.
CYROT Michel	Physique du solide
DAUMAS Max	Géographie
DEBELMAS Jacques	Géologie générale
DEGRANGE Charles	Zoologie
DELOBEL Claude (M.)	M.I.A.G. Mathématiques appliquées
DEPORTES Charles	Chimie minérale
DESRE Pierre	Electrochimie
DOLIQUE Jean-Michel	Physique des plasmas
DUCROS Pierre	Cristallographie
FONTAINE Jean-Marc	Mathématiques pures
GAGNAIRE Didier	Chimie physique

.../...

GASTINEL Noël	Analyse numérique - Mathématiques appliquées
GERBER Robert	Mathématiques pures
GERMAIN Jean-Pierre	Mécanique
GIRAUD Pierre	Géologie
IDELMAN Simon	Physiologie animale
JANIN Bernard	Géographie
JOLY Jean-René	Mathématiques pures
JULLIEN Pierre	Mathématiques appliquées
KAHANE André (détaché DAFCO)	Physique
KAHANE Josette	Physique
KOSZUL Jean-Louis	Mathématiques pures
KRAKOWIAK Sacha	Mathématiques appliquées
KUPTA Yvon	Mathématiques pures
LACAZE Albert	Thermodynamique
LAJZEROWICZ Jeannine	Physique
LAJZEROWICZ Joseph	Physique
LAURENT Pierre	Mathématiques appliquées
DE LEIRIS Joël	Biologie
LLIBOUTRY Louis	Géophysique
LOISEAUX Jean-Marie	Sciences nucléaires I.S.N.
LOUP Jean	Géographie
MACHE Régis	Physiologie végétale
MAYNARD Roger	Physique du solide
MICHEL Robert	Minéralogie et pétrographie (géologie)
MOZIERES Philippe	Spectrométrie - Physique
OMONT Alain	Astrophysique
OZENDA Paul	Botanique (biologie végétale)
PAYAN Jean-Jacques (détaché)	Mathématiques pures
PEBAY PEYROULA Jean-Claude	Physique
PERRIAUX Jacques	Géologie
PERRIER Guy	Géophysique
PIERRARD Jean-Marie	Mécanique
RASSAT André	Chimie systématique
RENARD Michel	Thermodynamique
RICHARD Lucien	Biologie végétale
RINAUDO Marguerite	Chimie CERMAV
SENGEL Philippe	Biologie animale
SERGERAERT Francis	Mathématiques pures
SOUTIF Michel	Physique
VAILLANT François	Zoologie
VALENTIN Jacques	Physique nucléaire I.S.N.
VAN CUTSEN Bernard	Mathématiques appliquées
VAUQUOIS Bernard	Mathématiques appliquées
VIALON Pierre	Géologie

PROFESSEURS DE 2ème CLASSE

ADIBA Michel	Mathématiques pures
ARMAND Gilbert	Géographie

.../...

AURIAULT Jean-Louis	Mécanique
BEGUIN Claude (M.)	Chimie organique
BOEHLER Jean-Paul	Mécanique
BOITET Christian	Mathématiques appliquées
BORNAREL Jean	Physique
BRUN Gilbert	Biologie
CASTAING Bernard	Physique
CHARDON Michel	Géographie
COHENADDAD Jean-Pierre	Physique
DENEUVILLE Alain	Physique
DEPASSEL Roger	Mécanique des fluides
DOUCE Roland	Physiologie végétale
DUFRESNOY Alain	Mathématiques pures
GASPARD François	Physique
GAUTRON René	Chimie
GIDON Maurice	Géologie
GIGNOUX Claude (M.)	Sciences nucléaires I.S.N.
GUITTON Jacques	Chimie
HACQUES Gérard	Mathématiques appliquées
HERBIN Jacky	Géographie
HICTER Pierre	Chimie
JOSELEAU Jean-Paul	Biochimie
KERCKOVÊ Claude (M.)	Géologie
LE BRETON Alain	Mathématiques appliquées
LONGEQUEUE Nicole	Sciences nucléaires I.S.N.
LUCAS Robert	Physiques
LUNA Domingo	Mathématiques pures
MASCLE Georges	Géologie
NEMOZ Alain	Thermodynamique (CNRS - CRTBT)
OUDET Bruno	Mathématiques appliquées
PELMONT Jean	Biochimie
PERRIN Claude (M.)	Sciences nucléaires I.S.N.
PFISTER Jean-Claude (détaché)	Physique du solide
PIBOULE Michel	Géologie
PIERRE Jean-Louis	Chimie organique
RAYNAUD Hervé	Mathématiques appliquées
ROBERT Gilles	Mathématiques pures
ROBERT Jean-Bernard	Chimie physique
ROSSI André	Physiologie végétale
SAKAROVITCH Michel	Mathématiques appliquées
SARROT REYNAUD Jean	Géologie
SAXOD Raymond	Biologie animale
SOUTIF Jeanne	Physique
SCHOOL Pierre-Claude	Mathématiques appliquées
STUTZ Pierre	Mécanique
SUBRA Robert	Chimie
VIDAL Michel	Chimie organique
VIVIAN Robert	Géographie

à Orleta

Je tiens à remercier

Monsieur J.Mossière, Directeur du Laboratoire de Génie Informatique, de m'avoir fait l'honneur de présider le jury de cette thèse. Qu'il me soit permis de lui témoigner ma plus profonde gratitude pour les précieux conseils qu'il m'a donnés lors de la rédaction de ce document.

Monsieur S. Krakowiak, Professeur à l'Université de Grenoble et Directeur de cette thèse, dont les idées et les encouragements ont largement contribué à la réussite de ce projet.

Messieurs B. Lorho, Professeur à l'Université d'Orléans, et B. Rougeot, Ingénieur au CNET Lannion, qui ont bien voulu faire partie de ce jury.

Monsieur E. André, Directeur du projet CONCERTO, pour l'intérêt qu'il a porté à ce travail.

Monsieur J. C. Hochain, mon coéquipier dans cette étude, qui a participé à tous les instants de ce travail et contribué par ses critiques et son aide matérielle à l'élaboration de ce document.

Tous les membres de l'équipe ADELE avec qui j'ai pu travailler et échanger des idées, particulièrement à B. Cassagne qui a participé activement à ce projet et qui a guidé mes premiers pas dans la recherche.

Miguel SANTANA

Ce travail a été partiellement financé, d'une part, par l'ADI et, d'autre part, par le CNET Lannion dans le cadre du projet CONCERTO.

TABLE DES MATIERES
-0-0-0-0-

I. INTRODUCTION. 1

PREMIERE PARTIE

II. LA GENERATION DE CODE : ETAT DE L'ART. 7

1. Présentation du problème. 7

2. Solution traditionnelle. 9

3. Le problème des NxM compilateurs. 9

4. Formalisation de la génération de code. 11

5. Automatisation de la production de générateurs 13
 5.1. L'approche procédurale. 14
 5.1.1. La macro-génération. 15
 5.1.2. L'interprétation pure. 16
 5.1.3. Les automates d'états finis. 18
 5.1.4. L'isolation des fonctions dépendantes de la machine. 20
 5.2. L'approche descriptive. 20
 5.2.1. Simulation. 21
 5.2.2. Grammaires à attributs. 23
 5.2.3. Reconnaissance de modèles arborescents. 28
 5.2.4. Autres. 36

6. Conclusion. 38

III. UNE SOLUTION A LA GENERATION DE CODE MULTICIBLE. 41

1. Cadre de travail. 41

2. Objectifs. 43

3. Notre méthode de génération de code. 46
 3.1. Concepts préliminaires. 46
 3.1.1. Notion de coût. 46
 3.1.2. Descripteur de valeur. 47
 3.1.3. Modèle de descripteur. 49
 3.1.4. Conformité entre un descripteur et un modèle. 50
 3.1.5. Opération. 50
 3.1.6. Possibilité d'opération. 51
 3.2. Description de la méthode. 52
 3.3. Modèles de génération. 56
 3.4. Sélection d'instructions pour une opération. 57

4. GEMME : Un GEnérateur de code Multi-MachinE. 61
4.1. Présentation du système de génération de code. 61
4.1.1. Organisation générale. 61
4.1.2. Description de la machine cible. 62
4.1.3. Construction d'un générateur de code. 63
4.1.4. Compilation d'un programme Pascal. 64
4.2. Architecture générale d'un générateur de code. 64

IV. OPTIMISATION DU CODE GENERE. 67

1. Revue des techniques d'optimisation. 68
1.1. Séquences d'instructions redondantes. 68
1.2. Relations créées par les branchements. 70
1.3. Simplifications algébriques. 74
1.4. Instructions et modes d'adressage plus efficaces. 75

2. Techniques appliquées par GEMME. 76

DEUXIEME PARTIE

V. DESCRIPTION DES MACHINES. 81

1. Modèle de description. 81
1.1. Type de données. 81
1.2. Ressource. 82
1.3. Modèle de descripteur. 83
1.4. Classe de modèles. 83
1.5. Formats de génération. 84
1.6. Instructions. 84
1.7. Quelques remarques sur le modèle. 86

2. Le langage de description. 87

3. Exemple de description. 88

VI. L'ALLOCATEUR DES VARIABLES. 95

1. Structure d'un programme Pascal. 95

2. Gestion de la mémoire pour un programme Pascal. 97
2.1. Gestion de la pile d'exécution. 98
2.1.1. Adressage des informations de la pile. 98
2.1.2. Format d'un environ de procédure. 100
2.2. Gestion du tas. 102

3. Compilation des définitions de types. 104
3.1. Les types simples. 105
3.2. Les types pointeurs. 106
3.3. Les types structurés. 106
3.3.1. Les tableaux. 107
3.3.2. Les enregistrements. 109
3.3.3. Les ensembles 110
3.3.4. Les fichiers. 112

4. Allocation de mémoire. 113
4.1. Classement des variables. 114
4.2. Analyse des intervalles de déplacement. 114
4.3. Assignation des adresses. 117

VII. LE CODEUR. 119

1. Noyau de génération de code. 120
1.1. Evaluation d'une opération élémentaire. 122
1.1.1. Evaluation d'une conformité parfaite. 125
1.1.2. Transformation d'une opération. 127
1.2. Génération effective de code. 128
1.3. Gestion des registres. 130
1.3.1. Structure de données utilisée. 131
1.3.2. Les algorithmes. 133
1.4. Gestion des temporaires et des constantes. 134
1.4.1. Les temporaires. 134
1.4.2. Les constantes non immédiates. 136
1.5. La gestion des descripteurs. 137
1.6. Exemple d'utilisation du noyau. 139

2. Compilation des expressions. 142
2.1. Les expressions d'accès. 142
2.1.1. Accès simple. 143
2.1.2. Indirection à travers un pointeur. 143
2.1.3. Champ d'un enregistrement. 144
2.1.4. Elément d'un tableau. 144
2.2. Les expressions arithmétiques. 146
2.2.1. Traitement d'une sous-expression commune. 147
2.2.2. Les opérateurs arithmétiques. 147
2.2.3. Les opérateurs ensemblistes. 148
2.2.4. Le constructeur d'ensembles. 148
2.3. Les expressions logiques. 150
2.3.1. Contextes d'évaluation. 150
2.3.2. Analyse d'une expression. 151
2.3.3. Traitement des comparaisons. 151
2.3.4. Traitement des opérations logiques. 153

3. Compilation des instructions. 153
3.1. L'affectation. 154
3.1.1. Affectation d'une valeur simple ou d'un ensemble. 154
3.1.2. Affectation d'une structure. 154
3.2. Les instructions de contrôle. 155
3.2.1. Instructions conditionnelles. 155
3.2.2. Instructions repétitives. 157
3.2.3. Autres instructions. 158
3.3. Appel de procédure. 159

VIII. L'OPTIMISEUR FINAL. 161

1. Interface avec le générateur. 161

2. Structures de données internes. 162

3. Phase d'optimisation. 164

4. Phase de résolution des branchements. 165
4.1. Résolution des cas simples. 166
4.2. Résolution des conflits. 168
4.3. Calcul des adresses de branchement. 168

5. Un exemple d'optimisation. 169

IX. RESULTATS ET CONCLUSIONS. 177

1. Réalisation d'un prototype du système GEMME. 177
1.1. Choix du langage de développement. 177
1.2. Etat actuel. 178
1.3. Expérimentation. 180
1.3.1. Qualité du code généré. 180
1.3.2. Production d'un nouveau générateur. 181
1.3.3. Vitesse de compilation. 181

2. Contributions. 182

3. Perspectives. 183

Annexe A : Définition du langage de description (LD).

Annexe B : Description du processeur Motorola 68000.

Annexe C : Description du processeur LSI-11.

Annexe D : Représentation interne GEMME.

Annexe E : Exemples de dérivation de code.

Annexe F : Comparaison du code généré.

Bibliographie.

I. INTRODUCTION.

Au cours des dix dernières années, les recherches sur l'automatisation de la production de compilateurs ont connu un essor important. Deux raisons principales ont suscité cet intérêt :

a) La multiplication des machines disponibles sur le marché, en raison des progrès réalisés dans le domaine de la micro-informatique (technologie VLSI, micro-programmation, etc.).
b) L'apparition en même temps d'un nombre important de langages nouveaux, de plus en plus complexes à compiler.

Des progrès considérables ont été réalisés dans l'automatisation de l'écriture de la partie analyse des compilateurs (<Aho77>). Ainsi, il est très courant de nos jours d'utiliser un compilateur dont l'analyseur syntaxique a été construit à l'aide d'un générateur automatique d'analyseurs (<Johnson75>, <Boullier80>, ...). Cet aspect de la production de compilateurs est donc devenu plus simple mais aussi plus fiable ; il est en effet plus facile de vérifier qu'une grammaire est complète et correcte que de le faire pour un analyseur écrit manuellement.

D'autre part, de nombreux travaux d'automatisation ont été entrepris sur la partie génération de code (<Catell77>, <Ganapathi82b>) ; les résultats obtenus sont toutefois plus modestes, à part quelques exceptions (<Leverett80>, <Graham82>). Ceci s'explique essentiellement par le manque de formalisation de cet aspect ; en effet, les méthodes de génération existantes sont peu nombreuses et basées en général sur une analyse cas par cas des possibilités de la machine cible. Par ailleurs, un bon nombre des méthodes formelles proposées sont d'une application pratique très restreinte soit parce qu'elles sont inefficaces, soit parce qu'elles sont d'implantation difficile.

Cette thèse essaye d'apporter une solution au problème de l'automatisation de la production des générateurs de code. Elle propose une méthode formelle de génération basée sur une séparation nette entre les algorithmes de génération et les données relatives à la machine cible ; cette méthode fait appel à des techniques de l'intelligence artificielle pour réaliser la sélection des instructions machine (reconnaissance de formes, retour arrière, etc.) et permet d'obtenir un code d'une qualité comparable au minimum à celle des compilateurs de production. Elle propose ensuite un système capable de créer, à partir d'une description de la machine cible, un générateur de code basé sur cette méthode. Enfin, elle présente la mise en oeuvre d'un prototype de ce système et une évaluation de ce prototype.

Plan de la thèse.

Ce document est divisé en deux parties. Dans la première partie (chapitres II à IV), nous présentons les différents aspects du problème de la génération de code ainsi que les principes de la solution que nous proposons. Dans la deuxième partie (chapitres V à VIII), nous décrivons les différents composants du système de génération proposé.

Le chapitre II est consacré à une étude de l'état de l'art. Nous présentons d'abord les problèmes posés par la génération de code. Nous analysons ensuite les différents travaux réalisés dans le domaine, en mettant en relief leurs atouts et leurs points faibles ; dans cette analyse, nous nous intéressons particulièrement aux travaux d'automatisation de la production des générateurs. Enfin, nous essayons de dégager les idées les plus importantes contenues dans ces travaux.

Dans le chapitre III, nous présentons nos objectifs et l'approche suivie pour essayer de les atteindre. Nous donnons une vue d'ensemble de notre système de génération, puis nous décrivons de façon détaillée la méthode de génération utilisée.

Le chapitre IV passe en revue les différentes techniques d'optimisation du code généré. Il montre ensuite comment

certaines de ces techniques peuvent être mises en oeuvre de façon indépendante de la machine. Il présente enfin la liste des optimisations appliquées dans notre modèle de génération.

Le chapitre V expose le formalisme utilisé pour décrire une machine. Il montre ensuite comment cette description est utilisée pour construire un nouveau générateur de code.

Le chapitre VI est consacré à l'étude du composant du générateur responsable de la gestion de la mémoire du programme compilé (l'allocateur des variables). Il présente d'abord les choix effectués en ce qui concerne l'organisation de cette mémoire. Puis, il décrit la façon de déterminer une représentation physique pour chaque type défini par l'utilisateur. Enfin, il présente la méthode d'allocation de mémoire que nous utilisons pour les variables du programme.

Le composant essentiel du prototype est décrit au chapitre VII. Nous décrivons en premier lieu le noyau de génération de code chargé de la traduction des opérations élémentaires et de la gestion des ressources de la machine. Nous décrivons ensuite la compilation des expressions et des instructions.

Le chapitre VIII traite de la mise en oeuvre des optimisations finales décrites au chapitre IV. Il montre également comment ces optimisations peuvent permettre de simplifier certaines tâches du générateur de code.

Dans la conclusion (chapitre IX), nous présentons les résultats des expériences réalisées avec le prototype du système ainsi qu'une évaluation de celui-ci par rapport aux objectifs initiaux. Nous réalisons ensuite une critique des choix effectués et nous essayons d'ouvrir quelques perspectives.

Enfin, nous présentons en annexe les descriptions de deux machines (MC 68000 et PDP-11) ainsi qu'une trace de l'exécution du générateur pour un ensemble d'opérations.

PREMIERE PARTIE

II. LA GENERATION DE CODE : ETAT DE L'ART.

1. Présentation du problème.

Génération de code est le terme utilisé pour désigner l'ensemble de phases de synthèse d'un compilateur (<Gries71>). Cette synthèse correspond à la traduction en code objet de la représentation intermédiaire produite par les phases d'analyse du compilateur ; elle regroupe de nombreuses tâches :

a) La sélection d'instructions machine. Dans la plupart des machines, certaines opérations peuvent être réalisées de différentes façons ; il faut, par conséquent, décider quelle séquence d'instructions sera utilisée pour traduire chaque opération, en fonction de son contexte. La difficulté de cette tâche dépend de la richesse du jeu d'instructions de la machine mais aussi du dégré de finesse que l'on veut atteindre.

b) La détermination de l'ordre d'évaluation d'une expression. Cet ordre est important car il peut permettre de minimiser le nombre de registres nécessaires à l'évaluation de l'expression ; la détermination du meilleur ordre est toutefois une tâche difficile, à cause de la diversité des opérandes qui peuvent être impliqués (différences de type, de taille, etc) et de la présence éventuelle de sous-expressions communes..

c) La gestion de registres. L'utilisation efficace des registres de la machine est un facteur prépondérant dans la qualité d'un programme ; le compilateur doit par conséquent essayer d'utiliser au mieux ces registres. Trois aspects doivent alors être abordés :
- l'allocation de registres aux valeurs les plus utilisées (allocation globale),

- l'assignation d'un registre en cours de génération (allocation locale),
- la gestion des informations concernant les registres (registres libres, dernières valeurs qui leur ont été affectées, registres dont la libération peut être forcée, etc).

Cette énumération donne une idée de la difficulté de cette tâche.

d) Le choix d'une représentation pour chaque type du programme. Certains types peuvent être représentés de plusieurs façons dans une même machine ; il faut donc déterminer les représentations les plus appropriées. Cette tâche concerne principalement les types construits par l'utilisateur.

e) L'allocation de mémoire aux variables et constantes du programme ainsi qu'aux valeurs intermédiaires générées par le compilateur. Cette tâche consiste à assigner une adresse à chacune de ces entités suivant les règles de durée de vie et de visibilité du langage source ; elle se charge également de la gestion interne de segments associés aux procédures (environs) et de la gestion des variables créées et détruites dynamiquement (tas).

f) L'accès aux données. La désignation d'une donnée doit être traduite à l'aide des modes d'adressage de la machine cible ; il est donc nécessaire d'utiliser une technique d'adressage adaptée à la structure du langage mais aussi à la machine (registres "display", liens statiques, etc).

g) La gestion des appels de procédure. Profondément liée aux deux tâches précédentes, c'est elle qui se charge de l'allocation et la libération des environs ainsi que de la mise en place des mécanismes nécessaires à leur adressage ; elle se charge également de la sauvegarde et de la restauration du contexte de l'appelant et du passage de paramètres.

h) Les optimisations finales. Après la génération de code,

certaines optimisations sont encore possibles ; deux raisons expliquent ce fait :

- certaines informations ne deviennent disponibles qu'après la génération de la totalité du programme,
- deux séquences d'instructions qui sont localement optimales peuvent ne plus l'être après concaténation.

Parmi ces optimisations, mentionnons la suppression d'instructions redondantes ou inaccessibles, l'utilisation d'instructions et de modes d'adressage plus performants, la rupture des cascades de branchements, etc.

La génération de code constitue par conséquent un problème complexe et très vaste.

2. Solution traditionnelle.

L'approche traditionnelle consiste à développer, pour chaque classe d'opérateur ou d'opérande du langage source, un ensemble de procédures spécialisées dans leur traduction ; toutes les tâches de génération nécessaires à cette traduction sont intégrées dans ces procédures.

L'approche n'est pas très modulaire et donne lieu à des générateurs volumineux, même si certaines tâches sont très simplifiées et parfois absentes ; d'autre part, les procédures sont réalisées de manière ad-hoc et sont par conséquent très dépendantes aussi bien du langage source que de la machine cible.

3. Le problème des NxM compilateurs.

Avec l'approche traditionnelle, tout le travail est à recommencer si l'on change soit de langage, soit de machine ; pourtant, de nombreux langages et machines apparaissent chaque année. Ces constatations ont donné naissance à de nombreuses recherches ; le problème peut être résumé ainsi : comment réaliser les compilateurs de N langages sur M machines

différentes, sans avoir à écrire NxM programmes?

Les premières propositions à ce sujet remontent au projet UNCOL (<Strong58>) dont le but était de définir un langage intermédiaire unique (UNCOL) qui serait produit par les analyseurs des N langages et qui serait accepté en entrée par les générateurs de code des M machines (figure II.1) ; ainsi, seul N+M programmes seraient nécessaires pour résoudre le problème énoncé.

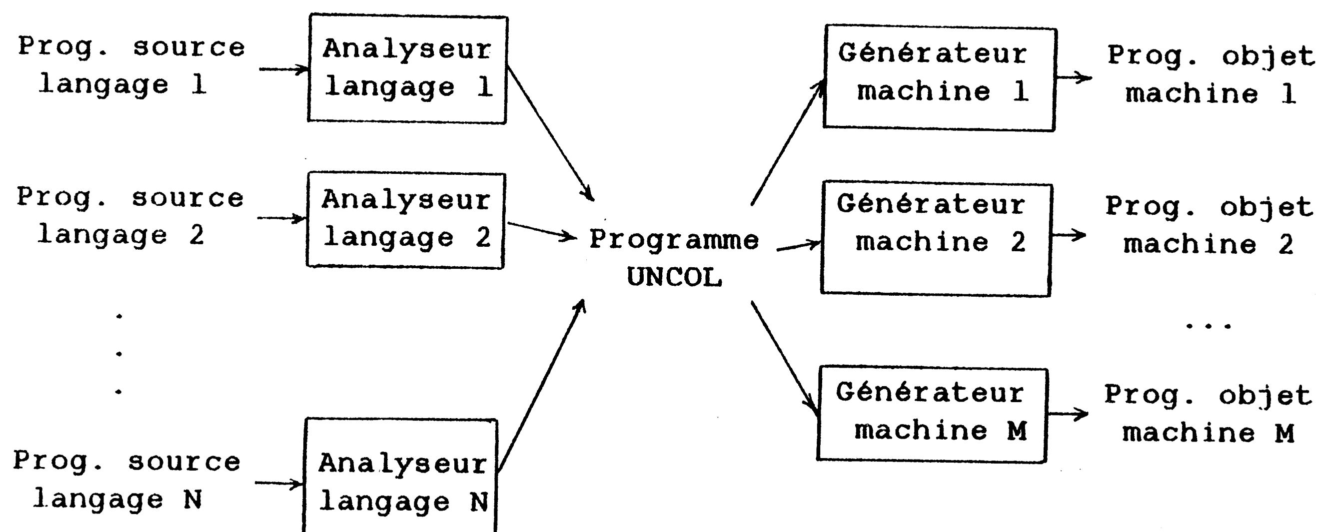


Figure II.1 Modèle de génération de code UNCOL.

Une première version du langage UNCOL fut proposée par <Steel60> mais elle ne fut suivie d'aucune réalisation. D'autres projets, tels que SLANG (<Sibley61>) et JANUS (<Coleman74>, <Haddon78>) ont toutefois repris et appliqué ces mêmes idées ; malheureusement, leurs résultats n'ont jamais été très bons, probablement à cause de leurs objectifs trop ambitieux.

Un deuxième type de projets s'est également inspiré d'UNCOL. L'idée centrale de ces projets est d'utiliser, à l'instar d'UNCOL, un langage intermédiaire proche des machines réelles et facile à produire par un analyseur ; par contre, ils ne se sont intéressés qu'à la traduction d'un seul langage source. Ces projets ont eu beaucoup plus de succès que les précédents ; en effet, leurs langages intermédiaires ont été utilisés dans de nombreux compilateurs ; c'est le cas, par exemple, du P-code

pour Pascal (‹Ammann73›, ‹Nori75›) et du Ocode pour BCPL (‹Richards71›).

Cette dernière approche permet de réduire fortement l'effort pour résoudre le problème des NxM compilateurs ; en effet, la partie analyse de chaque langage ne nécessite d'être écrite qu'une seule fois. Toutefois, l'effort reste encore très important car la partie génération de code doit être reécrite pour chaque machine.

Un deuxième problème est alors posé : comment réduire l'effort de production de ces générateurs de code?

4. Formalisation de la génération de code.

Une solution consiste à formaliser la génération de façon à rendre plus systématique l'écriture de ces générateurs ; plusieurs modèles de génération ont ainsi été proposés.

Certains travaux se sont intéressés exclusivement au problème de génération optimale de code, à partir d'un arbre qui représente une expression (‹Nakata67›, ‹Sethi70›, ‹Bruno75›, ‹Aho76›). Ces recherches ont été réalisées pour des modèles de machines abstraites très regulières et sans se soucier des coûts de mise en oeuvre ; leur intérêt pratique n'est par conséquent pas immédiat, malgré les résultats significatifs obtenus. Toutefois, elles ont permis de dégager un certain nombre de principes et d'orienter de nouvelles recherches.

Parmi ces travaux, celui de Aho et Johnson (‹Aho76›) est, à notre avis, le plus important. Ils proposent un algorithme de génération optimale pour des machines à n registres généraux, valable sous certaines hypothèses simplificatrices ; cet algorithme effectue 3 passes sur l'arbre de l'expression :

1) La première calcule, pour chaque noeud de l'arbre, le nombre minimum d'instructions que prendra sa traduction, si l'on utilise i registres ($0 \le i \le n$) ; toutes les permutations d'ordre d'évaluation sont prises en compte. Cette passe

permet de déterminer la séquence qui traduit chaque noeud de façon optimale ainsi que le meilleur ordre d'évaluation pour ses fils.

2) La seconde passe marque les noeuds de l'arbre qui devront être sauvegardés dans des temporaires, à cause du nombre limité des registres.

3) La dernière passe génère le code, suivant les choix des passes précédentes.

Etant donné son caractère exhaustif, l'algorithme est relativement simple ; de plus, son coût est seulement linéaire par rapport aux tailles de l'arbre et du jeu d'instructions de la machine cible, ce qui peut paraître étonnant. Malheureusement, les hypothèses prises sont trop restrictives ; ainsi, par exemple, ils supposent que tous les registres sont symétriques pour toutes les opérations, une propriété inexistante dans les machines réelles.

Une méthode de génération similaire est proposée dans ‹Newcomer75›. Cette méthode consiste à chercher, parmi un ensemble de modèles d'expression, celui qui est conforme à l'expression qu'il faut traduire ; les modèles sont des arbres, décorés avec les attributs à satisfaire pour obtenir la conformité et avec la séquence de code à générer si le modèle correspondant est choisi ; ils sont regroupés par opérateur, pour des questions d'efficacité. Si la recherche échoue, un ensemble de transformations est appliqué à l'expression, suivant ses différences par rapport aux attributs des modèles examinés ; ces transformations consistent soit à transférer les opérandes vers d'autres ressources, soit à modifier leur ordre en fonction des propriétés arithmétiques de l'expression. Une présentation concise mais très claire de l'algorithme est faite dans ‹Cattell77›.

Le méthode de Newcomer pourrait être d'une application plus directe que celle de Aho et Johnson mais son coût est trop élevé ; en fait, l'algorithme analyse toutes les possibilités de traduction de l'expression, ce qui prend parfois des semaines de temps de calcul, même pour des expressions simples (‹Ganapathi81›). Toutefois, les idées proposées ont été d'une

grande valeur et se retrouvent sous différentes formes dans des projets tels que PQCC (‹Wulf79›, ‹Leverett80›) ou le notre.

D'autres travaux se sont intéressés un peu plus à l'aspect mise en oeuvre du modèle de génération proposé ; parmi eux, Glanville et Graham (‹Glanville78›) dont le modèle est analysé plus loin (cf II.5.2). Akin et Leblanc (‹Akin82›), pour leur part, ont proposé une méthode particulièrement pratique qui est composée de 4 étapes :

1) Définition d'un langage intermédiaire proche du langage source.
2) Identification des séquences d'instructions machine pouvant être utilisées pour traduire les opérateurs du langage intermédiaire.
3) Développement des algorithmes de sélection, basés sur une analyse des possibilités de traduction de chaque opérateur en fonction de son contexte, de son type de données et de l'accessibilité de ses opérandes. Un schéma de ces algorithmes est proposé avec la méthode.
4) Développement des algorithmes d'optimisation du code généré, suivant les caractéristiques de la machine.

Cette méthode a été utilisée avec succès pour la construction de 2 compilateurs (Pascal et C).

Toutes ces méthodes permettent de simplifier l'écriture des générateurs de code mais, encore une fois, l'effort de réalisation demeure non négligeable.

5. Automatisation de la production de générateurs

Une autre solution consiste à développer des outils spécialisés, permettant d'automatiser (totalement ou en partie) la construction des générateurs de code.

Dans cette solution, certaines tâches de la génération sont directement mises en oeuvre par les outils ; il s'agit des aspects de la méthode de génération employée dont le traitement est identique quelque soit la machine cible. En ce qui concerne

les autres tâches, les outils assurent également leur mise en oeuvre mais nécessitent, pour ce faire, un certain nombre d'informations qui doivent leur être communiquées par l'écrivain du compilateur ; la nature et l'importance de ces informations peuvent différer énormément d'un outil à l'autre ; elles sont en général spécifiées dans un langage ad-hoc, défini avec les outils.

De nombreux travaux ont orienté leurs recherches dans cette direction ; toutefois, les outils proposés ainsi que les modèles de génération utilisés sont très variés. Il est donc très difficile de résumer l'ensemble en quelques lignes, ce qui nous a conduit à faire une présentation sommaire de chaque travail, tout en mettant en relief leurs atouts et leurs faiblesses ; par ailleurs, nous effectuons des comparaisons là où elles nous ont semblées appropriées.

Nous distinguons 2 approches dans notre présentation, suivant la nature des informations nécessaires pour la création d'un nouveau générateur : l'approche procédurale et l'approche descriptive.

5.1. L'approche procédurale.

Dans cette approche, le réalisateur doit décrire la traduction ("mapping") en langage machine de chaque opérateur du langage intermédiaire. Il s'agit par conséquent d'une spécification du processus de génération lui-même.

L'organisation de la description est de type procédural, ce qui explique le nom de l'approche. En effet, à tout opérateur doit être associée une unité du langage de description (une procédure, une macro, etc.) chargée de décrire sa traduction, en fonction des différents types de données et d'opérandes que cet opérateur peut manipuler.

La génération de code est réalisée par un interpréteur du langage de description ; elle consiste à parcourir la représentation intermédiaire du programme et à activer les

procédures associées aux opérandes rencontrés.

Plusieurs techniques ont été utilisées pour mettre en oeuvre cette approche ; nous en analysons ci-après quelques unes.

5.1.1. La macro-génération.

C'est la technique la plus simple ; elle consiste à définir des macro-instructions qui réalisent l'expansion des opérateurs en instructions machine ; le macro-processeur correspond donc à l'interpréteur du langage de description. Cette technique a été utilisée entre autres pour la mise en oeuvre d'un compilateur PL1 à Grenoble (‹Griffiths69›), de JANUS (‹Coleman74›) et du compilateur CPL1 de CAP-SOGETI (‹Hochain80›).

L'analyse du contexte d'un opérateur est en général très restreinte, ce qui conduit souvent à une génération de code maladroite. D'autre part, le langage intermédiaire utilisé est nécessairement de bas niveau et en général très proche d'une famille donnée de machines ; de ce fait, l'écriture des macros pour une machine qui n'appartient pas à cette famille, est une tâche malaisée.

Une variante plus intéressante de cette technique est proposée dans ‹Miller71›. Dans cette variante, un premier jeu de macros est utilisé pour traduire le langage intermédiaire en un langage d'un niveau plus proche de la machine ; ces macros constituent donc une interface indépendante de la machine qui peut permettre une modification, et éventuellement un remplacement, du langage intermédiaire sans trop de dégats ; en outre, le langage utilisé pour leur écriture (MIML : Machine Independent Macro Language) permet une analyse plus fine du contexte de chaque opérateur et, par conséquent, de meilleurs résultats. La traduction en langage machine est ensuite réalisée par un deuxième jeu de macros ; ces macros sont écrites dans un autre langage (OMML : Object Machine Macro Language) et constituent en pratique une description de la machine cible ; le transport d'un générateur vers une autre machine ne nécessite par conséquent que la réécriture de ces macros.

Cette méthode atténue l'étendue des limitations soulignées pour la macro-génération simple. Par ailleurs, elle introduit la possibilité de générer automatiquement, en cas d'échec d'une macro, des transferts mémoire-registre (ou vice versa) de façon à forcer son application ; comme une bonne partie des macros est en général consacrée à décrire ces transferts, cette possibilité allège fortement l'écriture des macros. Enfin, cette méthode a probablement été la première à essayer de séparer les algorithmes de génération des informations concernant la machine cible et constitue, par conséquent, un pionnier de l'approche descriptive.

Le modèle de Miller est toutefois trop limité en ce qui concerne les modes d'adressage ; en effet, il ne permet pas l'utilisation de modes d'adressage très répandus, tels que l'indirection ou l'indexation.

Certaines autres limitations sont communes aux 2 méthodes de macro-génération :

a) L'écriture des macros est une tâche complexe mais surtout fastidieuse, à cause du grand nombre de cas d'analyse à prévoir.
b) La qualité du code généré est déterminée par l'habileté de l'écrivain des macros.

5.1.2. L'interprétation pure.

Dans cette technique, la description du processus de génération est traduite en une forme interne plus compacte, puis interprétée lors de la compilation d'un programme. Le langage de description est en réalité un langage de programmation de haut niveau, spécialisé dans la génération de code ; une procédure de ce langage correspond à la spécification de la traduction d'un opérateur ou d'un type d'opérandes.

Elson et Rake (‹Elson70›) ont utilisé cette technique pour réaliser un compilateur PL/1. Les procédures, qu'ils appellent OMD (OPGEN Macro Definitions), sont écrites dans le langage GCL

(Generate Coding Language) et exécutées par l'interpréteur OPGEN ; la génération est effectuée en un seul parcours de l'arbre abstrait du programme, piloté par le type du noeud visité. Leur réalisation a démontré la faisabilité d'une telle méthode, bien qu'une seule machine cible ait été utilisée (IBM/360) ; quant aux performances, le temps de compilation est assez important mais les auteurs pensent qu'une optimisation de l'interpréteur peut le réduire fortement ; sinon, il existe toujours la possibilité de compiler les OMD.

Cette technique a également été utilisée par le projet ECS (Experimental Compiling System) d'IBM (‹Carter77›, ‹Allen80›). La particularité d'ECS est que le langage de description (IL : Intermediate Language) est aussi le langage intermédiaire du compilateur ; ainsi, la définition d'une nouvelle procédure IL permet d'enrichir ce langage, tout en indiquant la traduction qui doit en être faite. La génération de code se fait par affinements successifs du programme intermédiaire, jusqu'à ce que la traduction soit complète ; à chaque affinement, des opérations de plus bas niveau sont générées, puis soumises à un certain nombre d'optimisations. Les résultats obtenus ont été très significatis, surtout en ce qui concerne la qualité du code généré ; par contre, le temps de compilation est ici aussi assez important (toujours pour une version interprétée).

Contrairement à Elson et Rake, la représentation intermédiaire utilisée par ECS n'est pas arborescente mais linéaire ; il s'agit d'un choix très discutable car l'analyse du contexte et, à un moindre niveau, la manipulation du programme ne peuvent être qu'alourdies lors des optimisations.

Une technique à cheval entre la macro-génération et l'interprétation pure est celle proposée par Young (‹Young74›). Dans cette technique, le langage de description (ICL : Interpretive Coding Language) ressemble beaucoup à un langage de macro-génération ; cependant, ses constructions sont plus souples et, en outre, il prend en charge des tâches telles que l'accès aux données et la gestion de registres. Bien que cette technique n'ait jamais été implémentée, on peut y trouver de

nombreuses limitations ; l'une d'elles est, encore une fois, le choix d'une représentation intermédiaire linéaire, d'autant plus qu'une représentation arborescente est utilisée par les phases précédentes ; une autre limitation concerne le choix qui a été fait de réaliser toute la traduction en une seule passe, ce qui oblige à faire des acrobaties pour traiter les reférences en avant.

L'interprétation pure présente certains avantages par rapport à la macro-génération. Ainsi, elle permet une meilleure analyse du contexte de l'opérateur à traduire, d'où un meilleur résultat au niveau de la qualité du code généré ; d'autre part, l'utilisation d'un langage de haut niveau pour l'écriture de la description ne peut que faciliter cette tâche. Toutefois, elle conserve les inconvénients principaux de la macro-génération :

a) La réalisation de la description est toujours une tâche longue et délicate.
b) La qualité du code généré dépend également de l'habileté de l'écrivain de la description.
c) Les algorithmes de génération sont toujours mélangés avec les informations relatives à la machine cible.

5.1.3. Les automates d'états finis.

Cette technique utilise un modèle de génération de code basé sur la notion d'automates d'états finis. Dans ce modèle, la traduction d'un opérateur correspond à une suite de transitions entre les états de l'automate (la machine cible), en fonction des propriétés des opérandes et de l'état des ressources de la machine cible ; l'émission de code et les autres actions de génération (allocations, libérations, etc.) se produisent comme un effet secondaire de ces transitions. Une transition est en fait un transfert de données entre deux types de ressources (par exemple, mémoire-registre).

La description de la traduction est faite dans un langage spécialisé, permettant de spécifier les états et les transitions de l'automate ; cette description met en oeuvre l'automate et, par conséquent, le générateur de code. La tâche principale du

traducteur est de s'assurer que toutes les opérations peuvent être traduites et de générer ensuite les chemins les plus courts pour effectuer ces traductions ; pour ce faire, il construit d'abord le graphe complet de l'automate, en appliquant toutes les transitions aux états décrits et en créant de nouveaux états lorsque cela est nécessaire ; ensuite, il analyse ce graphe.

La technique des automates a été employée par M.K. Donegan, à deux reprises (‹Donegan73›, ‹Donegan79›). C'est le langage de description qui diffère entre ses propositions ; en effet, dans la première, le langage est de type procédural (CGPL : Code Generator Preprocessor Language) tandis que dans la seconde, il est plutôt descriptif (CGGL : Code Generator Generator Language). En fait, ce dernier est une version améliorée du premier dans laquelle beaucoup d'informations sont automatiquement déduites par son traducteur ; celui-ci est donc devenu plus "intelligent".

Nous retiendrons deux avantages de cette technique. Le premier concerne la séparation nette qui est faite entre l'étape de construction d'un générateur et l'étape de génération de code ; c'est, à notre connaissance, la première technique qui l'a utilisée. Le second est la simplicité du modèle utilisé, ce qui permet, d'une part, une compréhension et une mise au point plus rapide du générateur et, d'autre part, d'avoir une description beaucoup plus simple. Toutefois, ce modèle présente également de nombreux points faibles ; en particulier :

a) Il ne permet pas de modéliser l'ensemble du processus de génération ; de ce fait, l'écrivain du générateur doit écrire dans un langage de haut niveau certaines tâches de la génération (entre autres, la gestion de registres et l'émission effective du code).
b) La présence d'un nombre important de registres peut provoquer une explosion du nombre d'états de l'automate. Aucune solution valable n'a pu être trouvée pour ce problème.
c) Le traitement des sous-expressions communes n'est pas prévu dans le modèle ; il ne s'agit toutefois que d'une limitation mineure.

Ce modèle doit être enrichi, à notre avis, pour constituer une

solution complète à la génération ; par contre, il semble s'adapter assez bien au traitement de l'accès aux données. Nous verrons plus loin que certaines techniques de l'approche descriptive ont de nombreux traits communs avec cette technique.

5.1.4. L'isolation des fonctions dépendantes de la machine.

Cette dernière technique s'éloigne considérablement des principes énoncés pour l'approche procédurale. En effet, son objectif principal est de définir une structure de compilateur telle que la plupart des fonctions soient indépendantes de la machine cible et que celles qui ne le sont pas soient organisées de façon à faciliter leur réécriture. L'automatisation de la production du compilateur est par conséquent abandonnée au profit d'une réduction de l'effort de réécriture.

Cette isolation des fonctions dépendantes de la machine cible a été pratiquée dans la réalisation du compilateur portable Zed (‹Bonkowski79›) ; grâce à elle, le compilateur a pu être porté avec succès sur plus d'une douzaine de machines. Le transport sur une autre machine prend, d'après les auteurs, entre un et six mois ; ce temps se compare favorablement avec les techniques analysées précédemment, d'autant plus qu'il comprend l'analyse et la modélisation de la machine cible. Toutefois, les inconvénients de ces autres techniques sont également présents dans cette technique ; en particulier, la réécriture des fonctions dépendantes de la machine est toujours une tâche difficile car il faut prévoir tous les cas de sélection d'instructions ; elle est également délicate car la qualité du code généré en dépend directement.

5.2. <u>L'approche descriptive.</u>

La principale caractéristique de cette approche est la séparation qu'elle fait entre les algorithmes de génération et les informations relatives à la machine cible. Cette séparation permet de développer des algorithmes invariables par rapport à la machine et de les paramétrer par des tables décrivant la machine cible ; la construction d'un nouveau générateur de code

se réduit, par conséquent, à un assemblage de ces algorithmes avec les tables correspondant à la machine visée.

La création de ces tables est en général réalisée automatiquement, à partir d'une description de la machine. Cette description correspond à une spécification de la structure et des instructions de la machine ; c'est le constructeur des tables qui se charge d'en extraire et parfois d'en déduire les informations nécessaires à la génération. Toutefois, dans certains générateurs, la construction des tables est faite manuellement.

Cette approche est fortement inspirée des techniques utilisées pour la construction automatique d'analyseurs syntaxiques et, comme elles, a eu beaucoup de succès ; en effet, elle est utilisée par la plupart des recherches récentes en génération de code, avec d'excellents résultats.

L'approche descriptive a permis, en particulier, de combler les deux limitations principales de l'approche procédurale :

a) L'analyse du contexte d'une opération n'a plus besoin d'être spécifiée pour construire un nouveau générateur ; cette analyse est intégrée une fois pour toutes aux algorithmes de génération. De ce fait, la construction d'un générateur devient plus simple.
b) La qualité du code généré dépend des algorithmes de génération utilisés et non de l'habileté du réalisateur ; toutefois, celui-ci peut, dans certains cas, influencer sur le choix de certaines instructions.

Dans ce qui suit, nous présentons les différentes techniques utilisées pour mettre en oeuvre cette approche.

5.2.1. Simulation.

Dans cette technique, l'algorithme de génération est fondé sur une simulation de la machine abstraite définie par le langage intermédiaire. En effet, la simulation des opérations à traduire est utilisée pour déterminer les effets qu'elles

produisent sur l'état de cette machine ; ces informations permettent ensuite de générer les instructions qui produisent les effets équivalents sur la machine cible.

Toutes les informations concernant la machine cible sont réunies dans un ensemble de tables ; celles-ci définissent les différentes possibilités de traduction pour chaque opération du langage intermédiaire.

Le traitement d'une opération consiste à chercher, parmi ses possibilités de traduction, celle qui correspond exactement à l'état de la machine abstraite ; en cas d'échec, on modifie cet état en effectuant des transformations ("coercions") sur les ressources de la machine, de façon à le rendre conforme avec une de ces possibilités ; ces transformations correspondent en général à l'allocation d'une ressource et au transfert de l'un des opérandes vers celle-ci. La possibilité choisie spécifie le code qu'il faut générer, si cela est nécessaire, ainsi que les conséquences de ce choix sur l'état de la machine cible. Enfin, l'état de la machine abstraite est mis à jour en fonction de l'opération traduite ainsi que des instructions générées. Dans certains générateurs, l'analyse d'une opération prend en compte les opérations qui la suivent ; le nombre de ces opérations est alors fixé lors de la construction du générateur.

Cette technique a été utilisée, entre autres, pour la réalisation du générateur de code UGEN (‹Perkins79›, ‹Ganapathi82›) et du système ACK (Amsterdam Compiler Kit : ‹Tanenbaum83›).

La simulation constitue le successeur des techniques de macro-génération et d'interprétation pure ; en effet, elle se base sur l'interprétation du langage intermédiaire pour réaliser la génération. Par contre, elle isole tout ce qui concerne la machine cible dans des tables, ce qui permet une construction plus rapide des générateurs ; ainsi, par exemple, la réalisation d'un nouveau générateur UGEN prend, d'après les auteurs, environ six semaines. D'autre part, la possibilité d'analyser le contexte de l'opération (l'état de la machine abstraite) lui

permet de générer un code de meilleure qualité ; en effet, les résultats obtenus aussi bien par UGEN que par ACK sont très proches de ceux de compilateurs classiques (Pascal Stanford/Hambourg du Dec-20, C UNIX du Motorola 68000 (‹Motorola80›), etc.).

Cette technique a cependant quelques faiblesses. En premier lieu, certains aspects de l'algorithme de génération sont spécifiés avec la description de la machine cible ; c'est par exemple le cas des conséquences du choix d'une possibilité de traduction sur les états de la machine abstraite et de la machine cible ; c'est également le cas du passage de paramètres et de l'appel de procédure. En second lieu, la description de certaines opérations peut être très malaisée ; les opérations ensemblistes de Pascal en constituent un bon exemple.

Dans toutes les techniques analysées précédemment, la sélection d'instructions machine est réalisée par interprétation de l'opérateur à traduire ; ceci conduit en général à un grand nombre de cas d'analyse et, par conséquent, à des algorithmes volumineux et compliqués. Une autre possibilité est l'utilisation de techniques de reconnaissance de formes ; en effet, une opération peut être vue comme un ensemble de modèles ou de formes qui décrivent les différentes combinaisons d'opérandes qu'elle admet ; dans ce cas, l'algorithme de sélection consisterait à chercher le ou les modèles applicables (conformes) à l'opération que l'on veut traduire. Cette approche conduit à des algorithmes certainement plus simples et généraux ; de ce fait, elle a été incorporée dans de nombreuses autres techniques que nous décrivons ci-après.

5.2.2. Grammaires à attributs.

Une première possibilité, pour mettre en oeuvre la sélection par reconnaissance de formes, est d'utiliser des techniques semblables à celles de l'analyse syntaxique. Dans cette possibilité, les instructions de la machine sont décrites comme des productions d'une grammaire à attributs ; la partie droite d'une production décrit l'expression calculée par l'instruction

et la partie gauche indique où est obtenu le résultat. Par ailleurs, un ensemble d'attributs est associé à chaque production ; certains correspondent aux qualifications sémantiques qu'il faut respecter pour appliquer la production tandis que les autres correspondent à la description du code à générer, si l'application est effectuée.

Les symboles terminaux d'une telle grammaire sont les opérateurs du langage intermédiaire que l'on traduit ; les programmes de ce langage sont par conséquent des chaînes de la grammaire. La "reconnaissance" et la traduction de ces chaînes peuvent être réalisées en utilisant une des techniques classiques d'analyse syntaxique (LR(k), SLR, etc) ; toutefois, certaines différences sont à prendre en compte :

a) La grammaire est en général ambigue car la machine cible peut calculer une même expression de plusieurs façons. La technique de résolution des ambiguités est un facteur important pour la qualité du code généré car elle fixe l'ordre d'application des règles ambigues et privilégie par conséquent le choix d'une règle sur une autre.
b) Les réductions sont plus compliquées car le générateur doit choisir entre plusieurs productions en fonction des informations syntaxiques mais aussi des contraintes sémantiques de chaque production.
c) Le programme analysé est toujours valide (en principe) ; en conséquence, toute erreur détectée par l'analyseur implique soit une erreur dans la grammaire, soit dans l'analyseur.

Un générateur de code est par conséquent une sorte d'analyseur syntaxique, piloté par des tables qui représentent les règles de la grammaire. Ces tables sont construites automatiquement à partir d'une description de la grammaire ; la technique de construction est elle aussi similaire à celle des générateurs d'analyseurs syntaxiques.

Cette méthode recupère donc une grand partie du "savoir faire" de la partie analyse, en l'adaptant aux problèmes particuliers de la génération de code. Elle récupère en même temps de nombreux avantages de ces techniques : modularité,

facilité d'implémentation et surtout la possibilité de prouver que les générateurs sont corrects. D'autre part, les générateurs basés sur cette méthode sont très efficaces ; ils sont certainement les plus rapides parmi les générateurs construits automatiquement, pour une qualité de code généré comparable.

La méthode de grammaires à attributs a été introduite par Glanville et Graham (‹Glanville78›, ‹Graham80›). Leur algorithme de génération est basé sur la technique LR(1) ; elle a été modifiée de façon à accepter des règles ayant une partie gauche vide (pour les instructions sans résultat direct) et à effectuer l'émission de code lors des actions de réduction. Si plusieurs règles peuvent être utilisées pour une même réduction, c'est la première applicable qui est retenue ; ce choix est en principe le meilleur car le constructeur des tables ordonne les règles ambigues suivant un critère de coût défini avec la grammaire. L'allocation de registres est également effectuée par l'algorithme de génération ; chaque classe de registres est représentée par un symbole non terminal de la grammaire ; la présence d'un tel symbole, en partie gauche d'une règle en instance de réduction, provoque l'allocation d'un registre de la classe appropriée ; toutefois, si le symbole est lié à un opérande de la partie droite, c'est celui-ci qui sera utilisé.

Le constructeur de tables vérifie qu'il n'y a pas de cycles dans la grammaire ; il vérifie également qu'une production existe pour tous les symboles non terminaux. Par ailleurs, il rajoute des productions sans contraintes sémantiques pour toute production en possédant au moins une ; pour ce faire, il associe des instructions qui lèvent les contraintes de chaque production ; il évite ainsi les blocages possibles de l'analyseur.

La méthode de Glanville présente un certain nombre de limitations :

a) Il s'agit d'une méthode partielle car elle ne s'intéresse qu'à la sélection d'instructions et à l'allocation de registres. Toutes les autres tâches de la génération doivent être réalisées par ailleurs ; en particulier, l'allocation de

mémoire et l'expansion des identificateurs en adresses mémoire doivent être effectuées avant la génération ; étant donné qu'il s'agit des aspects dépendants de la machine, la portabilité des générateurs s'en voit limitée.

b) Chaque production décrit l'opération effectuée par une instruction pour une combinaison unique de modes d'adressage ; la grammaire peut par conséquent être assez importante. Ainsi, par exemple, une grammaire pour le VAX (‹Dec77›) nécessite environ 8 millions de productions (‹Graham77›).

c) Le contexte utilisé pour traiter une opération est limité, ce qui peut donner dans certains cas du code de pauvre qualité.

d) Les productions rajoutées pour lever les contraintes sémantiques conduisent parfois à du code incorrect ou de mauvaise qualité ; il est alors nécessaire de rédéfinir la grammaire.

Plusieurs implémentations de cette méthode ont été réalisées ; elles comportent pour la plupart des améliorations visant à pallier les limitations énumérées. Les premières réalisations (‹Glanville78›, ‹Henry81›) sont très fidèles à la méthode proposée et sont surtout destinées à montrer la validité de l'approche. Le travail de Henry a été partiellement repris pour réaliser un générateur de code pour le VAX (‹Graham82›), avec des résultats beaucoup plus intéressants ; en particulier, dans le but de réduire le nombre de règles, une factorisation est introduite. Dans une autre réalisation (‹Crawford82›), la méthode est intégrée à d'autres techniques pour produire un compilateur Pascal pour le 8086 (‹Intel78›) ; ce compilateur intègre toutes les tâches de la génération de façon harmonieuse et résoud donc la première des limitations soulignées. Une autre amélioration a été apportée dans le cadre de la réalisation d'un compilateur Pascal pour un Amdahl 470 (‹Bird82›) : il s'agit de la possibilité de rajouter un type aux opérandes d'une production. Enfin, la méthode a également été utilisée à l'Université de Karlsruhe (‹Landwehr82›) pour la réalisation de générateurs de code pour le Siemens 7000 et le Motorola 68000.

La méthode de Glanville a donné, dans tous les cas, des

résultats très significatifs ; elle constitue un véritable pas en avant dans le domaine de la génération de code. C'est pour cette raison que nous nous sommes étendus un peu plus dans sa présentation.

Une méthode similaire a été proposée par Ganapathi et Fischer (<Ganapathi82a>). Il s'agit en réalité d'une extension à la méthode de Glanville dans laquelle on peut associer des attributs plus riches aux productions de la grammaire ; l'objectif de ces attributs est de permettre à l'écrivain du compilateur de :

- résoudre lui-même les ambiguités de la grammaire,
- contrôler la gestion de registres,
- typer les opérandes,
- spécifier les optimisations qu'il désire,
- et, bien sûr, de qualifier les règles avec des contraintes.

La machine cible est décrite par une grammaire à attributs au lieu d'une grammaire hors contexte simple. L'algorithme de génération utilise la technique standard d'analyse LR(k), étendue de façon à traiter les attributs ; la génération est, par conséquent, réalisée en une seule passe. Contrairement à Glanville, l'allocation de mémoire est directement effectuée par l'algorithme de génération, ce qui permet d'obtenir un modèle de génération plus complet.

Des générateurs de code, utilisant la technique de Ganapathi et Fischer, ont été réalisés pour 3 machines : VAX, PDP-11 et Intel 8086 ; la qualité du code généré est comparable à celle des compilateurs de production de ces machines ; par contre, le temps d'exécution est sensiblement supérieur. Le nombre de productions nécessaire à la description d'une machine est très inférieur à celui demandé par la méthode de Glanville ; cette diminution a été obtenue en séparant la description des modes d'adressage de celle des instructions. Par ailleurs, un certain nombre d'optimisations ont été mises en oeuvre à l'aide d'attributs ; les résultats sont satisfaisants mais, à notre avis, la description des optimisations devient rapidement compliquée et peut conduire à des erreurs de génération.

Ces réalisations ont mis en évidence un inconvénient très important de la méthode ; l'utilisation d'attributs, bien qu'elle permet une plus grand souplesse et simplicité au niveau des algorithmes de génération, complique considérablement l'écriture de la grammaire ; de plus, certains aspects des algorithmes de génération se retrouvent souvent dans la description de la machine, ce qui est contraire aux principes de l'approche descriptive.

5.2.3. Reconnaissance de modèles arborescents.

Dans la technique précédente, les modèles utilisés pour la reconnaissance de formes sont linéaires et structurés comme les règles d'une grammaire. Une autre possibilité consiste à organiser ces modèles de façon arborescente et à utiliser pour leur reconnaissance des techniques de l'intelligence artificielle. Cette idée, qui est à la base de nombreux projets, a été mise en oeuvre de façons très différentes que nous présentons ci-après.

Arbre de description unique

Weingart est le premier à avoir utilisé la technique de reconnaissance de modèles arborescents. Dans sa méthode (‹Weingart73›), la machine cible est décrite par un arbre unique ; chaque opérateur du langage intermédiaire correspond à une instruction de la machine, représentée par un sous-arbre spécifiant sa sémantique ; des arbres de conversion sont prévus pour les opérateurs n'ayant aucune instruction machine équivalente. Le générateur de code fonctionne comme une coroutine de la partie analyse ; il reçoit les informations qui composent l'arbre abstrait et les empile jusqu'à ce qu'une conformité soit établie entre ces informations et un sous-arbre du modèle ; un parcours du modèle est réalisé à fur et à mesure que les éléments arrivent pour rechercher cette conformité.

L'implémentation de la méthode de Weingart a mis en évidence trois inconvénients majeurs. En premier lieu, la création

manuelle du modèle est une tâche très lourde et très délicate ; Weingart a donc essayé de l'automatiser mais avec des résultats peu significatifs. En second lieu, les "conversions" font l'objet d'un traitement ad-hoc ; or, il s'agit d'un aspect très important. Enfin, bien que Weingart ne fournisse aucune évaluation, il nous semble qu'il est difficile de générer du bon code avec une telle méthode ; en effet, la qualité du code dépend de l'ordre des sous-arbres du modèle alors que les relations entre ces sous-arbres rendent pratiquement impossible la détermination du meilleur ordre.

Base de connaissances

Une autre approche consiste à utiliser une base de connaissances chargée de piloter la génération (‹Fraser77›) ; cette base comprend un ensemble de règles de traduction et des modèles de description des instructions de la machine. La sélection d'instructions consiste à chercher un modèle conforme à l'arbre qu'il faut traduire ; en cas d'échec, un ensemble de règles de transformation est appliqué suivant le type de l'arbre et les raisons de l'échec ; ces règles ont la forme : remplacer l'indirection par l'indexation, charger l'opérande dans un registre, appliquer la commutativité, etc. D'autres règles sont utilisées pour piloter l'allocation de mémoire et convertir les identificateurs en adresses mémoire.

Les règles sont suffisamment générales pour être applicables à plusieurs machines ; de ce fait, les particularités de chaque machine sont sous-exploitées, ce qui se répercute sur la qualité du code généré. D'autre part, les règles ne peuvent pas être suffisamment générales pour couvrir l'ensemble des machines ; elles sont le plus souvent liées à une classe donnée de machines ; de ce fait, un bon nombre de ces règles sont à réécrire pour construire un nouveau générateur. Enfin, l'application des règles, bien que très élégante, conduit à de mauvaises performances pour une qualité de code médiocre.

Recherche heuristique

L'approche suivie pour la réalisation du compilateur portable C (<Johnson78>) est plus intéressante. Le générateur de code considère l'expression à traduire comme un modèle du calcul qu'il faut réaliser ; par ailleurs, il considère la génération comme une transformation de cette expression. Au centre du générateur, il y a une table contenant les modèles qu'il sait traiter ainsi que leurs règles de transformation, leurs besoins en ressources et les instructions à générer. La génération consiste à choisir successivement les modèles applicables à l'expression traitée et à effectuer les transformations correspondantes jusqu'à réduction complète de l'expression ; l'émission d'instructions est réalisée par effet de bord de cette réduction. Le choix des modèles est effectué par une recherche heuristique pour des raisons d'efficacité ; si la recherche échoue, l'expression est transformée en utilisant un ensemble de règles par défaut. Une stratégie de planification est utilisée pour éviter le blocage du générateur par manque de registres ; un poids, appelé numéro de Sethi-Ullman (<Sethi70>), est affecté à chaque noeud de l'expression pour déterminer quels résultats intermédiaires seront sauvegardés dans des temporaires ; le générateur émettra ensuite les instructions de sauvegarde, avant de traiter les expressions utilisant ces résultats.

Les modèles nécessaires à la génération sont écrits directement par le réalisateur. De ce fait, le transport et la mise au point du générateur demandent un travail non négligeable, estimé à "quelques mois" par les auteurs ; toutefois, le compilateur a été porté à maintes reprises car il fait partie du système UNIX. D'autre part, il n'est pas possible de savoir si la table des modèles est complète ; par conséquent, il existe toujours un risque de bloquer le générateur en essayant de traduire certaines expressions. Par ailleurs, la qualité du code généré dépend des choix de réalisation des modèles mais est en général très satisfaisante.

Le projet PQCC de Carnegie-Mellon (<Wulf79>, <Leverett80>)

suit une approche similaire mais avec des objectifs plus ambitieux. Ce projet a pour but de développer un système capable de générer un compilateur complet, à partir des descriptions du langage source et de la machine cible ; en outre, les compilateurs générés doivent être performants et générer du code très optimisé ; enfin, la réalisation d'un nouveau compilateur doit demander environ trois mois à une personne étrangère au projet. Dans la pratique, PQCC s'est uniquement intéressé à la partie génération de code (‹Cattell78›) ; d'autre part, tout ce qui concerne la structuration du compilateur et les optimisations a été repris en grande partie dans le compilateur Bliss-11 (‹Wulf75›).

Comme dans le compilateur C, les algorithmes de génération sont basés sur une recherche heuristique ; par ailleurs, une stratégie de planification est également employée pour l'allocation des registres et des temporaires. Toutefois, les modèles nécessaires à la génération sont construits automatiquement à partir d'une description de la machine ; dans cette description, un ensemble d'assertions est associé à chaque instruction, dans le but d'en décrire tous les effets (‹Cattell79a›).

Le constructeur de tables (‹Cattell80›) crée un modèle pour chacun des effets d'une instruction. Il crée, d'autre part, des modèles pour un ensemble d'arbres "intéressants", construits à l'aide d'un certain nombre de règles de dérivation (axiomes) ; ces règles garantissent, en particulier, la présence d'au moins un modèle pour chaque opérateur du langage intermédiaire (TCOL) et pour un transfert entre chaque type de ressources. D'autres modèles sont également construits pour les instructions de contrôle et pour les opérateurs sans équivalent dans la machine (la multiplication, par exemple) ; tous les modèles associés à un opérateur sont regroupés et classés suivant leur contexte d'évaluation. On peut remarquer que le constructeur des tables réalise à l'avance une partie de la sélection d'instructions, améliorant ainsi les performances du générateur.

La génération de code (‹Cattell79b›) est effectuée en un

parcours dans l'ordre inverse d'exécution du programme. Elle commence donc par la racine de l'arbre, en essayant de trouver le meilleur modèle pour son opérateur ; ce modèle sera, en général, celui qui est conforme au plus grand nombre de fils. Ce procédé est répété pour chacun des fils non conformes ; ils sont préalablement transformés de façon à forcer leur conformité avec le modèle ; un nombre minimal de modèles est ainsi nécessaire. Cette heuristique ne garantit pas la meilleure traduction mais uniquement une approximation ; toutefois, dans la pratique, elle se révèle être souvent la meilleure.

PQCC constitue l'un des modèles de génération les plus intéressants décrits dans la littérature. L'idée la plus remarquable est de réaliser une partie de la sélection d'instructions lors de la construction du générateur. PQCC a également mis en évidence les simplifications que peut apporter l'utilisation de techniques jusque là réservées à l'intelligence artificielle, dans la réalisation de systèmes complexes (‹Wulf80›).

Toutefois, la mise en oeuvre d'un tel système est relativement complexe et demande de gros investissements ; c'est probablement pour cette raison qu'il n'a été que partiellement réalisé. A notre connaissance, la phase de sélection d'instructions (CODE) est en effet la seule à avoir été mise en oeuvre ; les résultats obtenus sont très significatifs mais ne permettent pas d'évaluer globalement le système ; nous pensons, en particulier, que les interactions entre les différentes phases peuvent provoquer de nombreux conflits et remettre en cause certains choix. Par ailleurs, une étude (‹Leverett81›) sur la phase d'allocation des registres et des temporaires (TNBIND) a été réalisée ; malheureusement, elle n'a pas été menée à terme et, de plus, les algorithmes proposés ont été peu expérimentés.

Un modèle de génération similaire à celui de PQCC est proposé dans ‹Kozlak81›. Ce modèle est toutefois beaucoup moins ambitieux et comporte des limitations assez importantes ; ainsi, par exemple, il ne considère que 3 modes d'adressage : constantes immédiates, registres et adresses absolues. D'autre

part, un moniteur de contrôle du constructeur de tables est à écrire pour chaque nouveau générateur ; ce moniteur doit soumettre au constructeur la spécification des instructions de la machine et stocker les modèles obtenus ; il doit également contrôler la profondeur de la recherche et décider du nombre de modèles à associer à chaque instruction ; sa réalisation n'est donc pas triviale. L'application pratique du modèle de Kozlak nous semble par conséquent assez limitée.

Recherche exhaustive

Les heuristiques utilisées par les méthodes précédentes sont destinées à éviter une explosion combinatoire lors de la sélection d'instructions. Cependant, la possibilité d'une telle explosion n'est pas inhérente au problème de la sélection d'instructions ; en effet, la traduction d'une opération ne met en jeu qu'un petit nombre de modes d'adressage et d'instructions ; ainsi, par exemple, l'algorithme de Aho et Johnson (‹Aho76›) est d'ordre linéaire malgré son caractère exhaustif. Cette possibilité est par conséquent due à d'autres facteurs, liés au modèle de génération utilisé (par exemple, les générateurs PQCC risquent une telle explosion en raison du type de parcours effectué). La recherche exhaustive est donc envisageable, à condition que le modèle de génération choisi ne comporte pas de risques d'explosion combinatoire. Ce type de recherche a ainsi été utilisé pour la réalisation du système MUG2 (‹Ganzinger77›), à l'Université de Munich, et d'un compilateur C, à l'Université de Tufts (‹Krumme82›).

MUG2 est un système de production de compilateurs qui offre des outils de description, et les modules de génération correspondants, pour la réalisation des différentes phases d'un compilateur ; toutes les phases sont traitées et, en particulier, celles d'optimisation et de génération de code. Un programme est considéré comme un arbre décoré par un certain nombre d'attributs ; la traduction d'un programme correspond à une suite de passes d'évaluation et de transformation de ces attributs. La phase de génération de code (‹Ripken78›) utilise une méthode similaire à celle de Aho et Johnson.

Le programme à traduire est représenté par un graphe (MPROG) dont la structure reflète son flot d'exécution ; les noeuds de ce graphe sont des arbres d'expressions décorés. Les instructions de la machine sont représentées par des modèles arborescents : un modèle correspond à une règle de transformation et décrit, d'une part, les calculs réalisés par l'instruction et, d'autre part, les résultats de celle-ci ; une troisième information spécifie les instructions à générer, si la règle est applicable. Les modes d'adressage sont également représentés par des modèles arborescents, applicables exclusivement aux opérandes ; ces derniers correspondent aux feuilles de l'arbre et sont représentés par des descripteurs contenant la classe de cellule occupée (accumulateur, registre général, mot mémoire, etc.), l'adresse dans cette classe, la valeur de l'opérande et son domaine de définition (représentation, type, précision, etc.) ; l'allocation de mémoire aux opérandes et la construction de leurs descripteurs sont réalisées par une phase antérieure à la génération ("implementation"). Un troisième type de modèle est utilisé pour représenter les possibilités de transferts entre les différentes classes de cellules.

La génération de code est réalisée en 2 phases. La première phase applique à chaque arbre du MPROG les règles de transformation associées aux instructions de la machine ; le résultat de ces transformations est un ensemble de blocs d'instructions (un par arbre). La deuxième phase applique les règles de transformation associées aux arcs du graphe et génère ainsi les instructions de branchement correspondantes ; puis, elle assemble les blocs d'instructions avec les branchements.

La transformation d'un arbre s'effectue en 3 passes, très proches de celles de l'algorithme de Aho et Johnson. D'abord, une passe ascendante détermine la meilleure séquence de transformations pour chaque noeud, en fonction du nombre de registres disponibles et de la classe de cellule du résultat ; ensuite, une passe descendante propage les informations concernant les choix de la première passe (ordre d'évaluation,

allocations de registres, transferts entre cellules, etc) ; enfin, une passe ascendante effectue l'assignation des registres et des temporaires, puis génère les instructions choisies par la première passe.

Cette méthode produit pratiquement toujours une séquence optimale pour chaque arbre traité ; c'est l'énumeration exhaustive des modèles applicables à chaque arbre qui lui permet d'être optimal. Toutefois, cette même énumération provoque une baisse sensible des performances du générateur ; une implémentation partielle de la méthode (‹Holvoet82›) a en effet montré que l'algorithme est très lent. Une solution intéressante serait de réaliser, à l'instar de PQCC, une partie de la sélection d'instructions lors de la construction du générateur ; ainsi, certains arbres prédéfinis bénéficieraient d'un traitement très rapide et le traitement complet ne serait réservé qu'aux arbres peu courants.

Le modèle de Ripken constitue l'un des modèles de génération les plus intéressants. Il est regrettable qu'il n'ait pu mettre en oeuvre la totalité de son modèle afin d'en tirer des conclusions plus complètes.

Le compilateur C, développé à l'Université de Tufts (‹Krumme82›), utilise également une méthode basée sur une recherche exhaustive ; cette méthode est toutefois beaucoup plus simple que celle de MUG2. Il s'agit d'une méthode "intégrée", dans laquelle la détermination de l'ordre d'évaluation des expressions, l'allocation des registres et la sélection des instructions sont réalisées de façon simultanée ; les informations concernant la machine cible sont représentées comme des modèles arborescents qui pilotent entièrement la génération de code. La conception des algorithmes a été très soignée pour éviter tout risque d'explosion combinatoire ; les résultats obtenus sont très satisfaisants car les performances du générateur sont comparables à celles des générateurs ad-hoc ; par ailleurs, le code généré est de bonne qualité.

5.2.4. Autres.

D'autres techniques ont également été utilisées pour mettre en oeuvre l'approche descriptive ; nous en présentons deux dans ce qui suit.

Dans la première de ces techniques (<Lebarbier79>), l'idée de base est l'utilisation d'un même langage pour la description de la machine cible et du langage intermédiaire. Ce langage sert par ailleurs de pivot dans la traduction d'un programme ; en effet, le générateur de code traduit d'abord la représentation intermédiaire en ce langage pivot et c'est à partir de celui-ci qu'il génère le code objet. La difficulté de définir un langage intermédiaire universel a conduit Lebarbier à effectuer une classification des machines par famille, dans le but d'associer à chaque famille un langage intermédiaire adapté ; trois classes de machines ont ainsi été distinguées, ce qui explique la possibilité de paramétrer le générateur par une description du langage intermédiaire.

La génération de code est réalisée en 3 étapes. D'abord, le programme est traduit en langage pivot, en utilisant la description du langage intermédiaire ; pour ce faire, une technique de reconnaissance syntaxique classique est utilisée. Ensuite, les opérandes du programme sont transformés en opérandes de la machine cible ; cette transformation fait appel à une table de correspondances, fournie par le réalisateur lors de la construction du générateur. Enfin, la troisième étape réalise la traduction en code objet ; une technique similaire à celle de la première étape est utilisée mais il existe un risque d'échec ; dans ce cas, le générateur essaie de transformer l'opération en une opération équivalente mais contenue dans la description de la machine cible. Les transformations sont décrites par un ensemble de règles de "réécriture", similaires à celles de <Fraser77> ; ces règles sont indépendantes des machines et font donc partie du générateur ; il existe cependant la possibilité d'adjoindre de nouvelles règles, spécifiques à une machine. Les règles sont appliquées de façon exhaustive jusqu'à l'obtention d'une conformité.

Une mise en oeuvre de cette méthode a été réalisée en Prolog. Il ressort de son évaluation que la méthode est <u>très</u> lente ; en effet, la génération d'une instruction demande près de 30 secondes sur un IRIS-80, dans le cas le plus favorable (aucune transformation) ; si l'on considère que le Prolog utilisé était interprété, il faut diviser ce temps approximativement par dix (rapport interprétation/compilation), ce qui reste malgré tout considérable. Trois autres inconvénients majeurs sont à reprocher à cette méthode :

a) Il s'agit d'une méthode incomplète ; certains aspects de la génération sont "oubliés" (exemple : allocation de temporaires) ou bien reportés à une phase précédente (exemple : détermination de l'ordre d'évaluation).
b) Les langages intermédiaires utilisés sont proches d'un langage d'assemblage ; la génération de code dispose donc de peu d'informations contextuelles, ce qui limite les possibilités d'optimisation.
c) Le réalisateur doit spécifier certains aspects propres aux algorithmes de génération ; nous pensons, en particulier, à l'allocation de mémoire et à la transformation des identificateurs en adresses mémoire.

La deuxième technique (‹Davidson81›) propose un modèle de génération de code inhabituel. Dans ce modèle, la génération est réduite à une traduction simpliste entre le langage intermédiaire et le langage objet ; c'est l'optimiseur final qui se charge ensuite de transformer le résultat en un code de bonne qualité. Cet optimiseur est réalisé suivant l'approche descriptive ; en effet, il est constitué d'une partie algorithmique, indépendante des machines, et d'une partie données, regroupant toutes les informations sur la machine cible ; cette deuxième partie est construite automatiquement à partir d'une description de la machine. Par contre, le générateur de code est spécifique à chaque machine et doit par conséquent être réécrit pour chaque nouvelle machine ; pour faciliter cette réécriture, ses fonctions ont été réduites au strict minimum.

Cette technique a été utilisée pour mettre en oeuvre un compilateur pour le langage Y sur plusieurs machines (PDP-11, DEC-10, CDC Cyber 175). Le code généré est aussi bon, voire même meilleur, que celui des compilateurs disponibles sur ces machines (C UNIX, C portable, Pascal Zurich, Ratfor, etc.) ; par contre, la vitesse de compilation est de 5 à 6 fois inférieure. En ce qui concerne la mise en oeuvre du compilateur sur une nouvelle machine, l'auteur l'estime à quelques jours (3 à 5 !) ; nous considérons toutefois que cette estimation est trop optimiste.

6. Conclusion.

Nous avons passé en revue dans ce chapitre les différents modèles de génération de code recensés dans la littérature. Nous nous sommes particulièrement intéressés aux travaux d'automatisation de la production des générateurs.

Après cette analyse comparative et critique des différents travaux effectués, nous pouvons essayer de dégager les idées qui nous ont paru les plus intéressantes. Notre choix est fondé sur les trois critères suivants, énoncés dans l'ordre d'importance décroissante :

- qualité du code généré,
- coût de réalisation d'un <u>nouveau</u> générateur,
- performances vis à vis des compilateurs existants.

Ces idées sont les suivantes :

a) Utilisation d'une représentation intermédiaire arborescente. Celle-ci ne peut que faciliter la génération, en raison de sa souplesse de manipulation ; elle donne en outre la possibilité d'analyser un contexte plus vaste lors de la sélection d'instructions et d'obtenir ainsi un meilleur code.

b) Séparation des algorithmes de génération et des données relatives à la machine. Nous préférons donc l'approche descriptive car elle réduit considérablement l'effort de réalisation.

c) Construction automatique des tables de description de la machine. Cette construction est trop lourde et délicate pour

être réalisée manuellement ; par ailleurs, son automatisation donne la possibilité d'y rajouter un certain nombre de vérifications (description incomplète, incorrecte, etc) et d'optimisations. Elle contribue par conséquent à la réduction de l'effort de réalisation ainsi qu'à la production de bon code.

d) Intégration de toutes les tâches de la génération. Certains modèles ne s'attaquent qu'à un sous-ensemble de ces tâches ; le réalisateur est alors obligé de mettre en oeuvre lui-même les tâches restantes, avec tous les problèmes que cela représente. Un modèle complet nous parait donc souhaitable.

e) Sélection d'instructions par reconnaissance de formes. Le principal avantage de cette technique est de simplifier les algorithmes de sélection, en éliminant l'analyse cas par cas ; elle permet également de produire un code de meilleure qualité. En ce qui concerne le type des modèles (linéaires ou arborescents) et des techniques de reconnaissance associées (dérivées de l'analyse syntaxique ou de l'intelligence artificielle), nous ne ferons aucun choix car nous pensons qu'elles sont équivalentes.

f) Pour des modèles arborescents, trois idées sont à retenir :

   1) Les règles de transformation appliquées en cas d'échec doivent être simples et peu nombreuses ; les performances du générateur en dépendent directement. Ces règles doivent de préférence se limiter à des transferts entre ressources.

   2) Le choix du type de recherche (heuristique ou exhaustive) dépend des possibilités d'explosion combinatoire du modèle de génération utilisé. Nous préférons cependant la recherche exhaustive car elle produit un code localement meilleur.

   3) La sélection d'instructions peut être réalisée à l'avance pour les expressions les plus communes ; les performances du générateur seront ainsi améliorées de manière sensible.

g) Simplification de la génération de code par utilisation d'un optimiseur final. Certaines fonctions peuvent en effet être plus facilement réalisées après la génération ; le choix des instructions de branchement en constitue un bon exemple. Cette répartition des fonctions peut faciliter la conception

des algorithmes et permettre de générer un meilleur code.

Toutes ces constatations nous ont servi de base de réflexion pour la conception du système de génération de code, que nous présentons au chapitre suivant.

III. UNE SOLUTION A LA GENERATION DE CODE MULTICIBLE.

1. Cadre de travail.

Notre travail s'inscrit dans le projet Adèle (Atelier de DEveloppement de Logiciel), qui se propose d'offrir un environnement de programmation pour le langage PASCAL (‹Briat81›, ‹Estublier83›). Adèle est destiné à de petites équipes universitaires de recherche qui développent des applications dont l'objectif principal est de valider des concepts ; les prototypes qu'elles réalisent sont généralement complexes mais de taille moyenne (quelques dizaines de milliers de lignes) et de durée de vie limitée (peu ou pas de maintenance).

L'environnement est mis en oeuvre sur des postes de travail individuels, connectés à un ordinateur central CII-HB 68 DPS-3 sous le système Multics. Dans la version initiale, ces postes de travail sont des simples terminaux VT100 ; un de nos objectifs est toutefois de les remplacer par des ordinateurs 16 bits, équipés notamment d'un écran programmable au niveau du point et d'une souris pour la désignation ; ainsi, nous disposerons d'une puissance de calcul, nous permettant de réaliser localement une partie des fonctions d'Adèle.

Un programme construit sous Adèle est composé par un ensemble de modules connectés de façon hiérarchisée. Un module est l'association d'une interface qui est sa spécification fonctionnelle et d'un corps qui est la réalisation de cette spécification ; chacun de ces composants peut exister en versions multiples, leur gestion étant assurée par la partie archivage d'Adèle, suivant le modèle présenté dans ‹Cheval82› et ‹Estublier83›.

Un corps est conservé sous la forme d'une représentation

interne unique (RI) ; il s'agit d'un arbre abstrait syntaxique décoré avec des attributs lexicaux et sémantiques, et contenant toutes les informations du programme soit explicitement, soit moyennant un calcul simple. Cette représentation est la forme unique de manipulation d'un module par les différents outils de l'atelier (‹Mossière82›) : l'introducteur de programmes Pascal (sous la forme de fichier texte), l'éditeur syntaxique, le décompilateur, l'interpréteur/metteur au point et le générateur de code multi-machine. La figure III.1 illustre l'architecture générale d'Adèle.

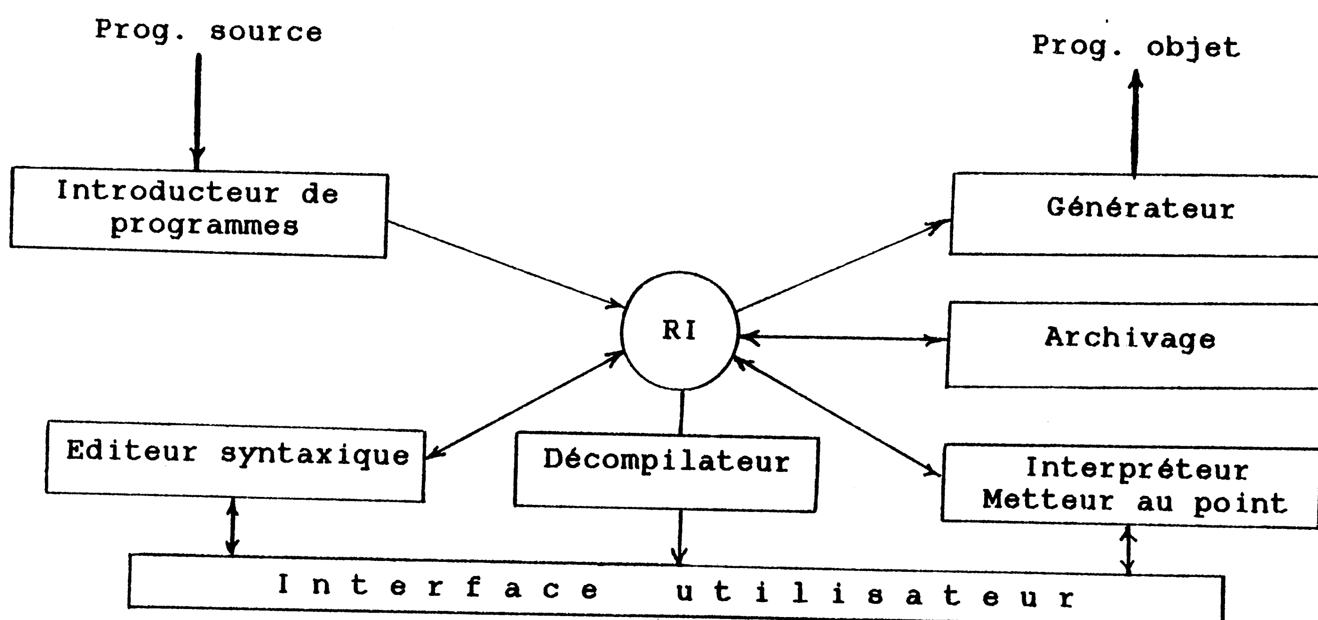


Figure III.1 Architecture générale d'Adèle.

L'utilisateur d'Adèle communique avec les différents outils à travers une interface utilisateur unique. Celle-ci est chargée de recupérer toutes les entrées faites par l'utilisateur via le clavier et la souris, et de les communiquer aux outils correspondants ; d'autre part, elle reçoit les sorties faites par les différentes applications et les affiche dans les fenêtres appropriées de l'écran (‹Herrmann82›).

Le cycle de développement d'un module est réduit à une seule phase qui intègre l'édition, l'interprétation et la mise au point ; l'utilisateur dispose simultanément des outils correspondants et peut les utiliser dans un ordre quelconque, sans sortir de l'environnement Adèle. Le module ne sera

"compilé" (traduit en binaire) qu'au moment où l'utilisateur considère qu'il est au point.

La traduction d'un module en code binaire est faite par le GEnérateur de code Multi-MachinE (GEMME) dont la présentation fait l'objet de cette thèse.

2. Objectifs.

Adèle supporte un seul langage source, choisi en raison de sa grande diffussion et de la disponibilité de ses compilateurs : il s'agit de Pascal (Norme ISO) avec les extensions nécessaires pour l'expression de la modularité ; nous avons toutefois essayé de faire en sorte qu'Adèle soit indépendant du langage, partout où c'était possible. Les extensions pour la modularité peuvent être transformées, au moment de la décompilation, de façon à être tout à fait conformes avec la norme ISO et à assurer ainsi la portabilité des programmes développés sous Adèle.

D'autre part, les postes de travail Adèle ne sont pas nécessairement uniformes, surtout en ce qui concerne le processeur central. En effet, nous comptons utiliser comme tels des SM90 (‹Finger82›), des Micromega (‹Thomson82›) et des Buroviseurs (‹Naffah79›) ; bien entendu, ce choix peut évoluer pour inclure d'autres machines.

Le générateur de code multi-machine est donc destiné à compiler un seul langage source mais pour un grand nombre de machines cibles.

Le problème de la génération de code est l'un des problèmes majeurs en compilation ; cette difficulté est accentuée dans notre cas, par la diversité des machines cibles. Pour cette raison, nous avons décidé de procéder en 2 étapes :

a) Développement d'une méthode formelle de génération de code, permettant de faire une construction relativement rapide de générateurs de code ; elle doit faire, en particulier, une séparation très nette entre les algorithmes de génération et

les données qu'ils utilisent.

b) Réalisation d'un système de construction automatique de générateurs de code, basé sur la méthode développée dans la première étape. Ce système doit prendre en entrée une description de la machine cible, l'analyser et l'organiser de façon à construire des tables qui doivent piloter le générateur de code. Ces tables doivent contenir toutes les informations nécessaires à la traduction des opérateurs de la RI Adèle en séquences de code de la machine cible ; elles seront constituées par des modèles de génération, composés d'une partie descriptive et d'une partie exécutive (ordres d'allocation et d'émission de code) ; un modèle sera sélectionné pour compiler une opération, si celle-ci est conforme à la description du modèle. La figure III.2 présente le schéma général d'un tel système de génération de code.

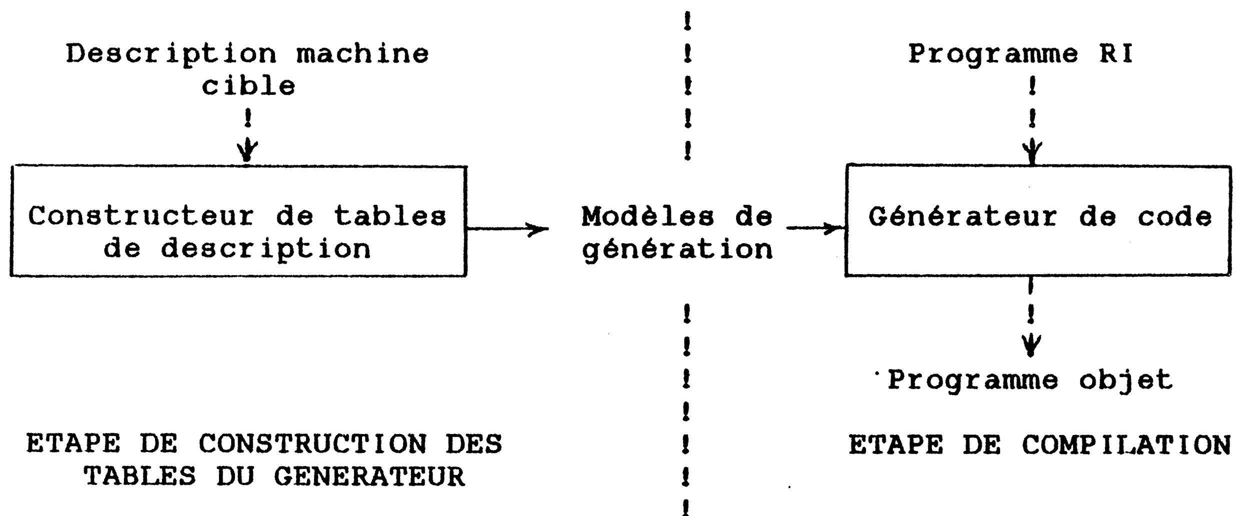


Figure III.2 Schéma de génération de code automatique.

L'objectif primordial du système de génération est de produire du code de bonne qualité, au minimum comparable à celle des compilateurs disponibles sur le marché et si possible meilleure. Pour atteindre cet objectif, nous avons concentré notre effort sur 2 aspects :

a) La sélection d'instructions machine pour une opération donnée. Notre but est d'exploiter au mieux les modes d'adressage et les instructions spécialisées de la machine cible ; ainsi, le code produit sera localement "optimal".

b) Les optimisations sur le code généré ; notre but est

d'analyser les relations entre les séquences d'instructions voisines et de les exploiter pour améliorer le code.
Nous avons traité ces 2 aspects de façon à les rendre complémentaires. D'autre part, nous avons réparti les tâches entre eux, selon un souci d'efficacité ; ainsi, certaines tâches, qui correspondraient normalement à la sélection, sont réalisés dans la partie optimisation et vice-versa.

Un autre objectif de notre projet est d'essayer de préserver autant que possible l'indépendance des aspects principaux du générateur vis à vis de Pascal ; ceci devrait nous permettre de changer plus facilement de langage source et de suivre l'évolution d'Adèle.

En ce qui concerne la classe de machines cibles, nous l'avons limitée dans un premier temps aux machines 16 bits, pour des raisons de simplicité. Nous nous intéressons surtout aux microprocesseurs 16 bits (Motorola 68000, Intel 8086, Zilog Z8000, National Semiconductors NS16000, DEC LSI-11, etc), vu le grand succès des uns et l'avenir prometteur des autres ; ces microprocesseurs doivent équiper la plupart de machines dans les années à venir et ils constituent déjà les processeurs des postes de travail choisis pour Adèle. Nous nous intéressons également aux minis 16 bits (Solar 16, Mini 6, Nord 10, etc) car ils sont encore très répandus, en particulier dans notre environnement de travail.

L'ensemble des machines visées appartient à la famille des machines à registres (généraux ou spécialisés). Un de nos objectifs à long terme est la généralisation de notre système à toute cette famille ; par contre, nous ne nous intéressons ni aux machines à pile ni aux machines spécialisées. Ceci ne semble pas être un handicap sérieux car la plupart des machines utilisées aujourd'hui sont des machines à registres.

Dans ce qui suit, nous présentons notre solution au problème de la génération de code paramétrée, avec les objectifs et contraintes énumérés ci-dessus. Tout d'abord nous proposons une méthode de génération de code ; ensuite, nous analysons la

faisabilité de son automatisation et, pour terminer, nous proposons un système de génération de code pour Adèle.

3. Notre méthode de génération de code.

Ce paragraphe est consacré à la présentation de la méthode formelle de génération que nous avons développée. En premier lieu, nous définissons un certain nombre de concepts qui nous aideront à mieux faire comprendre la méthode proposée ; ensuite, nous décrivons cette méthode ainsi que les structures de données qu'elle utilise ; pour terminer, nous montrons comment est faite la sélection du code qui correspond à une opération.

3.1. Concepts préliminaires.

3.1.1. Notion de coût.

Etant donné que nous voulons générer le meilleur code possible, il nous a été nécessaire d'introduire une notion de coût de réalisation d'une action ; elle doit nous permettre de comparer les différentes possibilités de mise en oeuvre d'une opération et de choisir celle de moindre coût.

Un coût est associé à toute action envisagée par le générateur de code : allocation d'un registre, utilisation d'un certain mode d'adressage, utilisation d'une certaine instruction machine, etc. Dans le cas d'une action composée, le coût est constitué par l'addition des coûts de toutes les actions élémentaires qui la composent ; par exemple, le coût d'une instruction est la somme du coût de l'adressage des opérandes et du coût de l'opération proprement dite.

En ce qui concerne l'aspect sémantique de notre notion de coût, nous n'avons pas voulu le figer pour rester plus généraux. En effet, nous voulons que le critère d'optimisation soit fixé par le réalisateur plutôt que par la méthode ; celui-ci peut préférer, suivant les circonstances, optimiser l'espace mémoire ou le temps d'exécution ou encore une combinaison des deux.

Nous avons donc choisi l'option suivante :

> "Un coût est un couple d'entiers (i, j), dont la sémantique est inconnue de la méthode de génération de code".

C'est le réalisateur qui fixe les valeurs de ces coûts et, en conséquence, leur signification.

- Deux opérations sont définies sur les coûts :

a) Relation "inférieur à".

$$(i, j) < (k, l) \iff \begin{cases} i < k & \text{ou} \\ i = k \text{ et } j < l \end{cases}$$

b) L'addition.

$$(i, j) + (k, l) = (i+k, j+l)$$

Remarquons dans l'opération de relation que le premier entier d'un coût est prioritaire par rapport au second. On peut ainsi privilégier un facteur d'optimisation ; par exemple, un utilisateur soucieux de l'encombrement mémoire prendra normalement comme critère de coût :

(taille mémoire, temps d'exécution).

Ainsi, les programmes compilés seront optimisés par rapport à la taille mémoire et, à taille égale, par rapport au temps d'exécution.

3.1.2. Descripteur de valeur.

Pendant la phase de génération de code, il est nécessaire de décrire chaque valeur qui sera manipulée à l'exécution ; cette description permet de déterminer les fonctions d'adressage qui seront utilisées lors de l'usage d'une valeur dans le code généré.

Les informations concernant une valeur et sa représentation sont souvent regroupées dans ce que l'on appelle "un descripteur de valeur". Cette technique est trés utilisée dans les générateurs de code (<Wilcox71>, <Cunin76>, <Cheval76>), à cause

de la simplification et l'uniformité qu'elle apporte.

Un descripteur contient des informations sur les caractéristiques d'une valeur (par ex. son type) ainsi que sur sa localisation à l'exécution (par ex. le registre qui la contient). Nous distinguons 3 types de descripteurs, suivant le type d'objet qui représente la valeur à l'exécution :

a) Descripteur de constante, si la valeur est une constante ; dans ce cas, le descripteur contient principalement la valeur elle même.

b) Descripteur de registre lorsque la valeur est contenue dans un registre ; l'information principale est alors le nom du registre.

c) Descripteur de mémoire pour les valeurs qui seront mises en mémoire ; on décrit alors l'adresse de la position mémoire concernée. Nous avons choisi de modéliser une adresse comme étant la somme de 3 éléments : une base, un déplacement par rapport à cette base et un index ; nous sommes capables de décrire à l'aide de ces éléments, les différents formes d'accès à une position mémoire ; en particulier, ils nous permettent d'exprimer la plupart des modes d'adressage existants.

   Chacun des éléments est représenté lui même par un descripteur. Les descripteurs de la base et de l'index peuvent être de n'importe quel type ; par contre, celui du déplacement doit nécessairement être un descripteur de constante.

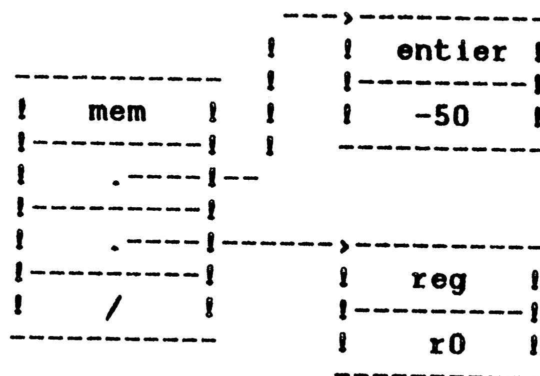


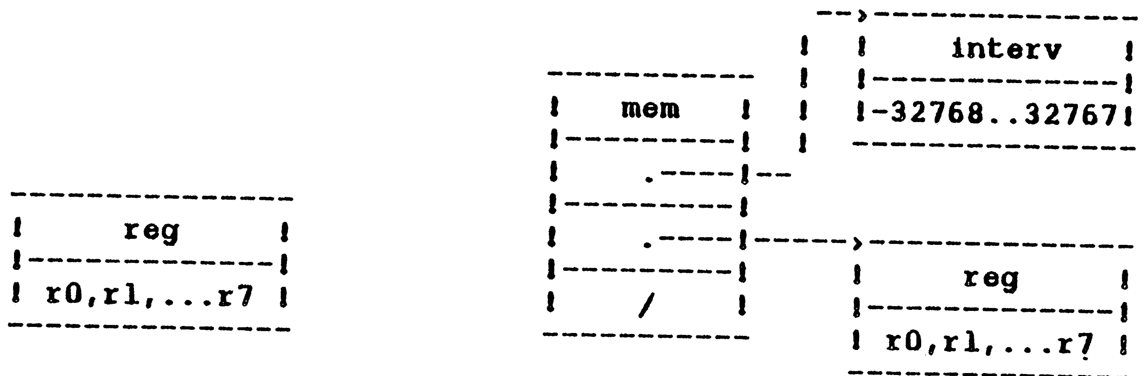
Figure III.3 Descripteur d'une position mémoire.

La figure III.3 montre le descripteur d'une position mémoire

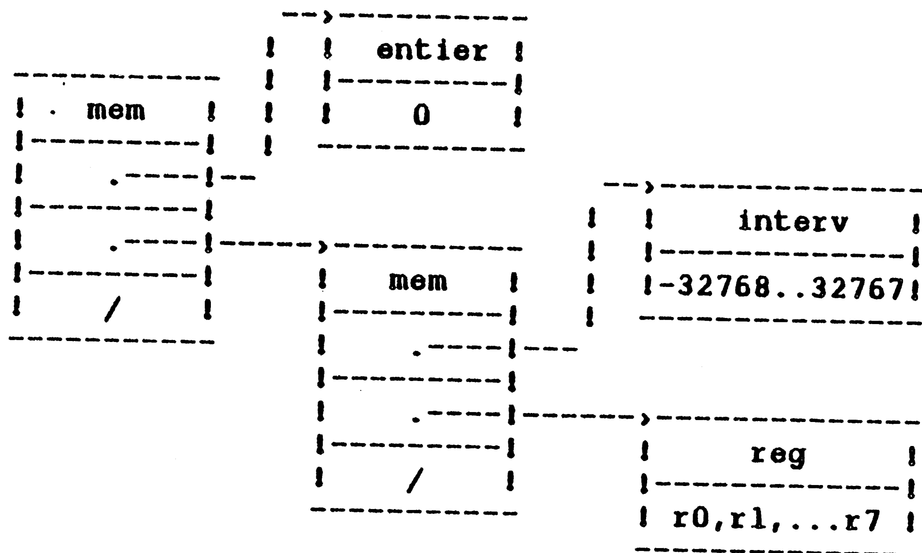
qui se trouve au déplacement -50 par rapport à l'adresse contenue dans le registre de base r0 ; cet exemple nous permet d'illustrer les 3 types de descripteurs définis et la représentation graphique que nous leur donnons au cours de cette thèse.

3.1.3. Modèle de descripteur.

Nous avons généralisé la notion de descripteur de valeur, de façon à pouvoir décrire avec un seul descripteur tout un ensemble de valeurs ; nous avons appelé le résultat un modèle de descripteur.



a) Registres généraux. b) Adressage basé.



c) Adressage indirect.

Figure III.4 Quelques modèles d'adressage pour le PDP-11.

Les valeurs décrites par un modèle doivent toutes avoir les

mêmes caractéristiques et le même type de localisation à l'exécution ; par exemple, un modèle peut décrire tous les registres de base d'une machine. Ces valeurs appartiennent donc à une certaine classe : constantes entières dans un certain intervalle, registres d'une même classe, etc.

Nous avons défini 4 types de modèles :

a) Modèle de constante qui représente une seule constante.
b) Modèle d'intervalle de constante, qui décrit un intervalle d'entiers.
c) Modèle de registre qui représente une classe de registres de la machine ; celle-ci est définie par énumération des registres qui la composent.
d) Modèle de mémoire, représentant un mode d'adressage mémoire. Ce type est semblable aux descripteurs de mémoire mais ses composants sont des modèles plutôt que des descripteurs.

Nous utilisons cette notion pour représenter les modes d'adressage de la machine cible. La figure III.4 donne des exemples de modèles pour les modes d'adressage du PDP-11 (‹Dec75›).

3.1.4. Conformité entre un descripteur et un modèle.

Notre méthode de génération fait appel à des techniques de reconnaissance de formes pour faire le choix des instructions à générer. Le problème consiste en effet à trouver le ou les modèles qui correspondent aux descripteurs des opérandes ; les "formes" utilisées sont donc des modèles de descripteurs.

Nous avons besoin alors d'une relation nous indiquant, pour un modèle et un descripteur donnés, si le premier correspond (est conforme) au second. L'algorithme de cette relation, que nous appelons conformité, est défini dans la figure III.5.

3.1.5. Opération.

Dans notre méthode de génération, une opération est l'une des 3 possibilités suivantes:

```
fonction CONFORMITE ( D : descripteur ; M : modèle ) : booléen ;
si D.Type = Constante
alors si M.Type = Constante
      alors CONFORMITE := D.Valeur = M.Valeur
      sinon si M.Type = Intervalle
            alors CONFORMITE := D.Valeur dans (M.Borne1, M.Borne2)
            sinon CONFORMITE := faux
sinon si D.Type = Registre
alors si M.Type = Registre
      alors CONFORMITE := D.NomRegistre appartient à M.NomsRegistres
      sinon CONFORMITE := faux
sinon si D.Type = Mémoire
alors si M.Type = Mémoire
      alors CONFORMITE := CONFORMITE(D.Déplacement, M.Déplacement) et
                          CONFORMITE(D.Base, M.Base) et
                          CONFORMITE(D.Index, M.Index)
      sinon CONFORMITE := faux ;
```

Figure III.5 Algorithme de conformité entre descripteur et modèle.

a) Une opération Pascal pour un type donné, telle qu'une addition entière, un "et" logique, etc.
b) Une opération abstraite, définie en général dans toutes les machines et équivalente à une opération Pascal, sous certaines conditions ; par exemple, le décalage logique peut servir à mettre en oeuvre une multiplication par une puissance de 2.
c) Une combinaison d'opérations pouvant être réalisée par une instruction ou une séquence simple d'instructions. L'instruction d'incrémentation de 1, telle que le "INC" du PDP-11, donne un exemple de ce type d'opération.

Cette définition nous est très importante car la sélection d'instructions machine de notre méthode est effectuée au niveau de l'opération.

3.1.6. Possibilité d'opération.

Un générateur de code peut traduire de plusieurs facons une même opération, selon les instructions et les modes d'adressage utilisés.

En effet, une machine dispose assez souvent de plusieurs instructions pour réaliser une même opération ; ainsi, par exemple, l'addition entière peut être faite dans le 68000 avec 5

instructions différentes : ADD, ADDA, ADDI, ADDQ et ADDX. Ce qui change d'une instruction à l'autre est l'ensemble de modes d'adressage qu'elle accepte pour chaque opérande, de même que le coût en espace mémoire et en temps d'exécution ; le choix d'une instruction est fait par conséquent en fonction des opérandes qu'il faut utiliser et du coût de l'instruction.

Chacune de ces instructions constitue pour nous une possibilité de mise en oeuvre de l'opération dans la machine cible ; c'est ce que nous appelons une possibilité d'opération.

3.2. Description de la méthode.

La plupart des méthodes de génération de code présentées dans la litérature n'essayent de résoudre qu'une partie du problème de la génération. Aho et Johnson (‹Aho76›, ‹Aho77›), par exemple, se sont intéressés à la détermination de l'ordre d'évaluation d'une expression ainsi qu'au traitement des sous-expressions communes ; Newcomer (‹Newcomer75›), Glanville-Graham (‹Glanville78›) et Akin-Leblanc (‹Akin82›), de leur côté, se sont intéressés à la sélection d'instructions machines, et ainsi de suite.

Cependant, certains travaux ont essayé de donner une solution complète à la génération de code, soit en proposant une méthode (‹Elson70›, ‹Krumme82›), soit en réalisant un compilateur qui intègre plusieurs méthodes partielles de génération (‹Wulf75›, ‹Crawford82›).

La méthode que nous proposons prend en compte tous les aspects de la génération de code, mais à des niveaux d'importance différents ; d'une façon générale, on peut dire qu'elle est proche de celle de Krumme (‹Krumme82›). Nous avons attaché une importance primordiale à la sélection d'instructions car, à l'instar d'Aho et Ullmann (‹Aho77›), nous considérons qu'elle est la source la plus riche d'optimisations ; notre seconde préoccupation a porté sur l'allocation des variables et le traitement des sous-expressions communes. Pour les autres aspects, nous avons préféré soit utiliser des techniques

classiques, soit chercher des solutions simples.

Les caractéristiques principales de la méthode sont les suivantes:

a) La sélection d'instructions est basée sur une recherche exhaustive plutôt que sur des heuristiques. Deux raisons nous ont conduit à faire ce choix :
   - il permet de trouver le meilleur code pour une recherche donnée,
   - les algorithmes sont plus simples et plus généraux.

   La recherche effectuée est donc combinatoire car elle analyse toutes les possibilités, mais le coût n'est pas prohibitif comme nous le verrons plus loin.
b) Toutes les données relatives à la machine cible sont conservées dans des tables, consultées par les algorithmes de la méthode. Ainsi, avec les mêmes algorithmes nous pouvons traiter des machines différentes ; il suffit alors de remplacer ces tables par celles qui décrivent la nouvelle machine cible.
c) Les différentes tâches de la génération sont faites simultanément, car nous considérons qu'il y a une forte interaction entre elles. Par exemple, le choix d'une instruction peut impliquer l'allocation d'un registre, laquelle peut nécessiter la libération d'un registre occupé, c'est à dire l'allocation d'un temporaire et le choix d'une instruction de sauvegarde ; par conséquent, une tâche a très souvent besoin des choix faits par les autres tâches. Leur séparation en sous-phases indépendantes implique une communication importante de structures de données entre elles, ou bien, la prédiction des choix qui seraient faits par les autres lors de chaque interaction.

Notre générateur de code est un interpréteur piloté par les tables de description de la machine cible. Il reçoit en entrée un arbre qu'il parcourt et décompose en opérations simples ; pour chaque opération, il effectue un parcours exhaustif des possibilités offertes par la machine pour cette opération, et génère la meilleure séquence d'instructions trouvée.

Durant le parcours, l'interpréteur traite les différents noeuds suivant leur type :

a) Définition d'un type Pascal.

Il associe une représentation dans la machine cible à ce type ; s'il s'agit d'un type simple, il est associé à un type de base de la machine ; s'il est structuré, ses composants sont traités d'abord pour être ensuite organisés de façon à optimiser leur accès.

b) Déclaration d'une variable.

Il lui associe une ressource, qui contiendra sa valeur à l'exécution (un registre ou une position mémoire), et le descripteur correspondant. La représentation de la variable est celle de son type.

Les informations sur la machine cible, utilisées par ces 2 tâches sont regroupées dans la table des ressources.

c) Instructions Pascal.

Chaque instruction est traitée de manière ad-hoc, dans le but d'optimiser sa traduction ; nous essayons, en particulier, d'exploiter les instructions spécialisées de la machine qui pourraient faciliter cette traduction. Nous avons accordé une grande importance à l'optimisation des instructions "case", "for" et à l'appel de procédure. Les informations nécessaires, concernant la machine cible, se trouvent dans la table d'instructions.

d) Expressions.

L'interpréteur détermine d'abord le meilleur ordre d'évaluation de l'expression et réorganise en conséquence le sous-arbre qui représente l'expression ; il fait ensuite un parcours postfixé du nouveau sous-arbre, le décomposant en opérations. Il s'occupe aussi du traitement des sous-expressions communes qu'il rencontre. Cette tâche fait appel à la table des ressources de la machine cible.

e) Opérations.

L'interpréteur effectue la sélection d'instructions pour l'opération concernée ; cette sélection est faite sur les différents modèles de génération de la machine cible qui sont associés à l'opération ; elle donne comme résultat la meilleure séquence d'instructions possible. Les informations nécessaires sont réunies dans la table d'opérations.

Nous présentons cet aspect de façon plus détaillée dans les paragraphes suivants.

La figure III.6 montre un schéma de l'interprétation et des tables qui sont utilisées. Nous pouvons remarquer que toutes les tâches consultent la table des ressources ; ceci est fait quand elles sont besoin d'allouer un registre, un temporaire ou une constante. Les techniques d'allocation utilisées dans tous les cas sont les plus simples ; par exemple, pour les registres, tant qu'il y a un registre libre de la classe demandée nous l'allouons, sinon nous prenons un registre occupé et sauvegardons sa valeur avant de l'allouer (technique "spill on the fly").

```
Type noeud               Tâche                        Table
                                                              -----
                 ----------------------<-----------------!   !
Def. type    -->!     Traduction       !                 !   !
                !       types          !<--!-------------!   !   !
                !----------------------!   !Table  types!   !   !
Decl variable -->!     Allocation       !<--!-------------!   ! T !
                !      variables       !                 !   ! A !
                !----------------------!<----------------!   ! B !
                !              ! arithm.!                 !   ! L !
                ! Traitement   !--------!                 !   ! E !
Expression   -->!    des       !logiques!<----------------!       !
                ! expressions  !--------!                 !   ! R !
                !              ! d'accès!   !-------------!   ! E !
                !----------------------!   !   Table     !   ! S !
                !              !   if   !<--!instructions!   ! S !
                !              !--------!   !-------------!   ! O !
                ! Traitement   !  for   !                 !   ! U !
Instruction  -->!    des       !--------!<----------------!   ! R !
                !instructions  !  ...   !                 !   ! C !
                !              !--------!<--!-------------!   ! E !
                !              ! affect !   !   Table     !   ! S !
                !----------------------!   ! opérations  !   !   !
Opération    -->!     Traduction       !<--!-------------!   !   !
                !   des opérations     !                 !   !   !
                 ----------------------<-----------------!   !
                                                              -----
```

Figure III.6 Schéma d'interprétation d'un arbre.

Dans ce qui suit, nous présentons l'organisation de la table des opérations qui constitue le pivot de notre générateur.

3.3. Modèles de génération.

La table des opérations contient l'ensemble des possibilités de traduction des opérations pour la machine cible ; chaque entrée correspond à un opérateur et contient la liste de possibilités qui lui sont associées.

Une possibilité d'opération correspond à ce que nous avons appelé auparavant un modèle de génération, avec sa partie descriptive et sa partie exécutive. La partie descriptive nous sert à la reconnaissance du modèle lorsque l'on fait une recherche ; elle donne le coût de base de la possibilité et décrit les opérandes qu'elle admet ainsi que le résultat qui sera obtenu. La partie exécutive définit la liste d'instructions qu'il faudra générer, si le modèle est choisi. La figure III.7 donne un exemple de possibilités d'opération pour l'addition entière dans le Solar 16 (‹Sems78›).

Po 1 :
Op. gauche = Pa 1
Op. droit = Pa 5
Résultat = op. gauche
Coût = (1500, 1)
Instructions= AD %d

Po 2 :
Op. gauche = Pa 2
Op. droit = Pa 2
Résultat = op. droit
Coût = (750, 1)
Instructions= ADR %g,%d

Po 3 :
Op. gauche = Pa 4
Op. droit = Pa 2
Résultat = op. droit
Coût = (750, 1)
Instructions= ADRI %g,%d

Po 4 :
Op. gauche = Pa 5
Op. droit = Pa 3
Résultat = op. gauche
Coût = (2375, 1)
Instructions= IC %g

Figure III.7 Possibilités d'opération (Po) du "+" entier.

La description d'un opérande est faite par l'intermédiaire de ce que nous appelons une liste de possibilités d'adressage ; les éléments de cette liste correspondent aux modes d'adressage qui peuvent être utilisés par la possibilité d'opération pour accéder à cet opérande.

Une possibilité d'adressage est elle-aussi un modèle de génération dont la partie descriptive est constituée principalement par un modèle de descripteur ; c'est ce modèle

qui sera comparé avec le descripteur de l'opérande correspondant, pour déterminer leur éventuelle conformité. Nous donnons dans la figure III.8, les possibilités d'adressage qui sont utilisées dans l'exemple précédent (figure III.7) ; remarquer que les coûts sont nuls : dans cette machine, le coût de chaque possibilité comprend déjà les coûts des opérandes.

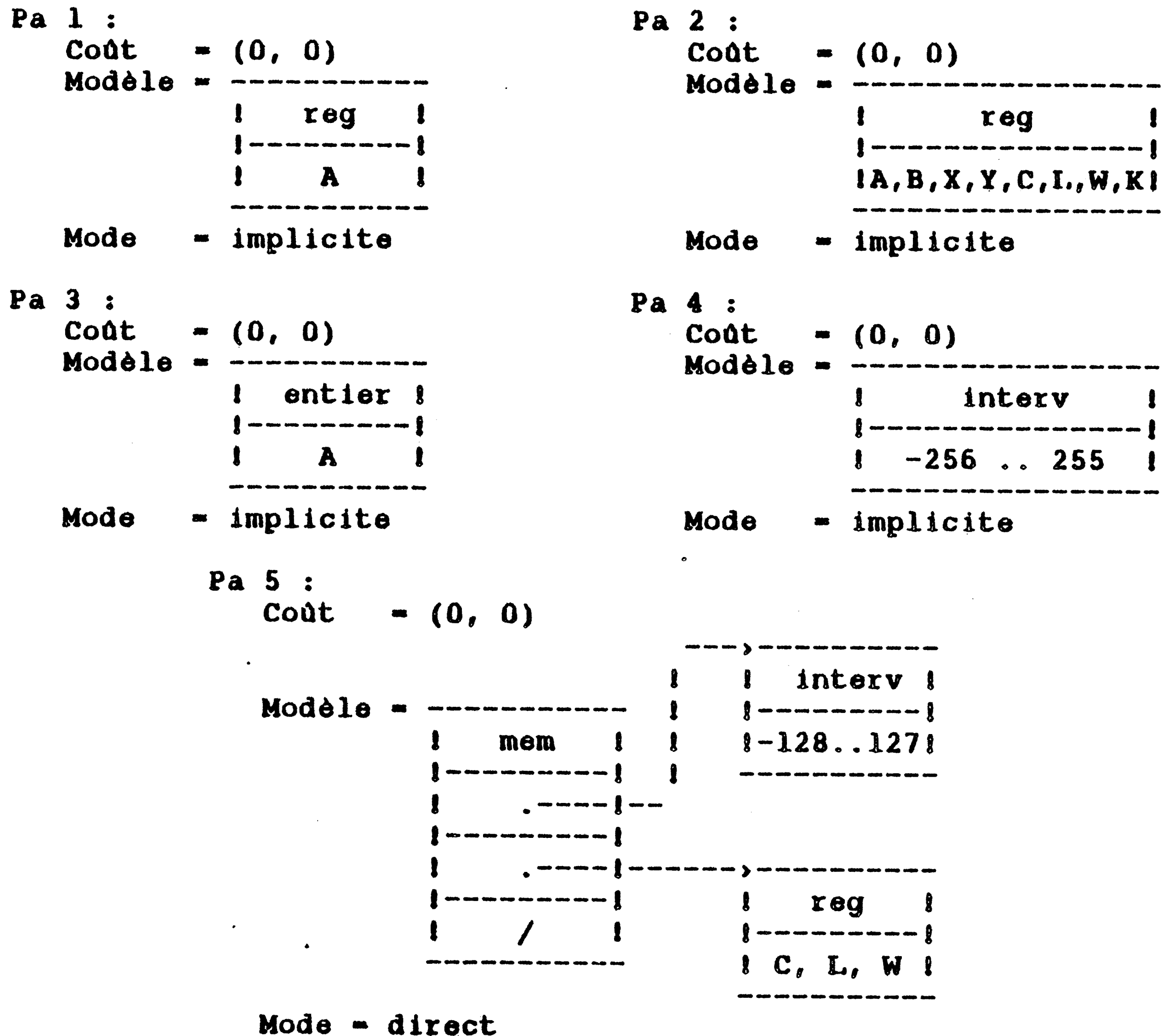


Figure III.8 Possibilités d'adressage pour le Solar 16.

Cette structure de données est consultée lorsque l'on veut faire la sélection d'instructions pour une opération donnée. Nous présentons ci-après la méthode utilisée pour faire cette sélection.

3.4. Sélection d'instructions pour une opération.

Nous avons vu que le générateur de code proposé effectue un

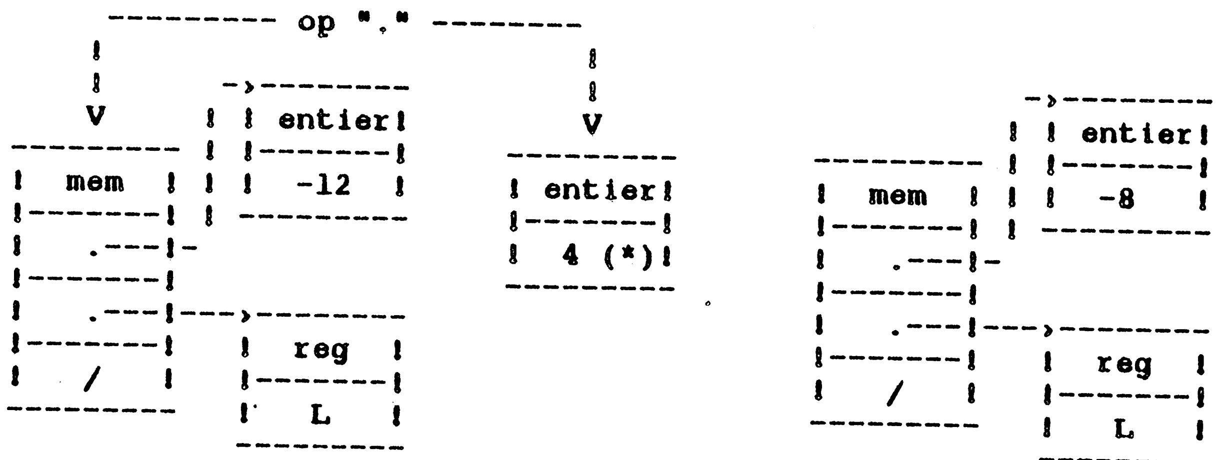
parcours de l'arbre du programme déterminé par le type de noeud visité ; il fait, en particulier, un parcours postfixé des expressions, après les avoir réorganisées suivant leur meilleur ordre d'évaluation ; c'est durant ce parcours qu'est faite la sélection des instructions machine qui correspondent aux opérations rencontrées.

L'aspect contrôle d'exécution des instructions Pascal (itérations, instr. conditionnelles) donne lieu à un deuxième type de sélection ; en effet, la traduction de cet aspect est faite en utilisant soit des instructions machine spécialisées, soit des instructions de branchement courantes. Comme c'est la sélection correspondant aux opérations qui a les aspects les plus intéressants, nous ne nous intéressons qu'à elle dans la suite.

Lorsque le générateur est sur un noeud opération, il visite d'abord les fils du noeud (les opérandes), les traite de façon appropriée et construit les descripteurs de leurs résultats ; ensuite, il récupère le numéro d'entrée qui correspond à l'opération dans les tables de description. Une fois qu'il dispose de toutes ces informations, il active le mécanisme de sélection et génère les instructions choisies ; enfin, il construit le descripteur du résultat de l'opération.

Un fils d'une opération peut être lui-même une opération ; dans ce cas, il subit le même traitement et donne lieu à une séquence de code qui précédera celle de son père. Autrement, il peut être soit une feuille, soit une opération d'accès ; le premier cas correspond à la désignation d'une variable ou d'une constante du programme ; le second correspond à un opérateur d'accès défini par le langage source, tel que l'accès à un élément de tableau ou à un champ d'un enregistrement. S'il s'agit d'une feuille, aucune instruction n'est générée ; une opération peut par contre donner lieu à une génération d'instructions. Par ailleurs, les opérations d'accès nécessitent une transformation du descripteur de la structure désignée, de façon à obtenir le descripteur de l'élément auquel on doit accéder ; un exemple de transformation d'une

construction Pascal est donné dans la figure III.9.



(*) Dépl champ / la structure

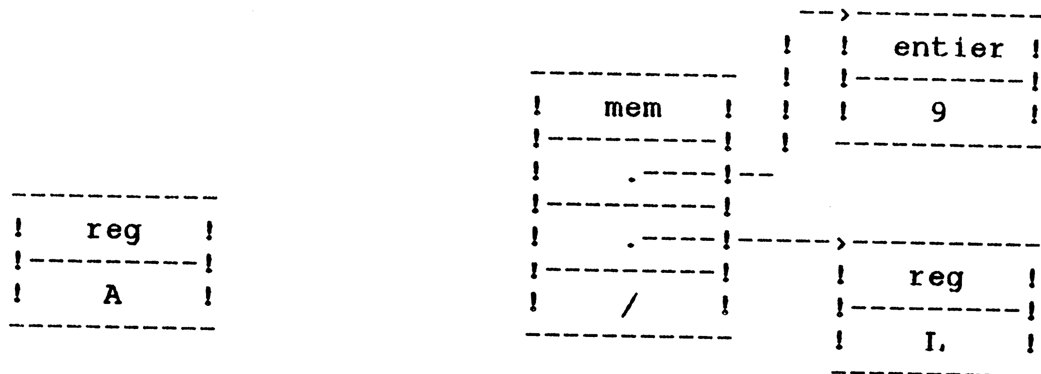
a) Représentation de Personne.Age   b) Après transformation

Figure III.9 Traitement de l'accès à un champ d'enregistrement.

L'objectif du mécanisme de sélection d'instructions est de trouver une possibilité de réalisation de l'opération telle que les descripteurs des opérandes soient conformes aux modèles de cette possibilité et que le résultat puisse être logé où celle-ci le demande ; nous disons alors que la possibilité donne une conformité parfaite. Si plusieurs possibilités sont dans ce cas, c'est la moins chère qui est choisie.

Ce mécanisme effectue une recherche exhaustive dans la liste de possibilités de l'opération, appliquant à chaque élément l'algorithme de conformité ; à la fin de cette recherche, il génère les instructions définies dans le modèle de génération choisi. Par exemple, supposons qu'on lui demande d'effectuer un "+" entier pour le Solar 16 (voir figures III.7 et III.8), entre le registre accumulateur et une position mémoire (leurs descripteurs sont définis dans la figure III.10) ; il analysera alors les 4 possibilités du "+" et ne trouvera une conformité parfaite que pour la première ; il générera, en conséquence, l'instruction "AD", en remplaçant le "%d" par les informations qui concernent l'opérande droit (la position mémoire).

Bien entendu, la recherche d'une conformité parfaite peut échouer. Dans ce cas, le mécanisme de sélection essaye de



a) Registre accumulateur

b) Une position mémoire.

Figure III.10 Descripteurs de variables pour le Solar 16.

transformer, pour chaque possibilité, le ou les opérandes qui ne sont pas conformes à leurs modèles ; la transformation doit essayer de forcer cette conformité, en transférant la valeur (si c'est possible) dans une ressource telle que son descripteur soit conforme au modèle. Il choisit la possibilité de moindre coût, en prenant en compte le coût des transformations nécessaires ; pour terminer, il génère le code associé à la possibilité choisie, d'abord celui qui correspond aux transformations et ensuite celui de la possibilité elle même.

Dans notre exemple, si l'on remplace le registre accumulateur par une autre position mémoire, aucune possibilité n'aura une conformité parfaite. Le mécanisme de sélection essaie donc de faire des transformations et trouve que c'est la première possibilité qui est la plus intéressante ; en effet, elle n'a besoin que d'une seule transformation : le premier opérande doit être chargé dans l'accumulateur par un "load" ; la deuxième possibilité, par contre, nécessite 2 chargements et la troisième aboutit à une impasse puisqu'on ne peut pas transformer une position mémoire en constante immédiate.

Si l'opérateur traité est commutatif, le mécanisme de sélection essaye les opérandes dans les 2 ordres possibles, ce qui lui permet de toujours trouver la meilleure séquence.

4. GEMME : Un GEnérateur de code Multi-MachinE.

Nous proposons dans ce paragraphe un système de génération de code multi-machine, adapté aux besoins du projet Adèle ; ce système est paramétré par une description de la machine cible et utilise la méthode de génération présentée plus haut.

4.1. Présentation du système de génération de code.

Dans ce qui suit, nous faisons une présentation générale du système de GEnération de code Multi-MachinE (GEMME) que nous avons développé.

4.1.1. Organisation générale.

L'approche que nous avons choisie consiste à créer un générateur de code pour chaque nouvelle machine ; le système comporte donc 2 étapes très distinctes :

a) construction d'un générateur de code,

b) utilisation du générateur ainsi construit.

La figure III.11 illustre cette approche.

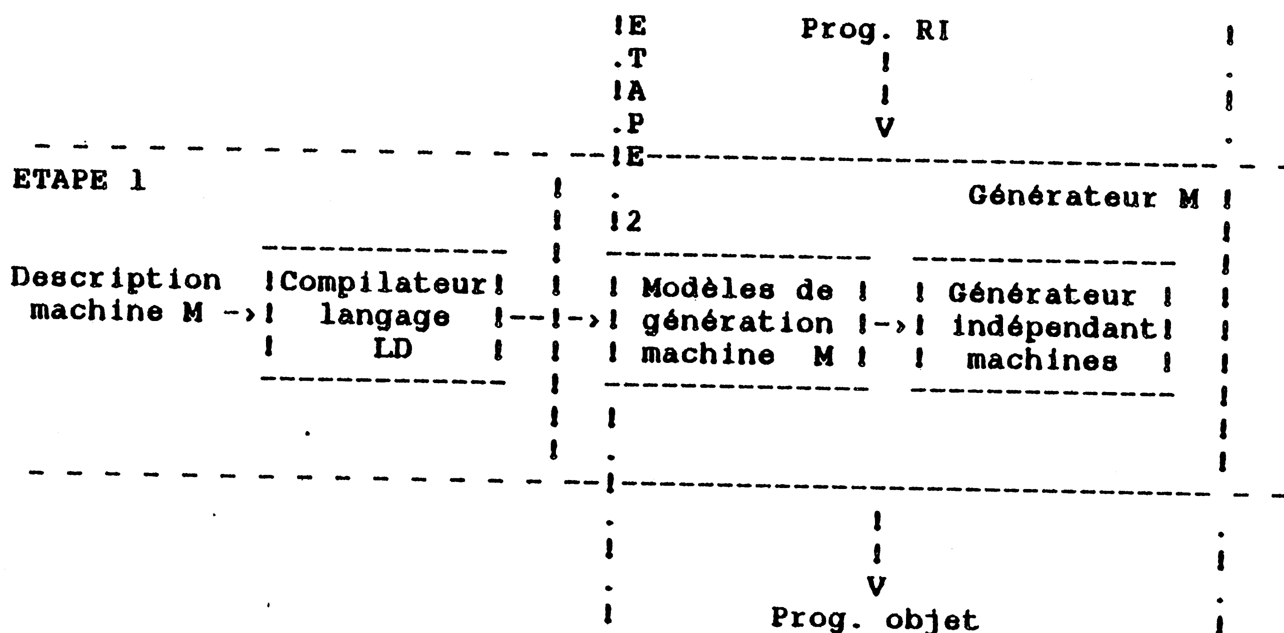


Figure III.11 Organisation générale du système GEMME.

Dans un système conventionnel, la première étape correspondrait à l'écriture du compilateur pour la nouvelle

machine ; dans notre cas, cette construction est automatisée. La seconde étape correspond à la compilation d'un programme.

4.1.2. Description de la machine cible.

La description de la machine cible est une spécification partielle de cette machine ; elle ne décrit, en effet, que les parties utilisables par le système de génération de code. Par exemple, elle doit contenir une description des instructions qui font une addition entière ; par contre, elle n'aura jamais la description des instructions de masquage ou de démasquage des interruptions.

Une description doit respecter une structure très hiérarchisée, figée par le langage de description. La structure choisie est très proche de celle des manuels de reférence des constructeurs ; on décrit dans l'ordre :

a) les objets manipulables par la machine (mot, octet, etc),
b) les ressources disponibles (les registres et la mémoire),
c) les classes de registres,
d) les modes d'adressage,
e) les instructions.

Chaque niveau définit des éléments de même nature et utilise pour ce faire des informations des niveaux précédents ; par exemple, une classe de registres est définie par une liste de registres, déclarés au niveau ressources. Le langage LD est par conséquent un langage uniquement déclaratif puisqu'il s'agit de donner une vision statique de la machine.

Une description contient également certains choix de compilation ; elle doit indiquer, par exemple, quels sont les registres de base qui peuvent être utilisés pour adresser les informations du programme.

Tous ces points sont traités de façon plus détaillée dans le chapitre V.

4.1.3. Construction d'un générateur de code.

La construction d'un générateur de code spécifique à une machine se fait par assemblage d'un générateur de code indépendant des machines et des données qui décrivent la machine cible. Le générateur correspond à la partie algorithmes et est unique pour toutes les machines ; les données, par contre, sont spécifiques à la machine et ce sont elles qui dirigent la génération de code.

Les données utilisées pour la construction du générateur sont obtenues par compilation de la description de la machine ; elles sont organisées en 5 tables :

a) Ressources de la machine, décrivant les objets de base qu'elle peut manipuler ainsi que les registres, les temporaires et les constantes non immédiates ; elle décrit aussi l'organisation de la mémoire à l'exécution.
b) Possibilités d'adressage, regroupant les différentes combinaisons de modes d'adressage qui sont acceptées par les instructions de la machine. Un exemple de possibilité d'adressage est la classe EA ("Effective Address") du 68000, qui regroupe la plupart des modes d'adressage de ce microprocesseur et qui peut être utilisée dans la majorité de ses instructions.
c) Opérations, qui contient les possibilités de réalisation de chaque opération.
d) Instructions, semblable à la table des opérations, mais pour les instructions complexes de Pascal (affectation de structures, for, appel de procédure, etc).
e) Branchements, décrivant les différents types de branchement de la machine.

Le compilateur LD rassemble toutes ces informations à partir de la description de la machine ; la plupart des informations sont directement disponibles mais d'autres doivent être déduites. Sa tâche principale consiste donc à organiser ces informations dans les tables, pour faciliter les recherches et optimiser le générateur.

L'intégration de ces tables au générateur indépendant des machines produit par conséquent un compilateur Pascal pour la machine décrite.

4.1.4. Compilation d'un programme Pascal.

Dans l'environnement Adèle, la compilation d'un programme correspond à sa traduction en code objet ; l'analyse du programme (syntaxique et sémantique) est faite de façon incrémentale par l'éditeur syntaxique. C'est donc la forme intermédiaire (RI) du programme qui sert de départ à sa compilation.

La compilation est lancée par l'utilisateur quand il considère que son programme est correct ; le générateur de code correspondant à la machine cible désirée est alors activé. Le programme ne sera compilé que s'il est complet et s'il a satisfait l'analyse sémantique.

Nous présentons dans ce qui suit la structure d'un générateur de code, de même que les traitements qu'il réalise pour compiler un programme.

4.2. Architecture générale d'un générateur de code.

Un générateur de code est structuré en 4 phases indépendantes, illustrées dans la figure III.12. Chaque phase travaille sur une représentation déterminée du programme ; elles se décomposent en sous-phases qui s'exécutent soit séquentiellement, soit comme des co-routines.

Les 4 phases d'un générateur sont les suivantes :

a) Optimiseur global.

Il effectue un ensemble d'optimisations indépendantes de la machine ; parmi ces optimisations, nous pouvons citer : le calcul d'expressions statiques, la propagation de constantes, l'identification de sous-expressions communes, la simplification d'expressions, etc. L'entrée de l'optimiseur est le programme RI ; en sortie, on obtient une autre

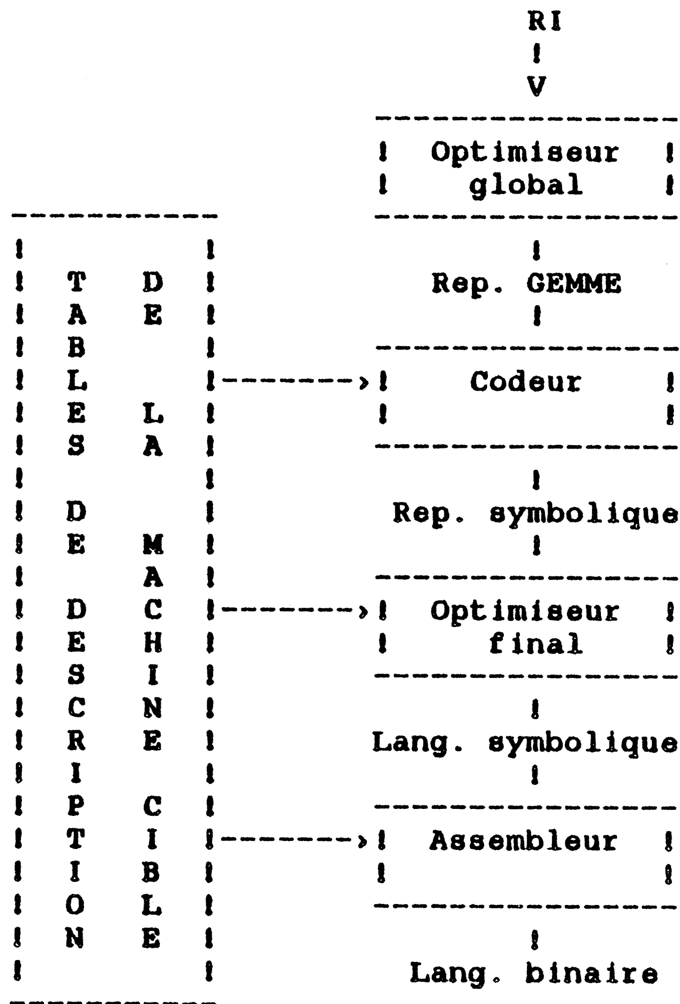


Figure III.12 Architecture d'un générateur de code.

représentation, proche de la RI, mais plus adaptée à la génération de code. Un certain nombre d'informations sont transmises dans cette représentation à la phase suivante.

b) Codeur.

Cette phase réalise la génération de code proprement dite. Le codeur effectue tout un ensemble de tâches en parallèle :

- traduction de types,
- allocation de variables,
- traitement des sous expressions communes,
- allocation de registres,
- allocation de temporaires,
- sélection d'instructions.

C'est cette phase qui met en oeuvre la méthode de génération de code que nous avons proposée.

La sortie est une représentation symbolique du langage de la

machine cible. traduction à ce stade n'est pas complète car le choix des instructions de branchement n'est fait que dans la phase suivante ; un branchement sera donc représenté par un bloc contenant toutes les informations nécessaires à sa traduction.

c) Optimiseur final.

Il fait un ensemble d'optimisations sur le code généré mais aussi la sélection d'instructions de branchement. La représentation utilisée en sortie est la même qu'en entrée, sauf en ce qui concerne les branchements ; d'autre part, les étiquettes sont éliminées puisqu'elles ne sont plus nécessaires. Parmi les optimisations réalisées, nous pouvons citer : l'élimination des affectations redondantes, la suppression des séquences inaccessibles, la factorisation de séquences identiques à la jonction de branchements inconditionnels, etc.

Le problème d'optimisation du code généré est traité de façon plus étendue dans le chapitre suivant.

d) Assembleur.

Cette phase transforme le programme en binaire, en consultant les tables de description du code objet de la machine ; sa tâche est simplifiée en ce sens que la résolution d'étiquettes est déjà faite.

IV. OPTIMISATION DU CODE GENERE.

Nous avons vu que le code produit par notre méthode de génération est optimal au niveau local ; c'est à dire que la séquence d'instructions choisie pour compiler une opération PASCAL est toujours la moins coûteuse pour la machine décrite. Cependant, la concaténation de toutes ces séquences optimales comporte en général des instructions inutiles.

En effet, les rapports entre 2 séquences voisines (au sens dynamique) ne sont pas analysés au moment de la génération, bien qu'ils peuvent donner lieu à de nouvelles opportunités d'optimisation. Par exemple, on peut faire le chargement de la même valeur dans un même registre dans 2 séquences voisines! ou encore, générer un branchement à une instruction qui est elle-même un branchement!

Un bon nombre d'optimisations sont ainsi réalisables sur le code généré (‹McKeeman65›, ‹Wulf75›, ‹Szymanski78›) et peuvent permettre des gains importants aussi bien en espace mémoire qu'en temps d'exécution (‹Rudmik79›, ‹Davidson81›, ‹Tanenbaum82›).

Ces optimisations, dites finales (ou "peephole"), ont pour objectif d'exploiter les relations existantes entre les instructions voisines ainsi que les propriétés particulières de chaque instruction ; de ce fait, elles sont fortement dépendantes de la machine cible. C'est pour cette raison d'ailleurs que la plupart des réalisations dans ce domaine ne prennent en compte que des optimisations spécifiques à la machine traitée (‹Wulf75›, ‹Rudmik79›, ‹Lamb81›).

L'optimiseur final d'un générateur de code GEMME met en oeuvre un certain nombre d'optimisations finales mais de façon indépendante de la machine. Il réalise aussi un autre traitement

qui est du ressort de la génération mais qui peut être mieux fait à son niveau ; il s'agit du choix des instructions de branchement, lorsque la machine traitée en dispose de plusieurs sortes (branchements courts et longs, par exemple).

Dans ce chapitre, nous analysons d'abord toute cette famille d'optimisations pour choisir ensuite celles qui s'adaptent bien à notre contexte.

1. Revue des techniques d'optimisation.

Beaucoup d'optimisations sur le code généré ont été expérimentées, depuis leur introduction par McKeeman (‹McKeeman65›). Certaines ont eu beaucoup de succès tandis que d'autres ne sont restées qu'au stade expérimental ; nous ne nous intéressons ici qu'aux premières.

Les opportunités qui donnent lieu à ces optimisations sont généralement dues à certains problèmes rencontrés à la compilation. On peut regrouper ces opportunités en 4 classes que nous présentons ci-après, de même qu'un certain nombre d'exemples qui les illustrent.

1.1. Séquences d'instructions redondantes.

Dans un programme compilé, il est fréquent de trouver des séquences d'instructions qui sont redondantes. Une séquence est redondante quand le résultat qu'elle cherche a déjà été obtenu par une autre séquence qui s'est exécutée avant ; la première séquence d'instructions est donc inutile. Il est donc intéressant d'éliminer ces séquences car nous pouvons gagner aussi bien en place mémoire qu'en temps d'exécution.

Nous en décrivons ici les 2 cas les plus intéressants :

a) Sous-expressions communes.

Une sous-expression commune est une expression qui apparait plusieurs fois dans le flot de contrôle du programme et qui

s'évalue partout à la même valeur ; voici un exemple :

```
I := J + K*L ;
J := K*L ;
```

où nous retrouvons 2 fois l'expression K*L. Dans ces cas, il convient donc de ne générer qu'une seule fois l'évaluation de l'expression, à l'endroit où elle est calculée pour la première fois ; partout ailleurs, on ne fera qu'utiliser le résultat de ce calcul. Dans notre exemple, l'optimisation revient à remplacer la séquence par :

```
Temp := K*L ;    /* Temp est une ressource temporaire */
I := J+Temp ;    /*    (un registre, par exemple)     */
J := Temp ;
```

En pratique, il est assez rare de trouver des sous-expressions communes écrites directement par le programmeur ; c'est surtout le compilateur qui en crée lorsqu'il fait l'expansion du programme pour générer le code objet. L'exemple classique est l'accès aux tableaux ; soit l'instruction :

```
T1[I] := T2[I] * 5 ;
```

Si l'on considère que le compilateur traduit l'accès à un tableau par :

orig virtuelle(tableau i) + I * taille élément(tableau i)

et que les éléments de T1 et de T2 sont du même type, la multiplication de I par la taille de l'élément est une sous-expression commune créée par le compilateur.

b) Affectations redondantes.

Une affectation redondante revient à affecter une deuxième fois la même valeur à une ressource (registre ou position mémoire), sans qu'elle ait été modifiée entre temps. Ces problèmes sont dus presque exclusivement à l'algorithme de génération de code employé qui n'est pas toujours capable d'identifier la dernière valeur chargée dans une ressource ; ils sont introduits surtout lorsque l'on charge une valeur dans un registre ou lorsque l'on range la valeur d'un registre. Prenons un exemple pour illustrer le problème ; soit les instructions :

```
I := J + K ;
J := I - 5 ;
```

un compilateur simple produirait pour cette séquence et pour un PDP-11 les instructions suivantes :

```
(1)  MOV K,R0
(2)  ADD J,R0
(3)  MOV R0,I
(4)  MOV I,R0
(5)  SUB #5,R0
(6)  MOV R0,J
```

Nous pouvons remarquer que la quatrième instruction est redondante car R0 contient contient déjà la valeur de I.

1.2. Relations créées par les branchements.

Les branchements sont une source importante d'opportunités d'optimisation dans un programme objet. Plusieurs circonstances y contribuent :

1) Lorsqu'un branchement en avant est généré, le compilateur n'a pas encore généré l'instruction cible, donc il ne peut pas disposer des informations sur celle-ci pour pouvoir faire les optimisations appropriées.
2) Si le branchement est en arrière, le problème contraire se pose : le code cible est déjà généré et il faudrait revenir en arrière pour l'optimiser, ce qui n'est pas toujours possible pendant la phase de génération. On est capable, par contre, de traiter de façon optimale le branchement lui-même.
3) Le code qui suit un branchement est généré très souvent depuis une partie éloignée (dans l'arbre du programme) de celle du branchement.

Pour cette raison, un nombre élevé d'optimisations sont fondées sur les relations créées par les branchements, dans le programme généré. Nous présentons ci-dessous celles qui nous ont paru les plus intéressantes :

a) Code inaccessible.

Si un branchement inconditionnel n'est pas suivi d'une étiquette, toutes les instructions qui le suivent jusqu'à la prochaine étiquette ne pourront jamais être exécutées ; ces instructions peuvent donc être supprimées.

Un compilateur non optimisé produit dans de nombreux cas du code inaccessible ; considérons, par exemple, l'instruction :

```
if <cond> then
begin
   <instr 1> ;
   ...
   <instr n> ;
   goto 1
end
else <instr n+1> ;
```

En général, les compilateurs génèrent après le "goto 1" une instruction de branchement inconditionnel pour sauter la partie "else" ; cette instruction sera inaccessible puisque elle est située derrière une instruction de branchement inconditionnel.

Des instructions inaccessibles aparaissent également au cours de la phase d'optimisations finales.

b) Séquences identiques à la jonction de branchements inconditionnels ("cross jumping").
Une étiquette est un point du programme où confluent plusieurs chemins d'exécution ; il peut arriver que les séquences d'instructions qui constituent ces chemins soient identiques, au moins en partie. Dans ce cas, on peut supprimer toutes les séquences identiques sauf celle qui précède l'étiquette ; toutes les séquences supprimées sont alors remplacées par un branchement au début de celle qui est conservée.

Cette opportunité est due particulièrement à la traduction des structures de contrôle telles que le "if-then-else" ou le "while".

Prenons par exemple le programme PASCAL et une de ses traductions possibles en Solar 16, qui sont présentés dans la figure IV.1. On peut remarquer que l'instruction "STA I" existe aussi bien avant l'étiquette ET1 qu'avant le "JMP ET1", c'est à dire sur les 2 chemins qui se rencontrent à l'étiquette ET1.

Si l'on applique cette optimisation, en supposant que ET1

a) Programme PASCAL

```
I := 2 ;
while I < LIM do
begin
   ...
   I := I+PAS
end ;
...
```

b) Traduction en Solar 16.

```
        LAI    2
        STA    I
ET1:    LA     I
        CP     LIM
        JG     ET2
        ...
        LA     PAS
        AD     I
        STA    I
        JMP    ET1
ET2:    ...
```

Figure IV.1 Un cas d'optimisation.

n'est référencée qu'une seule fois, on obtient le résultat montré dans la figure IV.2. Remarquer qu'une affectation redondante ("LA I") a été mise en évidence ; cette affectation était déjà redondante avant l'optimisation mais sa détection n'aurait pas pu être faite car l'étiquette ET1 définissait un nouveau bloc d'analyse.

```
        LAI    2
ET3:    STA    I
        LA     I
        CP     LIM
        JGE    ET2
        ...
        LA     PAS
        AD     I
        JMP    ET3
ET2:    ...
```

Figure IV.2 Résultat de l'optimisation.

c) Branchements à l'instruction suivante.

Ils peuvent être produits par un algorithme de génération trop simple ou bien apparaître comme conséquence d'une autre optimisation. Un exemple du premier cas est une traduction d'un "if" dans laquelle on ne vérifie jamais s'il y a une partie "else" ; en effet, dans ce cas, l'algorithme produit toujours un branchement vers l'instruction qui suit le "if", à la fin du traitement de la partie "then" ; s'il n'y a pas de "else", ce branchement pointe vers l'instruction suivante! On peut donc supprimer ce type de branchements.

d) Branchements en cascade.

Il s'agit de branchements dont la cible est un autre branchement ; on peut aussi avoir, bien entendu, toute une chaîne de branchements, ce que nous appelons une cascade. Cette chaîne ne peut pas être supprimée mais on peut la briser, remplaçant les étiquettes des branchements qui la composent par la cible finale de la chaîne ; de cette façon, on réduit le temps d'exécution, en supprimant le parcours inutile de la chaîne.

e) Branchement conditionnel suivi d'un branchement inconditionnel à la même étiquette.

Ce cas est provoqué, en général, par la suppression d'autres instructions. L'optimisation revient alors à supprimer le branchement conditionnel car il est redondant.

f) Instruction de comparaison ou de test inutile.

Quand une instruction de ce type n'est pas suivie d'un branchement conditionnel, elle n'a aucun interêt pour le programme et peut être supprimée. Toutefois, il y a un cas particulier : si la machine cible permet la conversion du code condition en valeur binaire, sans passer par des branchements (instruction Scc du Motorola 68000, par exemple), il faut aussi vérifier qu'une instruction de ce type ne suit pas le test.

g) Instructions autres que la comparaison ou les tests qui positionnent le code condition.

On peut profiter de ces instructions pour supprimer des tests ou des comparaisons ; en effet, si l'un d'eux est précédé par une instruction qui traite les mêmes données et qui positionne le code condition de la même façon, il devient redondant.

h) Branchement conditionnel à un autre branchement conditionnel mais de condition inverse.

Dans ce cas, on peut le faire pointer directement sur l'instruction qui suit le dernier branchement.

Il faut remarquer que les optimisations finales créent toujours de nouvelles opportunités d'optimisation. C'est ce qui justifie plusieurs des optimisations décrites car autrement il ne s'agirait que de cas théoriques.

1.3. Simplifications algébriques.

Certains algorithmes de génération de code produisent souvent des séquences d'instructions qui correspondent à des identités algébriques (<Aho79>). Dans ce cas, on peut profiter des propriétés de ces identités pour optimiser le code généré ; ainsi, si elles n'ont aucun effet de bord, on peut les supprimer (Figure IV.3a), sinon on peut les remplacer par des séquences équivalentes mais plus simples (Figure IV.3b).

```
I := I + 0 ;
I := I - 0 ;
I := I * 1 ;
I := I / 1 ;
```

a) Séquences à éliminer.

```
exp1 * 0              => 0
exp1 * 2              => exp1 + exp1
false and exp1        => false
exp1 or exp1          => exp1
not (exp1 = exp2)     => exp1 <> exp2
- (- exp1)            => exp1
```

b) Séquences à simplifier

Figure IV.3 Exemples de simplification algébrique.

On peut aussi faire d'autres simplifications basées sur la représentation de données utilisée par la machine traitée ; en effet, certaines instructions permettent de faire des opérations différentes de leurs objectifs, en profitant de la représentation employée. L'exemple le plus connu est celui des décalages qui permettent de faire plus efficacement des multiplications ou des divisions par une puissance de 2 ; un autre exemple en est la division flottante par une constante qui peut être remplacée par une multiplication, si l'on utilise l'inverse de la constante.

Finalement, on peut aussi inclure dans cette classe la propagation de constantes. Elle consiste à remplacer toutes les utilisations d'une variable, qui s'est vu affecter une constante, par celle-ci jusqu'à la prochaine affectation de la

variable.

1.4. Instructions et modes d'adressage plus efficaces.

Normalement, les machines ont certains modes d'adressage et instructions très efficaces pour réaliser des opérations courantes mais dans des conditions très particulières. Parmi les premiers, nous trouvons principalement les modes qui incrémentent ou décrémentent automatiquement un registre (de base ou d'index) ; en ce qui concerne les instructions, la liste est assez longue et dépend de la machine, bien que l'on retrouve des instructions semblables dans plusieurs machines : affectation de zéro (CLR du PDP-11, STZ du SOLAR-16), incrémenter de 1 une ressource (INC du PDP-11, IC du SOLAR-16), etc.

Il est assez difficile d'intégrer ce type de modes d'adressage et d'instructions dans la structure générale d'un compilateur ; en effet, la reconnaissance des cas où ils peuvent être applicables sont autant de cas particuliers à prévoir. En général, les compilateurs préfèrent ne pas les utiliser et générer d'autres séquences équivalentes même si elles sont plus chères.

Code généré	Code optimisé	Code généré	Code optimisé
ADD #177776,R1 CLR @R1	CLR -(R1)	LAI 1 AD VAR STA VAR	IC VAR
ADD #2,R1	TST (R1)+		
ADD #1,I	INC I	CPI 0 JE ETIQ	JAE ETIQ
MOV #0,I	CLR I	LAI 0 CPR A,B	CPZR B
a) PDP-11		b) Solar 16	

Figure IV.4 Utilisation d'opérations efficaces de la machine.

Une fois le code généré la tâche devient plus simple. Il faut d'abord déterminer statiquement quelles sont les séquences "optimisables" produites par le compilateur ; le travail de l'optimiseur consiste alors à trouver ces séquences et à les

remplacer par les modes d'adressage ou instructions équivalents mais moins chers.

Dans la figure IV.4, on peut voir des séquences de code qui réalisent les mêmes fonctions à des coûts très différents, pour 2 machines réelles (‹DEC75›, ‹Sems78›). Ces séquences correspondent au code produit par un générateur simple et au code qui serait obtenu après optimisation.

2. Optimisations réalisées par GEMME.

Il est peu réaliste de vouloir rassembler toutes ces techniques dans un même optimiseur car le coût d'exécution risque d'être trop élevé par rapport au gain obtenu ; en plus, certaines techniques sont très liées à une machine ou à une famille de machines. Il est donc plus intéressant de choisir les techniques les plus adaptées à chaque contexte ; c'est ce que nous avons fait pour notre optimiseur final.

Notre critère primordial de choix a été l'indépendance de la machine cible car c'est l'objectif principal de tout notre système ; d'autres travaux ont suivi cette même direction (‹Fraser79›, ‹Davidson82›), bien qu'avec des choix très différents.

Notre deuxième critère de choix concerne l'efficacité, certaines optimisations pouvant demander beaucoup de calculs pour des résultats peu intéressants. De même, nous restreignons notre analyse aux instructions très proches ; notre "fenêtre" de travail ("peephole") est composée au maximum d'un bloc d'instructions limité par 2 étiquettes ou branchements ; nous voulons ainsi réduire le nombre de combinaisons à analyser.

D'autre part, en analysant les résultats des expériences réalisées dans ce domaine, nous avons pu constater que :

1) les améliorations les plus intéressantes sont liées à l'analyse des branchements (‹Wulf75›),
2) les optimisations liées au choix des instructions machine ou

des modes d'adressage peuvent être faites à la génération de code,

3) certaines optimisations peuvent être réalisées dans une phase d'optimisations globales, où leur détection est plus simple (suppression des sous-expressions communes et simplifications algébriques).

En conséquence, nous avons basé notre choix sur ces critères et constatations.

Tout d'abord, nous avons essayé de modéliser de façon indépendante de la machine les optimisations basées sur les relations créées par les branchements ; nous sommes arrivés à le faire pour les cas suivants :

a) Suppression des instructions inaccessibles.
b) Factorisation des séquences identiques qui précedent une jonction de branchements inconditionnels.
c) Suppression des branchements adressant l'instruction suivante.
d) Rupture des branchements en cascade.
e) Suppression des branchements conditionnels suivis d'un autre inconditionnel, tous deux adressant la même étiquette.
f) Inversion de la condition de certains branchements.

En ce qui concerne la deuxième constatation, notre générateur de code choisit toujours les modes d'adressage et les instructions les plus appropriés à l'opération traitée. Par conséquent, des optimisations telles que le remplacement d'instructions ou de modes d'adressage par leurs équivalents moins chers ne sont pas nécessaires dans notre cas ; il faut signaler que l'optimiseur de Fraser et Davidson (‹Davidson80›), qui est indépendant de la machine cible, s'occupe principalement de ce type d'optimisations.

Enfin, nous avons retenu une dernière optimisation qui donne de bons résultats et dont la mise en oeuvre peut être indépendante de la machine. Il s'agit de la suppression d'affectations redondantes.

DEUXIEME PARTIE

V. DESCRIPTION DES MACHINES.

Dans le système GEMME, la construction d'un nouveau générateur de code est réalisée automatiquement à partir d'une description de la machine cible. Cette description constitue une spécification de la structure et du jeu d'instructions de la machine ; elle ne contient toutefois que les informations nécessaires à la génération de code.

Nous présentons dans ce chapitre le formalisme utilisé pour décrire une machine. Nous discutons d'abord des concepts de base du modèle de description que nous avons choisi. Nous proposons ensuite un langage de description basé sur ce modèle.

1. Modèle de description.

Contrairement au modèle de description de PQCC qui vise une classe d'applications plus vaste (‹Cattell79›), nous utilisons un modèle spécialisé dans la construction de générateurs de code. De ce fait, notre modèle est plus simple, principalement en ce qui concerne la description des instructions. Toutefois, les concepts de base utilisés sont les mêmes, d'où une certaine similitude entre les deux modèles.

Nous présentons dans ce qui suit les différents aspects de notre modèle.

1.1. Type de données.

Nous utilisons la notion de type pour spécifier deux sortes d'informations :
- les types d'objets manipulables directement par la machine,
- la représentation des types de base du langage PASCAL.

Un type de la machine est défini par une longueur exprimée en bits et par une contrainte d'alignement mémoire. La représentation d'un type PASCAL est définie en lui associant un type de la machine ainsi qu'un domaine de valeurs et une fonction de codage capable de construire la représentation binaire de toute valeur de ce domaine. La fonction de codage doit être choisie parmi un ensemble de fonctions prédéfinies du modèle ; ainsi, par exemple, on peut choisir pour les entiers le complément à deux, le complément à un ou la convention valeur absolue et signe.

Un type doit être associé à tout opérande et à toute opération définis avec le modèle. Ce type peut être indifféremment un type de la machine ou un type de base PASCAL.

1.2. <u>Ressource.</u>

Cette notion nous permet de spécifier les ressources de mémorisation disponibles dans la machine. Etant donné que le système GEMME s'intéresse uniquement aux machines à registres, seules deux classes de ressources sont à décrire : la mémoire principale et les registres (au sens large : accumulateurs mais également compteur ordinal, code de condition, etc.).

La mémoire principale est considérée comme un ensemble d'éléments ordonnés, ayant tous le même type ; elle est par conséquent définie par le type et le nombre (sa taille) de ses éléments.

En ce qui concerne les registres, nous avons adopté une structure de description hiérarchisée pour pouvoir traiter leurs problèmes de superposition. Le sommet de cette hiérarchie est constitué par les registres qui ne sont contenus dans aucun registre ; ils sont définis par leur taille en bits et par leur type. Tout autre registre (sous-registre) est défini en fonction du registre qui le contient (son sur-registre) ; sa définition est par conséquent constituée par un lien vers son sur-registre, sa position dans celui-ci et son type.

1.3. Modèle de descripteur.

Les modes d'adressage de la machine sont spécifiés en utilisant la notion de modèle de descripteur (cf III.3.1.3). Cette notion nous permet de décrire la plupart des modes d'adressage existants, exception faite des modes dans lesquels le calcul de l'adresse est suivi d'un effet secondaire (par exemple, la modification de la base dans l'adressage basé post-incrémenté).

Les modèles de constantes et d'intervalles permettent de définir les constantes implicites et immédiates. Les modèles de registres sont utilisés pour définir les classes de registres. Les modèles de mémoire correspondent aux modes d'adressage mémoire.

Une définition de mode d'adressage est constituée par :
- le modèle qui le décrit,
- le type des données adressées,
- le coût de l'adressage,
- une valeur qui permet de l'identifier dans le code généré.

1.4. Classe de modèles.

La correspondance entre les modes d'adressage et les opérandes des instructions de la machine est spécifiée en utilisant la notion de classe de modèles ; plus précisément, une classe de modèles définit l'ensemble des modes d'adressage pouvant être utilisés pour accéder à un opérande donné d'une instruction. Par exemple, une instruction d'addition peut demander un registre général comme opérande gauche et permettre une constante immédiate ou bien un registre général comme opérande droit ; dans ce cas, deux classes de modèles sont à définir : le premier sera uniquement formé par le modèle de registres généraux tandis que le second sera formé par ce même modèle ainsi que celui des constantes immédiates.

Une classe de modèles est définie par la liste des modèles qui la composent ; par ailleurs, un format de génération (cf

V.1.5) est associé à chaque élément de la liste.

La notion de classe de modèles a été introduite dans le but de réduire le nombre d'instructions à spécifier ; ainsi, par exemple, l'instruction MOVE du Motorola 68000 peut être écrite de 144 façons différentes car elle accepte 12 modes d'adressage pour chaque opérande. Dans certains cas, cette notion est par conséquent indispensable pour maintenir la taille de la description dans des limites raisonnables.

1.5. Formats de génération.

Ces formats spécifient la représentation binaire des instructions et des opérandes de la machine. Un format est composé d'une liste de champs constitués par :

a) Le mot de l'instruction dans lequel apparaît le champ ; ce champ permet la description des instructions de longueur variable.
b) Les bits initial et final du champ dans le mot.
c) Le type du champ : code opération, mode d'adressage ou constante immédiate.

Nous distinguons deux types de format de génération : l'un pour les opérandes et l'autre pour les instructions. Le premier est constitué uniquement de champs mode d'adressage et constante immédiate ; il définit la représentation binaire d'une classe de modèle de descripteur. Le second est composé d'un ensemble de champs de type code opération et de formats d'opérande ; il définit la représentation binaire d'une classe d'instructions.

A la génération de code, les informations générées sont organisées et traduites en binaire selon le format de l'instruction correspondante.

1.6. Instructions.

La notion d'instructions est le centre de notre modèle de description. Elle permet la spécification des opérations disponibles dans la machine et leur correspondance avec les

opérations du langage Pascal.

Une instruction de la machine est considérée dans notre système de génération comme une possibilité de réalisation (cf III.3.1.6) d'une opération Pascal donnée. Le modèle de description le plus simple consiste par conséquent à définir pour chaque opération Pascal l'ensemble des opérations pouvant être utilisées dans sa traduction ; une instruction est définie par les informations suivantes :

- la classe de modèles à utiliser pour l'opérande gauche,
- la classe de modèles à utiliser pour l'opérande droit,
- l'endroit où sera obtenu le résultat,
- le coût de l'instruction,
- un code opération mnémonique,
- un format de génération,
- une liste de valeurs à générer correspondant aux champs code opération du format (par exemple : code_op=5, lg_opérandes=1, etc.).

Une possibilité d'opération correspond en réalité à une séquence d'instructions de la machine ; ainsi, si la machine ne dispose d'aucune instruction pour réaliser directement une opération, on pourra décrire une séquence d'instructions qui la met en oeuvre. Cette possibilité est également très utile pour spécifier certaines fonctions prédéfinies du langage (abs, odd, etc.).

Le résultat d'une opération est en général obtenu dans l'un des opérandes. Cependant, certaines instructions produisent un résultat qui n'occupe qu'une partie de l'opérande (la division) ou bien l'opérande tout entier plus d'autres ressources (la multiplication). La description de ces cas particuliers est effectuée dans notre modèle à l'aide d'un certain nombre de fonctions prédéfinies : sous-registre, sur-registre (cf V.1.2), poids faibles ou poids forts d'une position mémoire, etc.

Nous distinguons plusieurs types d'instructions :

a) Arithmétiques et logiques, utilisées pour l'évaluation des expressions arithmétiques et booléennes ainsi que pour les

calculs d'adresse.

b) De transfert de données, utilisées pour mettre en oeuvre les affectations des variables du programme mais également les affectations introduites par le générateur (le chargement de registres, par exemple).

c) De contrôle, constituées par les comparaisons et les branchements.

d) De manipulation de bits, utilisées pour l'évaluation des expressions ensemblistes.

e) D'appel et de retour de procédure.

f) Spéciales, utilisées pour l'optimisation du code généré :
   - les décalages arithmétiques,
   - les affectations de zones mémoire,
   - la transformation du code de condition en valeur booléenne (SCC du Motorola 68000, par exemple),

   etc.

Ces différents types d'instructions sont définis séparément dans notre modèle.

1.7. <u>Quelques remarques sur le modèle.</u>

Après la présentation du modèle, nous sommes en mesure de souligner quelques caractéristiques importantes de celui-ci :

a) La description d'une machine ne comprend pas nécessairement tout le jeu d'instructions de celle-ci ; elle décrit uniquement les instructions pouvant être utilisées pour traduire les opérations Pascal.

b) Une description ne dit jamais comment générer le code pour la machine cible. Elle se limite à spécifier la correspondance entre les instructions de la machine et les opérations Pascal.

c) Notre modèle est purement descriptif ; il est donc assez différent du modèle procédural utilisé par la plupart des langages de description de machines : ISPS (<Barbacci78>), DDL (<Dietmeyer74>), CONLAN (<Piloty80>), etc.

d) Le modèle est restreint aux machines à registres classiques ; il ne permet pas de décrire des machines à pile ou des machines spécialisées (vectorielles, matricielles, etc.). Par ailleurs, il est limité dans cette classe aux machines à deux

adresses ; il suppose en effet que le résultat d'une instruction est obtenu dans l'un des opérandes ou bien dans une ressource explicite (code de condition, accumulateur, etc.) ; de ce fait, on ne peut pas associer au résultat une classe de modèles propre. Toutefois, ces limitations sont relativement peu importantes car la classe de machines traitée recouvre la plupart des machines disponibles sur le marché.

2. Le langage de description.

Le Langage de Description des machines (LD) utilisé par le système GEMME est fondé sur le modèle présenté ci-devant. Nos contraintes de temps nous ont toutefois amené à réaliser une version simplifiée de ce modèle ; en particulier, nous avons supprimé la possibilité d'utiliser une séquence d'instructions pour définir une possibilité d'opération (cf V.1.6), bien que les générateurs GEMME soient capables de l'exploiter. La définition du langage LD n'a par ailleurs pas été très élaborée en raison de ces mêmes contraintes ; elle constitue en effet une version provisoire mais suffisante pour les besoins de cette présentation.

La structure du langage LD est proche de celle des manuels de référence des constructeurs. Une description est organisée en rubriques hiérarchisées, définissant des éléments de même nature (types, ressources, modèles de descripteur, etc.) ; ces éléments sont définis par énumération de leurs propriétés. Un certain nombre de facilités ont été prévues pour simplifier la tâche d'écriture d'une description.

Nous présentons ci-après un exemple de description de machine. La définition exacte du langage est donnée dans l'annexe A ; de nombreux commentaires ont été introduits pour expliquer la sémantique des constructions définies. Dans les annexes B et C, nous donnons des descriptions plus complètes pour deux machines (Motorola 68000 et PDP-11).

3. Exemple de description.

Ce paragraphe est consacré à un exemple de description de machine, la machine choisie étant un Motorola 68000. Cet exemple a une fonction purement explicative et est donc simplifié par rapport à une véritable description. Dans la description, tout ce qui figure entre 2 caractères "%" est un commentaire et explique les lignes qui précèdent.

```
TYPES;
   MACHINE;
      Octet : 1,1;
      DemiMot : 2,2;
      Mot : 4,2;

      % On décrit ici tous les objets manipulables par la
      machine. Pour chaque objet, on indique :
      - un nom (choisi au gré de l'utilisateur),
      - une taille (exprimée en une unité arbitraire mais
        identique pour tous les objets),
      - une contrainte d'alignement définie de la manière
        suivante : si un objet possède une contrainte
        d'alignement C, A est une adresse valide pour lui si le
        reste de la division de A par C est nul.
      On voit ici que l'utilisateur a choisi comme unité de
      taille la taille d'un octet ; la ligne :
         Mot : 4,2;
      s'interprète par conséquent de la manière suivante : la
      machine manipule des objets de longueur 4 (octets), que
      l'on appelle Mot, et qui ont comme contrainte d'alignement
      2. %

   PASCAL;
      integer : DemiMot;
      boolean : Octet;
      char : Octet;
      real : Mot;
      pointer : Mot;
```

% Dans cette rubrique, l'utilisateur indique la correspondance qu'il choisit entre les types de base du langage PASCAL et les objets manipulables par la machine. Les mots integer, boolean, etc. sont des mots clefs du langage de description. En partie droite, se trouvent les noms définis dans la rubrique précédente. %

```
REGISTRES;
  REFREG;
    % description des registres "sommet de hiérarchie" %
    dl[0,7] : Mot;
    al[0,7] : Mot;
```

% Chacune de ces lignes est un raccourci d'écriture. En particulier, la ligne

```
dl[0,7] : Mot;
```

est un raccourci pour

```
dl0 : Mot;
dl1 : Mot;
...
dl7 : Mot;
```

Chaque ligne

```
dln : Mot;
```

s'interprétant comme la déclaration d'un registre de nom dln, comportant des bits dont les positions vont de 0 à 31. %

```
SOUSREG;
  dwh[0,7] : dl[0,7] (0,DemiMot);
  dwl[0,7] : dl[0,7] (16,DemiMot);
  aw[0,7] : al[0,7] (16,DemiMot);
```

% Là aussi chacune des lignes est un raccourci d'écriture. Par exemple, la ligne

```
dwl[0,7] : dl[0,7] (16,DemiMot)
```

est une abbréviation pour :

```
dwl0 : dl0 (16,DemiMot);
dwl1 : dl1 (16,DemiMot);
```

```
...
   dwl7 : dl7 (16,DemiMot);
```

Une ligne quelconque

```
   dwln : dln (16,DemiMot);
```

est interprétée comme la déclaration d'un sous-registre dwln du registre dln ; ce sous-registre est constitué par les bits 16 à 31 du registre dln. %

MODELES;

% Les modèles de descripteur nécéssaires à la description des instructions %

DEPLACEMENTS;

```
Depl8 : Octet, (-128,127);
Depl16 : DemiMot, (-32768,32767);
```

% Depl8 et Depl16 sont des noms donnés à l'ensemble des valeurs se trouvant respectivement dans les intervalles (-128,127) et (-32768,32767). Ces valeurs sont contenues dans les ressources de la machine que l'on a nommées respectivement Octet et DemiMot %

CLASSREG; % les modèles de registres %

```
ClasseRegDW : DemiMot,dwl0,dwl1,dwl2,dwl3;
ClasseRegAL : Mot,al0,al1,al2,al3;
ClasseRegAW : DemiMot,aw0,aw1,aw2,aw3;
```

% Nous définissons ici 3 ensembles de registres et donnons un nom à ces ensembles. Par exemple, la ligne

```
   ClasseRegDW : DemiMot,dwl0,dwl1,dwl2,dwl3;
```

donne le nom ClasseRegDW à l'ensemble des registres dwl0,dwl1,dwl2,dwl3, tout en déclarant que ces registres contiendront un DemiMot. %

CLASSMEM; % les modèles de mémoire (modes d'adressage) %

```
AdrDirect : DemiMot, Depl16, ClasseRegAL, nil;
AdrIndexe : DemiMot, Depl8, ClasseRegAL, ClasseRegAl,
ClasseRegAW;
```

% Ici, nous définissons les adressages direct et indexé à l'aide de modèles de descripteur. Rappelons qu'un tel descripteur a la forme :

```
                 ------------
Type             !   'mem   !
                 ------------
Type_Valeur      !          !     type de la valeur décrite
                 ------------
Depl             !          !     descripteur du déplacement
                 ------------
Base             !          !     descripteur de la base
                 ------------
Index            !          !     descripteur de l'index
                 ------------
```

L'adresse mémoire définie par un tel descripteur a pour valeur :

DeplD + BaseD + IndexD

Nous voyons que la définition des adressages se fait à l'aide de 4 valeurs qui vont remplir respectivement les champs Type_Valeur, Déplacement, Base et Index. "nil" est un mot réservé du langage de description signifiant : "absence de valeur".

Prenons le cas de l'adressage direct :

AdrDirect : DemiMot,Depl16,ClasseRegAL,nil;

Si l'on se réfère aux définitions qui ont déjà été faites de Depl16 et de ClasseRegAL, on voit que le descripteur ainsi défini est :

```
                 ------------
Type             !   'mem   !
                 ------------
Type_Valeur      ! DemiMot  !
                 ------------
Depl             !     .----!-----> (-32768,32767)
                 ------------
Base             !     .----!-----> (al0,al1,al2,al3)
                 ------------
Index            !   nil    !
                 ------------     %
```

POSSADR;

% Description de ce que nous avons appelé classe de modèle ou "possibilité d'adressage" (Pa en abrégé). Une Pa est l'ensemble :

- d'un modèle de descripteur (décrivant l'adressage

proprement dit),
- d'un mode (une valeur qui caractérise le mode d'adressage),
- d'un coût (un couple de 2 valeurs caractérisant le coût de ce mode d'adressage). %

```
PaRegDW : ClasseRegDW, Mode=0, Cout=(0,0);
PaRegAW : ClasseRegAW, Mode=1, Cout=(0,0);
PaImm16 : Dep116, Mode=(7,4), Cout=(1,2);
PaAdrDirect : AdrDirect, Mode=5, Cout=(1,4);
PaAdrIndexe : AdrIndexe, Mode=(6,1), Cout=(1,5);
```

% On voit que pour décrire ces Pa, l'utilisateur a choisi comme fonction de coût le couple :
(encombrement mémoire,temps d'exécution) %

CLASSPA;

% Dans cette rubrique, on donne la possibilité de définir des ensembles de Pa et de leur donner un nom %

```
EAW : PaRegW,PaRegAW,PaImm16,PaAdrDirect,PaAdrIndexe;
```

% Cette ligne décrit ce qui dans le manuel Motorola est appelé "Effective Address" %

```
AltMemW : PaAdrDirect,PaAdrIndexe;
```

% l' "Alterable memory" du manuel %

```
DRGW : PaRegDW;
ARGW : PaRegAW;
CM16 : PaImm16;
DARGW : PaRegDW,PaRegAW;
```

% l'ensemble des registres de données et d'adresse %

INSTRUCTIONS;

% Dans cette rubrique, on indique pour chaque opérateur PASCAL la manière de le réaliser par les opérateurs de la machine. Les noms d'opérateur Plus, Moins, etc. sont des mots réservés du langage de description %

```
OPERATEURS: Plus;
   Plus1 : DRGW,   EAW,     Cout=(2,2),   Result=('implicite,'g),
           Taille=2, Codop='ADD;
   Plus2 : DRGW, AltMemW, Cout=(2,4), Result=('implicite,'d),
           Taille=2, Codop='ADD;
   Plus3 : ARGW,   EAW,     Cout=(2,4),   Result=('implicite,'g),
           Taille=2, Codop='ADDA;
   Plus4 : DARGW,  CM16,    Cout=(4,4),   Result=('implicite,'g),
           Taille=2, Codop='ADDI;
```

% Une opération est décrite par un ensemble de "possibilités d'opération" (Po en abbrégé).
Une Po est formée de l'ensemble suivant :
- Pa de l'opérande gauche.
- Pa de l'opérande droit.
- Coût de l'instruction.
- Indication de l'endroit où se trouve le résultat. ('implicite,'g) signifie que le résultat se trouve dans l'opérande gauche.
- Taille de l'instruction (nécessaire pour résoudre les branchements).
- Code opération.

On voit que l'on a décrit les instructions ADD,ADDA et ADDI. ADD a été découpé en 2 Po selon que le résultat de l'opération se trouve dans l'opérande gauche ou droit. %

...

Une véritable description est suivie par la description des Po des opérations Moins, Mult, Div, etc. On trouvera en annexe B une description plus complète pour le Motorola 68000.

VI. L'ALLOCATEUR DES VARIABLES.

L'allocateur des variables est le composant du codeur spécialisé dans la compilation des déclarations du programme ; il est chargé de trouver une représentation physique pour chaque type défini par l'utilisateur ainsi que d'associer une adresse mémoire et un descripteur de valeur à chaque variable du programme.

Nous présentons dans ce qui suit la tâche réalisée par l'allocateur. D'abord, nous analysons la structure d'un programme Pascal, de façon à mettre en relief les règles de visibilité et de durée de vie définies en Pascal ; ensuite, nous proposons en fonction de cette structure une organisation et une gestion de la mémoire utilisée par le programme durant son exécution. Puis, nous décrivons la façon de compiler les types définis par l'utilisateur ; pour terminer, nous présentons la méthode d'allocation de mémoire que nous utilisons pour les variables du programme.

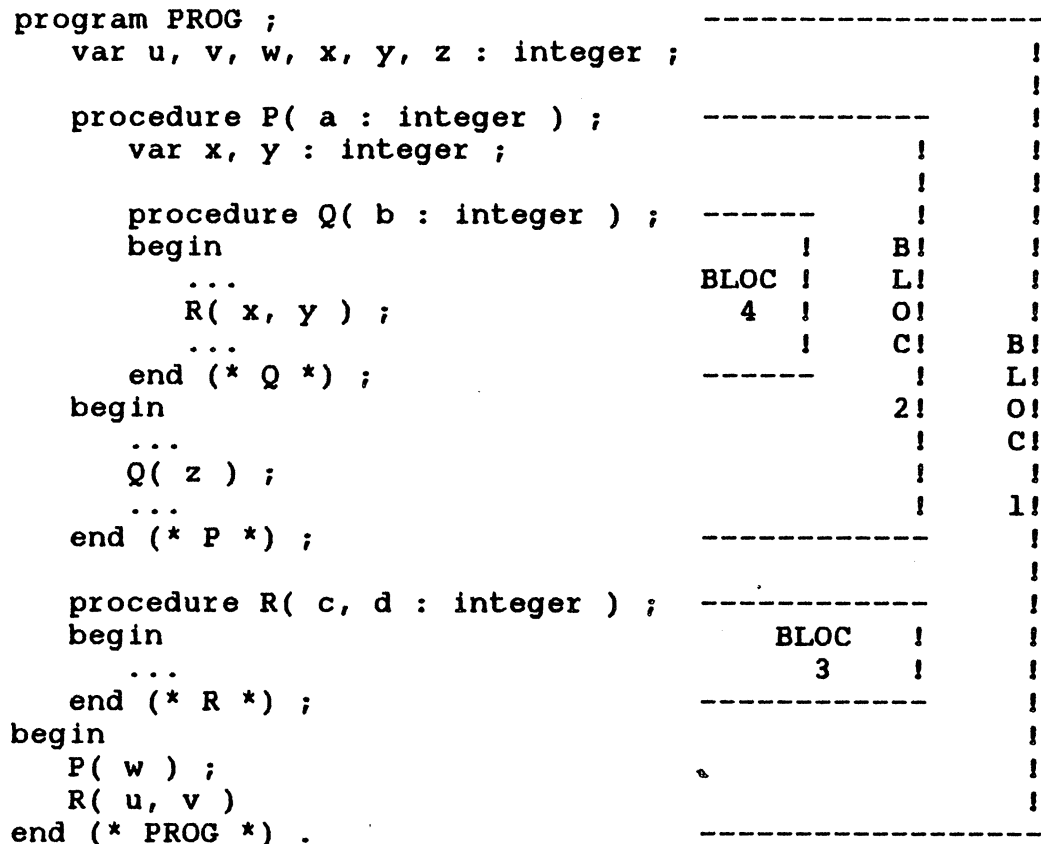
1. Structure d'un programme Pascal.

Pascal est un langage à structure de blocs, où la notion de bloc s'applique exclusivement aux procédures. Cette structure permet de définir des procédures imbriquées et de déclarer des objets à l'intérieur de ces procédures ; elle permet également d'associer un même identificateur à des objets déclarés dans des procédures différentes, même si elles sont imbriquées. La durée de vie et la visibilité des identificateurs sont définies par 2 règles, que nous empruntons à Aho et Ullman (‹Aho77›) :

a) Un objet déclaré dans un bloc B n'est utilisable qu'à l'intérieur de ce bloc.
b) Si un bloc B' est imbriqué dans B, tout objet utilisable dans B est aussi utilisable dans B', à moins que l'identificateur

de cet objet n'ait été redéclaré dans B'.

D'après ces règles, il est possible d'utiliser à l'intérieur d'un bloc tous les identificateurs définis dans ce bloc et dans ceux qui le contiennent ; l'utilisation d'un identificateur est toujours une référence à la déclaration la plus interne (dans l'ordre d'imbrication) de cet identificateur.



```
program PROG ;
   var u, v, w, x, y, z : integer ;

   procedure P( a : integer ) ;
      var x, y : integer ;

      procedure Q( b : integer ) ;
      begin
         ...
         R( x, y ) ;
         ...
      end (* Q *) ;
   begin
      ...
      Q( z ) ;
      ...
   end (* P *) ;

   procedure R( c, d : integer ) ;
   begin
      ...
   end (* R *) ;
begin
   P( w ) ;
   R( u, v )
end (* PROG *) .
```


Figure VI.1. Structure d'un programme Pascal.

Le programme de la figure VI.1 illustre la structure d'un programme Pascal ; on peut remarquer que :

a) Le programme PROG constitue un premier bloc ; ce bloc n'a accès qu'aux objets déclarés par lui-même (variables u, v, ..., z et procédures P et R).

b) Les procédures P et R définissent 2 autres blocs contenus dans celui du programme principal mais disjoints ; ces 2 procédures peuvent utiliser les objets définis dans le programme principal (par exemple P utilise la variable z) et dans leur bloc (P utilise ainsi la procédure Q) ; elles peuvent aussi redéfinir des identificateurs déclarés dans le

programme principal, cachant ainsi les objets correspondants (par exemple les variables x et y de la procédure P).

c) La procédure Q définit un dernier bloc, imbriqué dans celui de la procédure P.

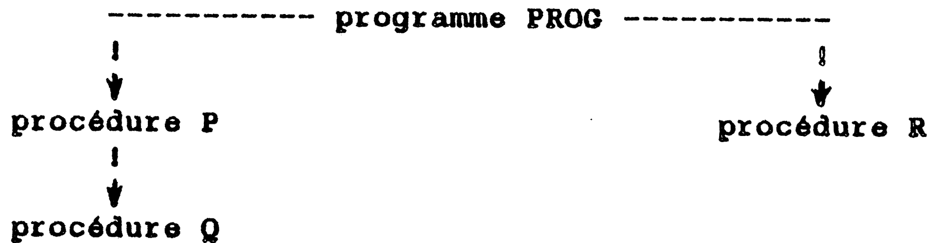


Figure VI.2. Structure statique d'un programme Pascal.

Les règles de visibilité définies donnent lieu à une structure arborescente de l'imbrication des procédures (voir figure VI.2) ; dans cet arbre, chaque noeud représente une procédure et chaque lien une relation d'imbrication. Il s'agit de la structure statique d'un programme ; par association, on appelle niveau statique d'une procédure le niveau du noeud de l'arbre qui représente la procédure.

2. Gestion de la mémoire pour un programme Pascal.

Un programme Pascal nécessite une gestion dynamique de la mémoire qui sera allouée à ses données.

D'une part, les données déclarées statiquement peuvent exister en plusieurs exemplaires, en raison de la recursivité ; en effet, à un instant donné, une variable locale à une procédure appelée recursivement a autant de valeurs que d'appels de la procédure encore actifs. Ce type de données peut être géré à l'aide d'une pile (‹Aho77›, ‹Gries71›), que nous appelerons la pile d'exécution du programme.

D'autre part, des données peuvent être créées et détruites dynamiquement, au gré de l'utilisateur ; ces données seront placées dans ce que l'on appele le tas et gérées de façon appropriée.

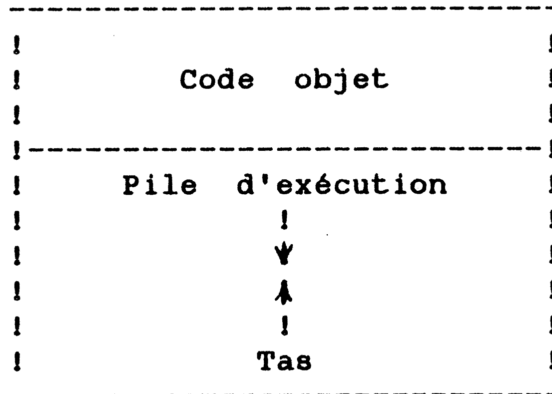


Figure VI.3. Partage de l'espace mémoire du programme.

La pile d'exécution et le tas se partagent ce qui reste de l'espace mémoire du programme, après le chargement du code objet. La figure VI.3 donne un exemple de ce partage ; dans cet exemple, la pile occupe les adresses basses et croît vers les adresses hautes ; le schéma contraire peut également être utilisé en fonction de la machine cible.

Dans ce qui suit, nous présentons les choix que nous avons faits en ce qui concerne la pile d'exécution et le tas.

2.1. Gestion de la pile d'exécution.

C'est dans cette pile que nous allouons l'espace mémoire nécessaire aux objets locaux des procédures actives. Nous présentons ci-après la technique d'adressage utilisée pour accéder à ces objets ; nous décrivons ensuite l'organisation du bloc d'informations associé à toute procédure (environ de procédure).

2.1.1. Adressage des informations de la pile.

Les adresses des informations locales d'une procédure sont exprimées comme des déplacements relatifs au début de leurs environs respectifs ; ces déplacements sont fixés au moment de l'allocation des variables et sont, en conséquence, connus dès la compilation.

Par contre, les adresses des environs varient en fonction des

appels de procédure et doivent par conséquent être gérées dynamiquement. Nous utilisons pour ce faire deux types de liens :

a) Dynamiques, correspondant à l'enchaînement des appels de procédure ; ils sont en nombre de deux :
   - lien vers l'environ de la procédure appelante,
   - pointeur sommet de pile.

b) Statiques, donnant accès aux environs globaux visibles par la procédure. Nous avons choisi d'utiliser la technique du display (‹Gries65›) pour mettre en oeuvre ces liens ; un display est un vecteur dont la ième entrée contient l'adresse de l'environ de niveau statique i activé le plus récemment.

Tous ces liens pointent en général vers le début de l'environ correspondant ; toutefois, si l'adressage basé de la machine cible admet des déplacements négatifs, les liens seront chargés avec une adresse permettant d'exploiter ces déplacements.

Nous avons choisi de garder une copie du display dans chaque environ ; l'accès à une information globale se fait donc par indirection à travers l'entrée correspondante de cette copie. En raison du coût de l'indirection, nous chargeons une partie du display dans des registres de base, ce qui nous permet d'utiliser l'adressage basé pour effectuer ces accès. La description de la machine doit contenir une liste des registres utilisables à cet effet (registres display) ; plus elle sera grande, plus d'informations globales seront accédées via l'adressage basé ; pour des raisons d'efficacité, notre système impose qu'au moins un registre compose cette liste.

Les registres display sont employés pour adresser les environs <u>les plus utilisés</u> ; ainsi :

a) Le premier registre contient l'adresse de l'environ de la procédure en cours d'exécution.

b) Le deuxième registre contient l'adresse de l'environ du programme principal.

c) Les autres registres pointent vers les environs de niveau statique immédiatement inférieurs à celui de l'environ courant. Si celui-ci a le niveau i, ils pointent donc vers les environs de niveau i-1, i-2, ... jusqu'à ce que tous les

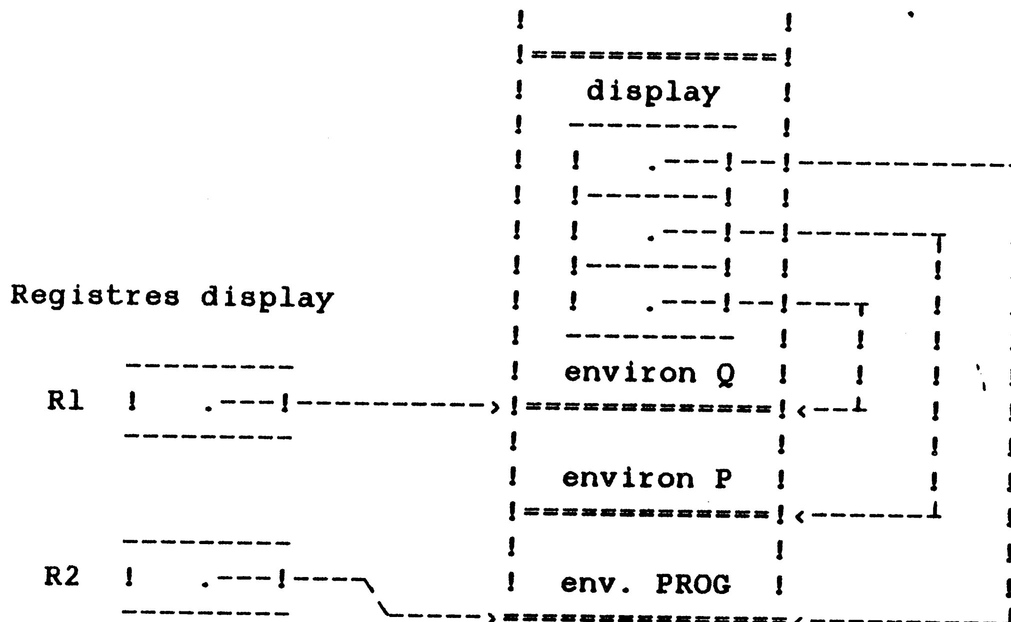


Figure VI.4. Utilisation des registres display.

registres display soient occupés, ou bien, tous les environs pointés.

A un instant donné, s'il y a plus d'environs à adresser que des registres display, l'adressage indirect est utilisé pour le reste d'environs. La figure VI.4 donne un exemple de cette méthode pour la chaîne d'appels (voir figure VI.1) :

PROG --> P(w) --> Q(z)

Il faut indiquer que les registres display sont réservés en exclusivité à l'adressage des environs ; ils ne seront donc jamais utilisés par le générateur de code pour d'autres usages, même s'ils sont libres.

Avec la méthode décrite, l'adressage des environs est facilement adaptable au nombre de registres de base de la machine cible ; il est possible, en particulier, de déterminer par expériences successives le meilleur compromis entre le nombre de registres réservés à cet usage et ceux qui peuvent être utilisés ailleurs. D'autre part, cette méthode n'impose aucune limite au degré d'imbrication des procédures.

2.1.2. Format d'un environ de procédure.

Un environ de procédure contient 2 types d'informations :

a) Les objets locaux à la procédure ; il s'agit de ses variables locales mais aussi des temporaires et des relais

d'indirection (pour l'accès aux données structurées) dont la procédure a besoin.

b) Les informations nécessaires à la gestion de l'appel (adresse de retour, paramètres) ainsi qu'à l'adressage des objets utilisables par la procédure (liens dynamiques et display).

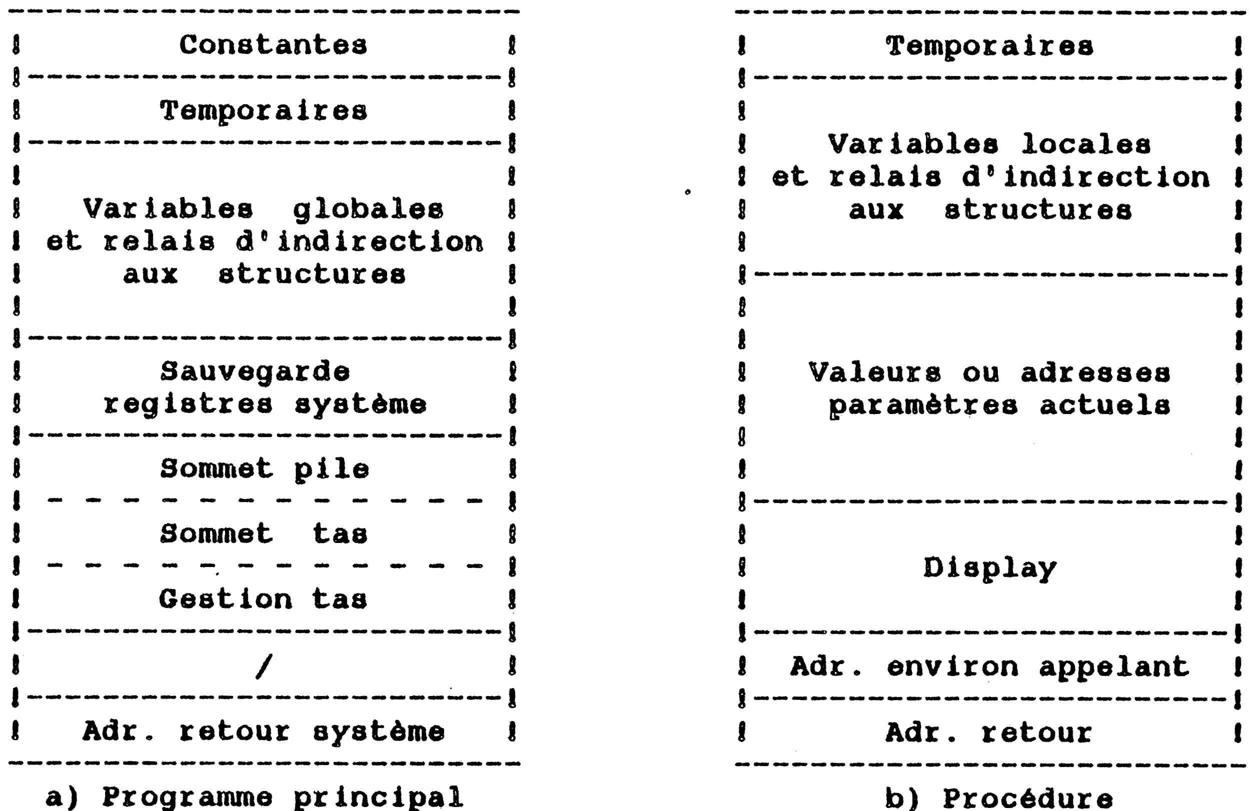


Figure VI.5. Organisation des environs de procédure.

Ces informations sont organisées suivant le schéma de la figure VI.5. On peut remarquer que :

a) La taille de toutes ces informations est connue dès la compilation ; par conséquent, celle de l'environ l'est aussi.

b) En tête de l'environ, nous trouvons une zone de liaison, constituée par les liens de la procédure avec son contexte et par les paramètres qui lui sont passés.

c) La position relative de chaque composant de la zone de liaison est identique pour toutes les procédures.

d) L'environ du programme principal est légèrement différent :
- il n'y a pas de display car il s'agit du niveau statique 0,
- le lien vers l'appelant est nul car le programme est activé par le système,
- la zone de paramètres est remplacée par une zone de variables d'état de la gestion mémoire et de sauvegarde des

registres du système,

- il existe une zone dite de constantes, dans laquelle sont logées toutes les constantes n'ayant pas pu être utilisées en tant qu'opérandes immédiats.

e) Les objets locaux sont regroupés dans 2 zones à la fin de l'environ. La première zone contient les variables locales et les relais d'indirection ; son organisation est déterminée par l'allocateur de variables (cf. VI.4). La seconde contient les temporaires dont la gestion est assurée par le noyau de génération (cf. VII.1.4).

La construction et l'initialisation d'un environ sont décrites dans la présentation du mécanisme d'appel de procédure.

2.2. Gestion du tas.

La gestion du tas pose principalement deux problèmes :
- l'existence de trous, provoqués par la libération de données,
- la recherche du bloc mémoire adéquat à la donnée allouée.

Pour résoudre ces problèmes, nous avons choisi d'utiliser la méthode "first fit" (‹Knuth71›).

Cette méthode maintient une liste des blocs mémoire libres, ordonnée en fonction des adresses de ces blocs. Une allocation consiste à parcourir cette liste jusqu'à ce que l'on trouve un bloc assez grand pour satisfaire la demande ; si le bloc trouvé est trop grand, l'excédent est restitué à la liste de blocs libres. Une libération consiste à rajouter l'espace libéré à la liste de blocs ; si cet espace est adjacent à un bloc libre, il est fusionné avec ce dernier.

Cette méthode n'est pas très efficace lors d'une libération car elle est obligée de chercher la position exacte du bloc libéré, dans la liste de blocs libres. D'autres méthodes permettent un accès plus direct ; toutefois, nous avons choisi cette méthode pour deux raisons principales :

a) Pour obtenir l'accès direct, on doit réserver quelques mots supplémentaires par bloc de mémoire (libre ou occupé) ; nous considérons que cette pénalisation est trop forte dans notre

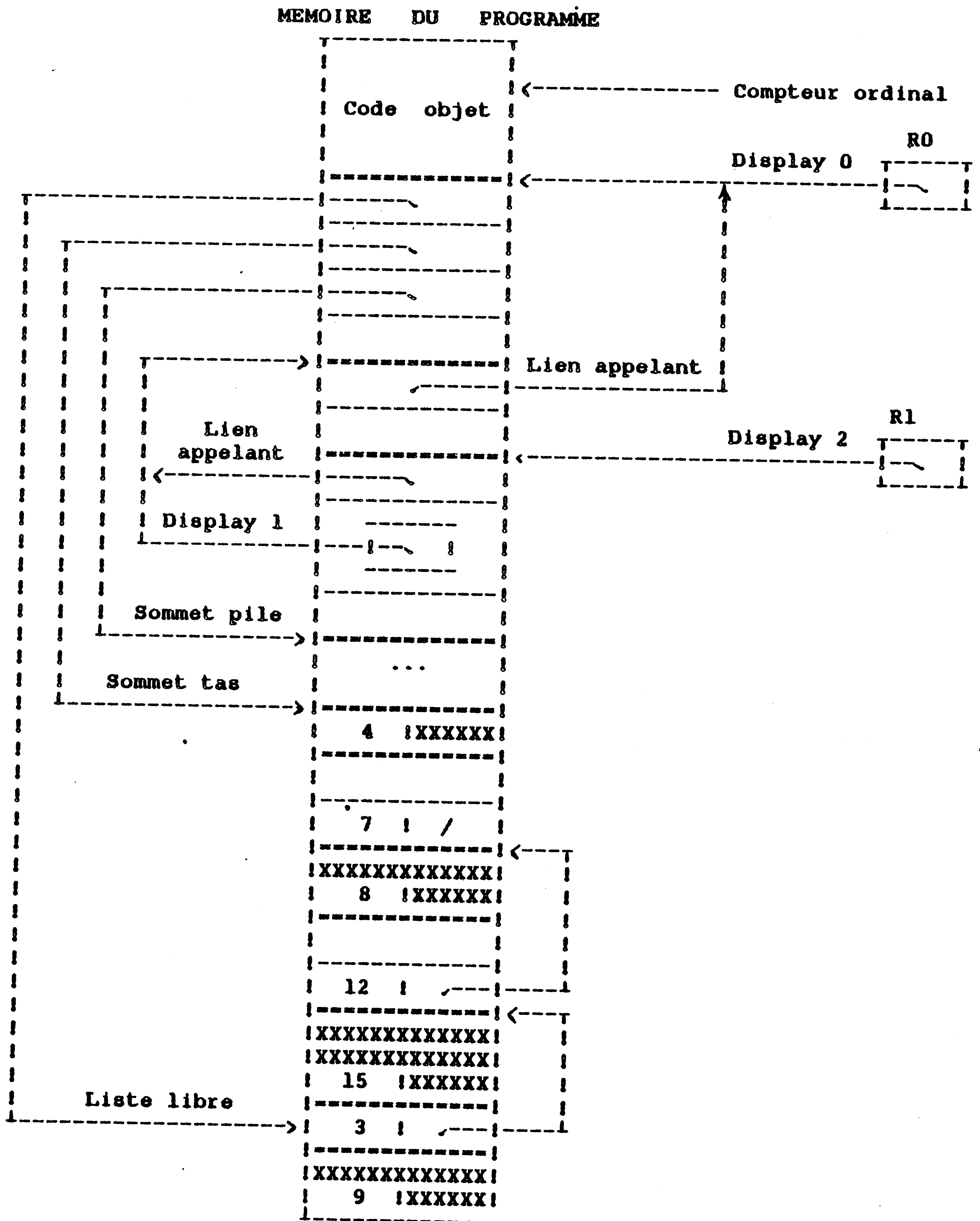


Figure VI.6. Instantané de la mémoire d'un programme à l'exécution.

cas car les blocs manipulés sont en général de petite taille. Avec la méthode "first fit", la surcharge par bloc est

réduite à la place occupée par un entier et est destinée à contenir la taille du bloc.

b) D'après notre expérience, le nombre de libérations est assez faible. Nous considérons donc que l'inefficacité de notre méthode n'est pas capitale.

Pour terminer, il faut indiquer que nous ne réalisons jamais de compactage du tas (garbage collection) ; par conséquent, si le tas et la pile d'exécution se rejoignent à un instant donné, l'exécution du programme sera arrêtée.

La figure VI.6 illustre la gestion de la mémoire du programme à un instant donné de l'exécution.

3. Compilation des définitions de types.

Pascal offre un certain nombre de types élémentaires (integer, real, boolean et char) mais il donne aussi la possibilité de se définir ses propres types, à l'aide d'un certain nombre de constructeurs qu'il fournit. La compilation de ces définitions de type consiste à trouver une représentation des types ainsi définis dans la machine cible ; c'est l'allocateur de variables qui réalise cette tâche dans les générateurs GEMME.

Les résultats de ce traitement sont gardés dans la table des types. Une entrée de cette table est une description complète de la représentation qui a été associée à une définition de type ; le numéro de cette entrée identifie les types correspondants durant tout le reste de la génération. La représentation des types élémentaires est aussi gardée dans la table des types ; il s'agit de la représentation qui leur a été assignée dans la description de la machine cible.

Nous présentons ci-dessous le traitement effectué pour chaque définition de type ; nous distinguons dans notre exposé 3 classes de types : simples, pointeurs et structurés.

3.1. Les types simples.

Nous regroupons dans cette classe les types scalaires et les types intervalles. Les premiers sont définis par énumération de l'ensemble ordonné des valeurs qui les composent ; ces valeurs sont des constantes et sont désignées par des identificateurs. Les types intervalles sont définis comme un intervalle d'un type scalaire ou d'un type élémentaire (à l'exception des réels).

Nous avons choisi de traiter les types simples comme des entiers. Pour ce faire, nous associons les valeurs 0, 1, 2 et ainsi de suite aux constantes de chaque type scalaire, dans leur ordre de définition ; ainsi, tous les types simples ont des valeurs entières. Il ne faut pas oublier qu'un intervalle ne peut être défini que sur des entiers ou sur un type scalaire (on peut considérer comme tels les types "boolean" et "char").

L'allocateur essaye de réduire l'espace qui sera occupé par les objets de types simples ; ainsi :

a) Si toutes les valeurs du type traité peuvent être représentées dans l'espace assigné au type "char", sans aucune manipulation, il utilise cet espace. Nous supposons que le type "char" occupe toujours une place inférieure (au maximum égale) à celle du type "integer", ce qui est assez réaliste.
b) Sinon, il utilise l'espace occupé par un entier.

La place occupée par les types élémentaires est un paramètre donné dans la description de la machine cible.

Supposons, par exemple, que la machine cible soit le Motorola 68000, que les caractères soient représentés sur un octet et les entiers sur 4 octets. Les types suivants seront associés au type "char" :

```
Couleur = ( rouge, bleu, jaune, violet, orange, vert ) ;
CouleurBase = rouge..jaune ;
Depl8Bits = -128..127 ;
```

Par contre, ces autres types seront associés au type "integer" :

```
Depl16Bits = -256..255 ;
Entiers = ( Zéro, Un, Deux, ..., Mille ) ;
```

Années = 1900..1999 ;
Remarquer que le type Années n'a que 100 valeurs et qu'il est possible de le représenter sur un octet, en décalant ses valeurs. Cependant, sa compatibilité avec les entiers et avec les intervalles d'entiers pose certains problèmes ; en effet, si une donnée de ce type était utilisée conjointement avec un entier, il faudrait générer une conversion en plus de l'opération proprement dite, ce qui surchargerait le code généré.

Pour terminer, il faut indiquer que l'alignement d'un type simple est identique à celui de son type associé ("char" ou "integer").

3.2. Les types pointeurs.

En Pascal, un pointeur ne peut faire référence qu'à des données d'un seul type ; l'association entre le pointeur et ce type est réalisée dans la définition du type pointeur correspondant. En outre, tous les pointeurs peuvent prendre la valeur "nil", pour indiquer qu'ils ne repèrent aucune donnée.

Cette restriction est verifiée par la partie analyse du compilateur et n'a aucune incidence au niveau génération de code. Nous pouvons par conséquent traiter tous les types pointeur de la même façon.

En effet, l'allocateur associe à tous ces types la représentation du type adresse de la machine cible, donnée en paramètre dans la description de celle-ci. Un type pointeur prend donc les propriétés d'alignement et de taille du type adresse ainsi que l'ensemble des opérations possibles sur celui-ci ; toutefois, il garde pour les besoins de la compilation, une information qui lui est propre : un lien vers le type qu'il peut référencer.

3.3. Les types structurés.

Les types structurés sont définis à partir d'autres types, en

utilisant l'une des méthodes de structuration offertes par le langage : les tableaux, les enregistrements, les ensembles et les fichiers ; la recherche d'une représentation peut par conséquent ne pas être triviale.

Dans ce paragraphe, nous présentons la technique de représentation que nous utilisons pour chaque méthode de structuration ; l'accès à ces structures n'est discuté que plus loin dans ce chapitre. Il faut signaler que nous ignorons l'attribut "packed" sauf s'il s'agit d'une chaine de caractères.

3.3.1. Les tableaux.

Un tableau est constitué d'un nombre fixe d'éléments de même type ; les tableaux Pascal ont certaines particularités :

a) Le nombre d'éléments d'un tableau est donné dans sa définition ; il est donc connu statiquement.
b) Le type des éléments peut être quelconque ; en particulier, il peut s'agir d'un autre tableau, donnant ainsi lieu à un tableau à plusieurs dimensions.
c) Un tableau peut être utilisé en entier dans certaines opérations (affectation, comparaison).

D'après ces points, nous avons choisi d'utiliser un rangement contigu par ligne pour les tableaux, la représentation des éléments étant celle de leur type. L'alignement d'un élément est toujours respecté, ce qui peut provoquer des trous entre les éléments mais qui accelère l'accès ; l'alignement du tableau sera par conséquent identique à celui de ses éléments.

L'allocateur des variables construit un descripteur pour chaque type tableau traité et le garde dans la table de types ; ce descripteur est formé :

- de la taille du tableau,
- du type des éléments,
- de l'origine virtuelle du tableau (l'adresse de l'élément 0 s'il existait),
- d'autant de triplets que le tableau a de dimensions.

Chaque triplet comprend l'enjambée (distance entre 2 éléments),

la borne inférieure et la borne supérieure de la dimension correspondante.

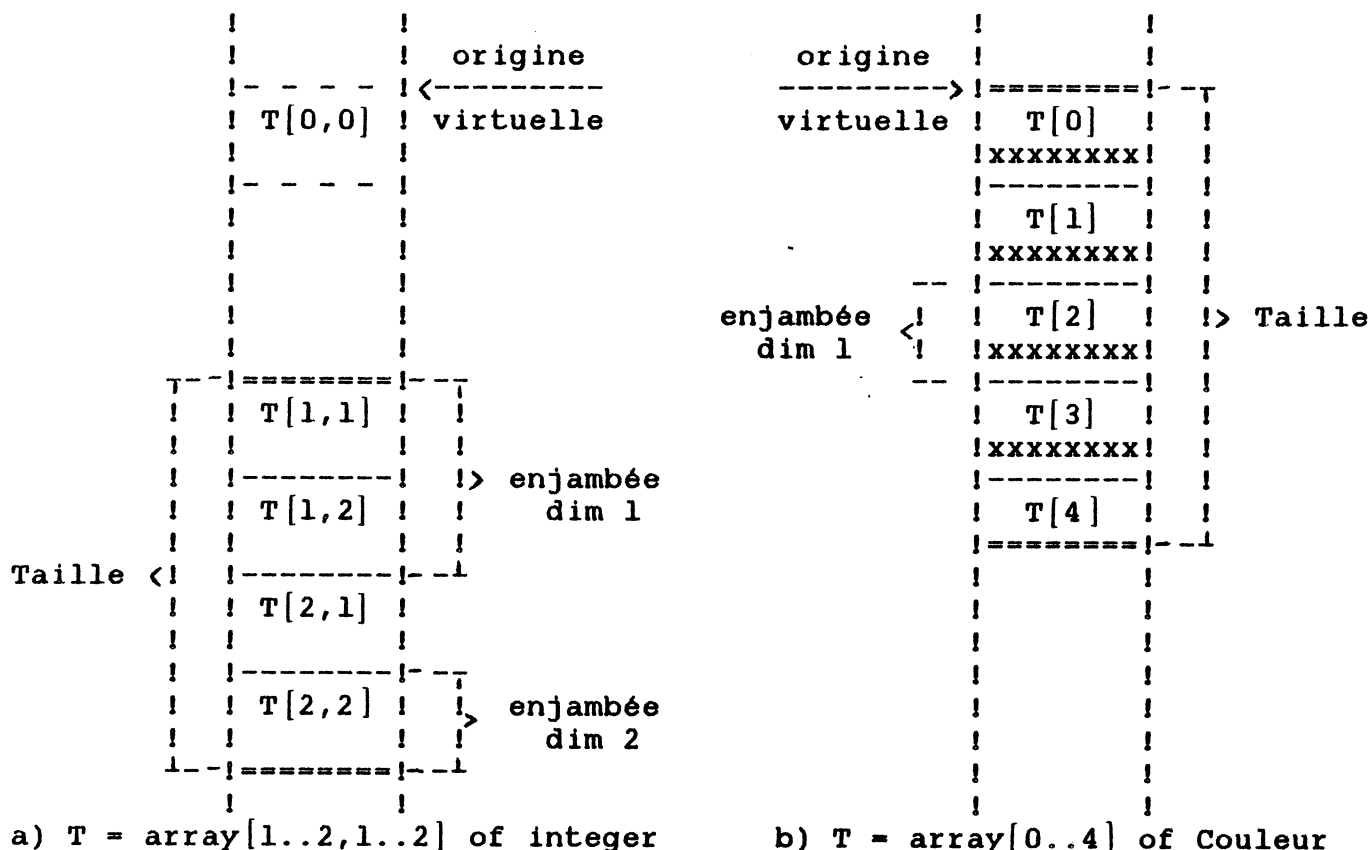


Figure VI.7. Représentation des tableaux.

La figure VI.7 illustre cette représentation ; on a supposé que tous les objets doivent être alignés sur une frontière mot (2 octets) et que le type Couleur est représenté sur un octet.

Le descripteur d'un tableau indique également si l'adressage du tableau doit se faire via un relais d'indirection. Ce choix est fait dans le but d'optimiser l'accès au tableau lorsque l'adressage indexé de la machine cible est en fait un adressage indirect post-indexé ; en effet, avec ce type d'adressage le relais d'indirection est indispensable et il est préférable de le prévoir statiquement. L'allocateur associera par conséquent un pointeur à chaque tableau et demandera la génération (en tête de la procédure) des instructions nécessaires à leur initialisation ; ce pointeur contiendra l'adresse de l'origine virtuelle du tableau correspondant. Si le tableau a plusieurs dimensions, un seul relais est utilisé : celui du tableau englobant.

3.3.2. Les enregistrements.

Contrairement aux tableaux, les composants d'un enregistrement ne sont pas nécessairement de même type. En outre, la sélection d'un composant est faite par désignation directe plutôt que par une valeur calculée ; en effet, tout composant ou champ d'un enregistrement est repéré par un identificateur, déclaré lors de la définition de l'enregistrement.

Par contre, certaines propriétes des tableaux ont leurs équivalentes dans les enregistrements Pascal :

a) La structure d'un enregistrement est connue dès la compilation.
b) Le type d'un champ peut être lui-même structuré.
c) L'enregistrement en entier peut servir comme opérande dans certaines opérations.

La représentation la plus simple d'un enregistrement est le rangement contigu de ses composants ; ainsi, l'adressage d'un champ peut se faire de façon relative à l'origine de l'enregistrement, avec un déplacement connu par le compilateur. La représentation d'un champ est celle de son type ; en particulier, l'alignement correspondant est forcé, ce qui peut donner lieu à un trou entre 2 champs. Pour optimiser l'espace occupé, l'allocateur ordonne les champs en fonction de leurs alignements, de façon à réduire ces trous ; après ce traitement, l'allocateur détermine l'alignement et la taille de l'enregistrement.

La figure VI.8 illustre la représentation du type suivant :

```
Date = record
          Jour : 1..31 ;
          Mois : 1..12 ;
          Année: 1900..1999
       end ;
```

Le premier cas correspond à une machine 16 bits où les objets doivent être alignés sur une frontière mot tandis que le

deuxième correspond à un alignement sur une frontière octet.

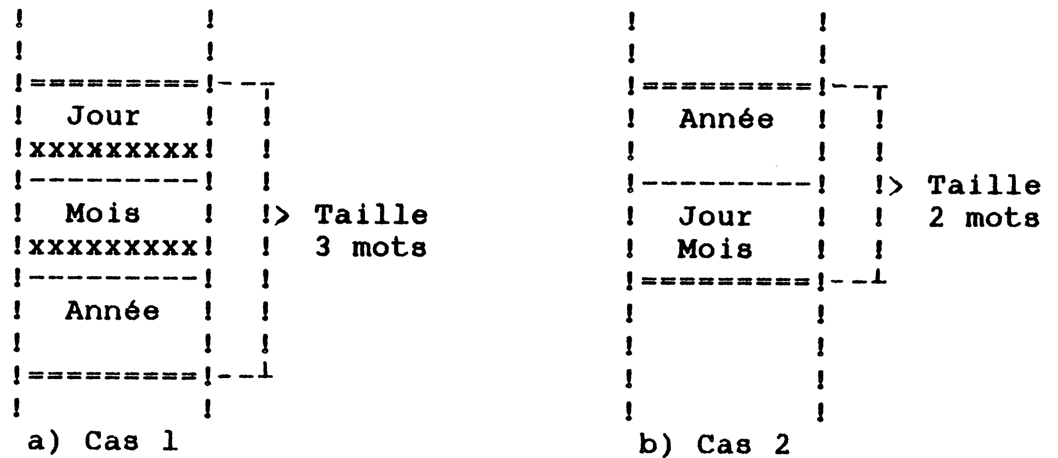


Figure VI.8. Représentation d'un enregistrement.

L'imbrication de structures ne pose pas de problèmes majeurs car toutes les structures Pascal sont statiques. Il y a toutefois une exception : les tableaux contenus dans un enregistrement qui sont adressés via un relais d'indirection ; le relais d'un tel tableau est placé à l'intérieur de l'enregistrement, ce qui nécessite un traitement exceptionnel lorsque la structure entière est utilisée comme opérande. Par exemple, s'il s'agit d'une affectation d'enregistrements, toute la structure doit être recopiée sauf les relais ; ceux-ci ne doivent pas être modifiés car l'origine virtuelle de leurs tableaux respectifs ne changent pas.

En ce qui concerne les parties variantes d'un enregistrement, elles sont représentées par recouvrement d'un même espace ; la représentation et l'accès aux champs est identique à celle de la partie fixe. La taille de la zone de recouvrement est celle de la partie variante la plus longue et son alignement est le plus fort de ceux des parties variantes. Il faut signaler que les parties variantes sont traitées avant la partie fixe et que toute la zone de recouvrement joue le rôle d'un champ unique lors du traitement de la partie fixe.

3.3.3. Les ensembles

Un type ensemble définit l'ensemble des parties d'un autre

type (son type de base) ; ses valeurs possibles sont donc des ensembles de valeurs de ce dernier.

Le type de base d'un ensemble doit être ordinal, c'est à dire : "integer", "char", "boolean" ou bien un type simple ; d'autre part, l'ensemble doit être défini dans les limites fixées par l'implémentation.

Pour représenter un ensemble, nous utilisons une chaine de bits dont la longueur est déterminée par le nombre de valeurs du type de base ; chaque bit est associé à une valeur et indique la présence ou l'absence de cette valeur dans l'ensemble. La figure VI.9 donne des exemples de cette représentation.

bits	0	1	2	3	4	5	6	7
	rouge	bleu	jaune	violette	orange	vert		

a) EnsCouleur = set of Couleur

bits	0	1	2		24	25		31
	'a' ou 97	'b' ou 98	'c' ou 99	...	'y' ou 121	'z' ou 122	...	

b) EnsMinuscules = set of 'a'..'z'

Figure VI.9. Représentation des ensembles.

Dans notre implémentation, la taille de la chaine de bits est limitée à 128, ce qui nous permet, en particulier, d'accepter les ensembles de caractères. Il faut noter que cette taille est assez importante car elle nécessite 16 octets ; elle n'est toutefois qu'une limite supérieure. En effet, nous n'utilisons pour un ensemble donné que la taille nécessaire à la représentation de toutes ses valeurs, arrondie en fonction des contraintes de la machine cible.

Avec cette représentation, les opérations définies sur les ensembles peuvent être mises en oeuvre en utilisant les opérations logiques de la machine ; ainsi, par exemple, l'union de 2 ensembles peut être réalisée par un "ou" logique. Certaines opérations peuvent nécessiter une suite d'opérations logiques ; c'est le cas de l'inclusion qui peut être réalisée par un "ou" logique suivi d'une comparaison avec l'ensemble réputé englobant.

3.3.4. Les fichiers.

Une structure de fichier est une séquence d'éléments de même type dont l'ordre est défini naturellement par cette séquence.

Les propriétés et les opérations définies sur cette structure correspondent bien à la notion de fichier séquentiel ; nous utiliserons par conséquent l'implémentation donnée à cette notion par le système hôte de la machine cible. Pour ce faire, nous avons défini une interface qui permet de relier notre représentation interne d'un fichier avec celle du système ; cette interface est la spécification d'un module qui doit être fourni avec la description de la machine et qui doit permettre de faire appel aux fonctions d'entrée/sortie du système.

La représentation interne d'un fichier est constituée par un descripteur qui regroupe les informations relatives au fichier ; ces informations sont déterminées soit :

a) statiquement, grâce à la définition du type fichier (type et taille des éléments, longueur du tampon qui sera associé au fichier) et à la déclaration de la variable fichier (nom interne, fichier local ou global, fichier texte ou autre) ;
b) dynamiquement, suivant les opérations effectuées sur le fichier (liaison externe établie ou non, nom de la liaison, nom externe du fichier, type d'ouverture, etc).

Lors de la définition d'un type fichier, l'allocateur des variables se contente de déterminer les informations indiquées ainsi que la taille mémoire et l'alignement du descripteur. La taille des descripteurs varie à cause du tampon d'entrée/sortie

associé au fichier car celui-ci dépend du type des éléments du fichier.

Notre implémentation des fichiers comporte 2 contraintes ; nous n'admettons ni les fichiers imbriqués dans une structure ni les pointeurs vers des fichiers.

4. Allocation de mémoire.

Après avoir trouvé une représentation aux types définis par l'utilisateur, la tâche suivante de l'allocateur est d'assigner une adresse mémoire à chaque variable du programme et de lui associer le descripteur de valeur correspondant. L'allocation de mémoire aux temporaires et aux constantes non immédiates n'est pas réalisée par l'allocateur mais par le noyau de génération, durant la compilation des expressions.

Une optimisation consiste à assigner des registres (au lieu d'adresses mémoire) aux variables les plus utilisées. Cependant, une telle optimisation implique une analyse globale du programme pour identifier ces variables, ce qui concerne l'optimiseur global plutôt que l'allocateur ; ce dernier se contente donc de repartir toutes les variables en mémoire.

L'allocation de mémoire aux variables est en général une tâche simple dans un compilateur classique ; en effet, toutes les contraintes mémoire de la machine cible étant connues, on peut trouver une stratégie simple qui s'accommode de ces contraintes. Cette même tâche devient plus compliquée lorsqu'il s'agit d'un compilateur indépendant des machines ; 2 problèmes principaux se posent :

a) Dans certaines machines, les données doivent être alignées sur une frontière donnée suivant leurs tailles. L'allocateur doit par conséquent forcer l'alignement correspondant à chaque variable, ce qui peut faire apparaître des trous entre les variables.

b) L'intervalle des déplacements utilisables avec l'adressage basé varie d'une machine à l'autre. De plus, certaines

machines offrent la possibilité d'utiliser plusieurs intervalles de déplacements ; évidemment, le coût de l'adressage augmente avec la portée de l'intervalle.

Nous avons développé une méthode d'allocation qui apporte une solution à ces problèmes, tout en optimisant la place totale allouée et l'adressage des variables traitées. Cette méthode se déroule en 3 étapes que nous présentons ci-dessous ; elle est appliquée successivement aux déclarations de chaque procédure.

4.1. Classement des variables.

La première étape de notre méthode d'allocation regroupe les variables de taille similaire et constitue avec elles des zones de données ; à l'intérieur de chaque zone, les variables sont d'autre part triées selon leurs alignements (le plus fort d'abord).

Les zones de données constituées sont au nombre de 3 :
- Zone 1 regroupant les variables de type élémentaire, de type scalaire ou intervalle, les pointeurs, les ensembles de petite taille et les relais d'indirection des tableaux.
- Zone 2 formée par les enregistrements, les fichiers et le reste des ensembles.
- Zone 3 constituée par les tableaux.

L'objectif de ce classement est double. D'une part, il permet à la deuxième étape de trouver le meilleur positionnement des variables du point de vue adressage ; d'autre part, il permet de réduire les trous dus à l'alignement.

4.2. Analyse des intervalles de déplacement.

Dans cette étape, nous déterminons la meilleure position dans l'environ de la procédure pour chaque zone de données ; ceci est fait en fonction des intervalles de déplacement utilisables avec l'adressage basé de la machine cible et de la priorité que nous associons à chaque zone. Le rôle de cette priorité est de privilégier l'adressage des variables suivant leur taille et

leur type de structuration ; ainsi, les variables de la zone 1 ont la priorité la plus forte tandis que celles de la zone 3 ont la plus faible.

Les intervalles les plus courts étant les moins coûteux, nous nous efforçons de les appliquer autant que possible aux zones les plus prioritaires ; ce choix a été fait en considérant que l'accès aux structures nécessite assez souvent un calcul dynamique d'adresse et, qu'en conséquence, un coût additionnel dans leur adressage était moins pénalisant que pour des variables simples.

En plus des 3 zones définies dans le paragraphe précédent, nous faisons intervenir 2 autres zones dans l'analyse des intervalles : la zone de liaison et la zone de temporaires de la procédure (cf VI.2.1.2).

La zone de liaison doit nécessairement être placée au début de l'environ ; elle occupe par conséquent les déplacements inférieurs qui risquent en général d'être les moins avantageux. Etant donné que cette zone contient les paramètres et le display de la procédure, nous essayons de faire coincider autant que possible ces déplacements avec les déplacements les moins coûteux ; ainsi, si un intervalle contient des déplacements positifs et négatifs, nous le traitons comme s'il s'agisait de 2 intervalles : le premier formé par ses déplacements positifs plus les déplacements négatifs de l'intervalle précédent, et le deuxième formé par l'intervalle en entier.

La zone des temporaires est toujours placée à la fin de l'environ sauf s'il s'agit du programme principal ; dans ce cas, c'est la zone des constantes qui est placée en dernier, précédée par celle des temporaires. Nous ne cherchons en aucun cas à optimiser l'adressage de ces 2 zones ; en effet, nous considérons que leur utilisation n'est pas très fréquente et que nous pouvons employer sans grande perte les déplacements qui restent (qui peuvent d'ailleurs être les moins coûteux).

Il peut arriver qu'aucun intervalle ne soit suffisant pour

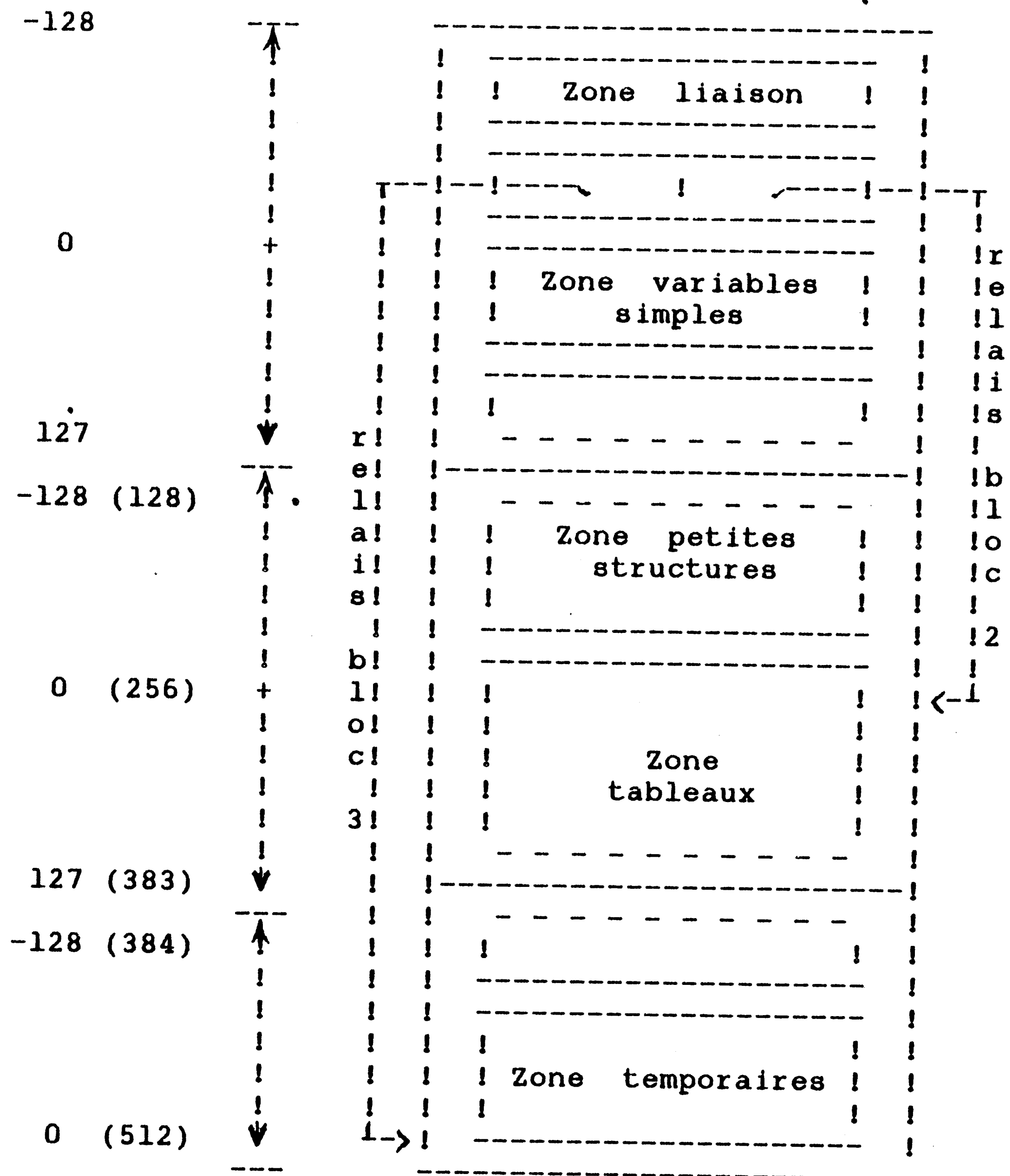


Figure VI.10. Exemple d'allocation de mémoire (Solar 16)

adresser toutes les informations d'un environ. Dans ce cas, nous procédons de la façon suivante :

- Nous déterminons la taille de l'environ, en plaçant les zones suivant leurs priorités,
- Nous calculons ensuite le nombre de blocs, de taille égale à celle de l'intervalle le plus grand, nécessaires pour contenir l'environ. Tous ces blocs, à l'exception du premier, seront adressés par l'intermédiaire de relais d'indirection placés dans le premier bloc.
- Nous générons enfin les instructions d'initialisation de ces relais, en tête de la procédure ; chaque relais pointera à la position correspondant au déplacement 0 de son bloc.

La figure VI.10 donne un exemple de cette technique pour le

Solar 16, où un seul intervalle de déplacement peut être utilisé (-128..127).

Il faut remarquer que les résultats de la méthode décrite sont d'autant plus appréciables que la taille de l'environ est grande.

4.3. Assignation des adresses.

L'adresse de tout objet de la procédure est constituée d'une base et d'un déplacement par rapport à celle-ci. La valeur de la base est dynamique car elle dépend de l'état de la pile d'exécution ; toutefois, on peut déterminer statiquement sa position par rapport au début de l'environ et, grâce à celle-ci, le déplacement de l'objet. L'assignation des adresses se charge de réaliser cette tâche et de construire le descripteur de valeur correspondant.

Comme la position de chaque zone et de chaque objet à l'intérieur de sa zone respective sont déjà connues, cette tâche se voit très simplifiée.

VII. LE CODEUR.

Le codeur est le composant principal d'un générateur de code GEMME car c'est lui qui est chargé d'effectuer la génération de code proprement dite ; il utilise pour ce faire la méthode proposée au Chapitre III.

L'entrée du codeur est une représentation arborescente du programme source (Annexe D), obtenue à partir de la RI Adèle. Elle est moins chargée que celle-ci car elle ne contient que les informations nécessaires à la génération de code ; de plus, la représentation de l'arbre est faite en utilisant moins de pointeurs, en raison du parcours simple effectué par le codeur.

Durant le parcours de l'arbre, les différents composants du codeur sont activés, en fonction du type de noeud visité ; tous les composants sont spécialisés dans le traitement d'une certaine classe de noeuds :

a) Le noyau de génération traite les opérations dont les opérandes sont soit des feuilles, soit des sous-arbres déjà traduits.
b) Le traducteur d'expressions traite les différents types d'expressions : arithmétiques, logiques, de comparaison et d'accès aux éléments d'une variable structurée.
c) Le traducteur d'instructions traite les instructions Pascal ainsi que les procédures et les fonctions.

Toutefois, l'imbrication des noeuds de l'arbre fait que les rapports entre ces composants sont très étroits ; ainsi, par exemple, le traducteur d'expressions fait appel au noyau pour traduire les opérations qui forment une expression.

Dans ce qui suit, nous présentons les 3 composants du codeur ainsi que la représentation utilisée pour le code objet qui est produit.

1. Noyau de génération de code.

Le noyau de génération est la couche la plus basse du codeur ; il fournit toutes les fonctions de base nécessaires à la génération et fait la gestion des ressources de la machine cible.

Sa fonction principale est de générer la séquence d'instructions qui réalise une opération donnée entre 2 opérandes (au maximum). Il le fait en consultant les tables de description de la machine d'où il extrait la séquence qui est la moins chère ; il fait aussi toutes les allocations et les libérations de ressources nécessaires à cette séquence. Le noyau ne sait traiter que des opérations élémentaires : il saura générer les instructions machine qui correspondent à l'addition de 2 opérandes mais non celles d'une expression telle que "a+b-c".

Les ressources décrites par l'utilisateur (les registres) et celles dérivées par le générateur à partir de la description (temporaires, constantes immédiates, constantes non immédiates, etc.), sont entièrement gérées par le noyau. Cette gestion est invisible pour celui qui décrit la machine ; en effet, il fournit des informations utilisées par le noyau mais il ne peut pas paramétrer la gestion et moins encore la redéfinir.

Le noyau est constitué de 5 modules, auxquels nous avons donné les noms suivants :
- Evaluer.
- Générer.
- Gestion des registres.
- Gestion des temporaires et des constantes non immédiates.
- Gestion des descripteurs.

Chaque module fournit un ensemble de primitives qui sont utilisées par les autres ; des appels croisés fortement recursifs sont trés fréquents (figure VII.1). Un sous ensemble de ces primitives est fourni à l'extérieur du noyau : il est

formé par certaines primitives d'allocation et de libération de ressources et par la primitive Génération. Les premières permettent aux couches englobantes de forcer l'allocation d'une ressource pendant un certain temps ; par exemple, la variable de contrôle d'une boucle "for" peut être mise dans un registre durant toute la boucle, ce qui réduit le temps d'exécution.

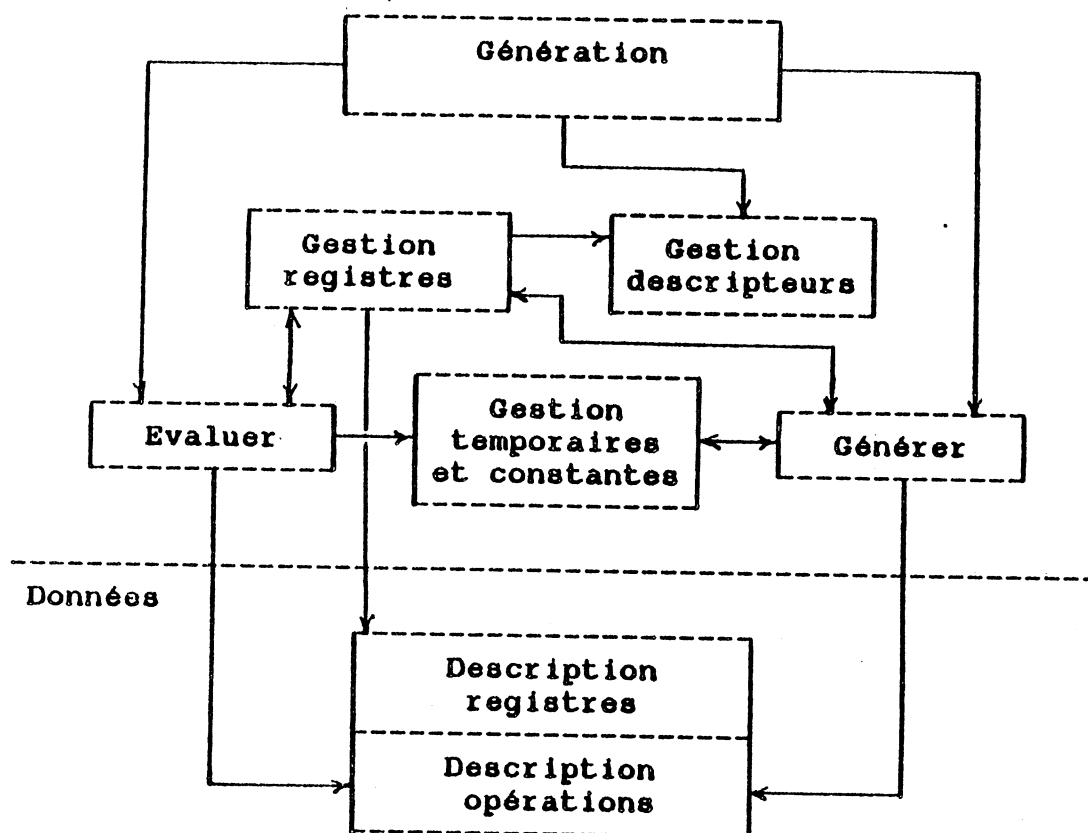


Figure VII.1. Graphe d'appels inter-modules du noyau.

La génération de code se déroule toujours en 2 étapes, la première analyse les possibilités de traduction de l'opération et choisit la séquence d'instructions la plus appropriée (étape d'évaluation) ; la deuxième fait la génération effective de code (étape de génération effective). Il est impossible de générer directement du code car nous ne connaissons la séquence d'instructions à générer qu'après avoir analysé toutes les possibilités ; en effet, une possibilité peut s'avérer impossible ou trop chère pendant son analyse, ce qui va provoquer un retour en arrière dans l'évaluateur. Toutefois, une

description des actions à effectuer lors de la génération effective est produite par la première étape ; cette description est utilisée par la seconde étape.

L'enchaînement de ces 2 étapes est fait par la primitive "Génération" ; elle accepte en entrée le nom de l'opération à faire et les descripteurs de ses opérandes, et elle rend le descripteur du résultat de l'opération.

Nous passons en revue ci-après les modules qui constituent le noyau.

1.1. Evaluation d'une opération élémentaire.

Cette évaluation est mise en oeuvre par le module Evaluer ; son rôle est de chercher la séquence d'instructions qui réalise l'opération demandée au moindre coût. Par exemple, s'il s'agit d'une addition et si la machine possède n façons de réaliser cette opération, il faut choisir la moins coûteuse par rapport à un critère fixé dans la description de la machine ; si on a besoin d'un registre, il faut choisir celui dont l'allocation coûte le moins, etc ; il faut remarquer que tous ces choix donnent le meilleur code mais au niveau strictement local. L'algorithme général de l'évaluation est donné dans la figure VII.2.

L'algorithme d'évaluation utilise la méthode de sélection d'instructions présentée au chapitre III. En conséquence, il essaye d'abord de trouver une conformité parfaite entre les opérandes et les modèles de génération de l'opération ; en cas d'échec, il essaye de transformer les opérandes, de façon à les rendre conformes aux modèles correspondants.

Si un opérande n'est pas du même type que celui demandé par l'opération, le système fait la conversion appropriée. L'analyseur sémantique rajoute explicitement toutes les conversions propres au langage : celles demandées par l'utilisateur (ord, trunc, round, chr) et celles dues aux expressions qui mélangent des entiers et des réels (conversions

Algorithme 1 : Evaluer(Opération, Opérande1, Opérande2)

Pas 1 :
Si les opérandes ne sont pas du même type que ceux de l'opération, évaluer leurs conversions respectives.

Pas 2 :
Faire un premier parcours de la liste de possibilités de réalisation de l'opération : chercher une conformité parfaite entre les opérandes et les modes d'adressages respectifs, qui sont acceptés par chaque possibilité. Si la recherche réussit, choisir la possibilité la moins chère et retourner le chemin de génération correspondant ; si elle échoue, continuer au pas 3. L'algorithme detaillé de ce parcours est donné dans la figure VII.3.

Pas 3 :
Faire un deuxième parcours de la liste de possibilités. Cette fois-ci nous savons qu'aucune possibilité ne convient directement à nos opérandes : donc, nous allons essayer de les transformer pour forcer leur conformité avec les modes d'adressage de chaque possibilité. Ce parcours réalise plusieurs actions sur chaque possibilité :

a) Bloquer les ressources nécessaires au résultat de l'opération et aux conversions faites dans le premier pas : nous empêchons ainsi des doubles allocations.
b) Transformer l'opérande gauche s'il n'est conforme à aucun des modes d'adressage.
c) Idem pour l'opérande droit.
d) Débloquer toutes les ressources gélées dans a).

Pas 4 :
Choisir la possibilité qui est la moins chère après transformation de ses opérandes. Retourner son chemin de génération.

Figure VII.2. Algorithme d'évaluation d'une opération.

implicites) ; de plus, certaines conversions n'apparaissent qu'au moment de la génération de code car elles sont dues à certaines instructions de la machine. Il s'agit toujours de problèmes de longueur d'opérande : par exemple, une multiplication entière 16 bits du Motorola 68000 produit toujours un résultat 32 bits, qui doit être reconverti en 16 bits pour pouvoir l'utiliser ; notre algorithme détecte ces problèmes et force la conversion correspondante. Les conversions sont décrites par l'utilisateur de la même façon que les autres opérations mais c'est toujours le noyau qui les introduit dans le programme compilé.

Les ressources réservées par l'allocateur des variables sont

protégées contre toute modification ; en effet, le noyau ne doit pas y mettre des résultats intermédiaires car il détruirait la valeur des variables du programme. S'il s'agit d'une affectation du programme, le mécanisme de protection est contourné et la modification forcée ; le noyau est donc capable de reconnaître une affectation du programme et de la traiter correctement. D'autre part, le noyau peut toujours modifier les ressources allouées temporairement (par lui-même) et il doit même essayer de les réutiliser autant que faire se peut.

Si l'évaluation de l'opération est réussie, un chemin de génération est construit et retourné à l'appelant. Ce chemin est une description detaillée des actions à exécuter lors de la génération effective du code ; il contient, en particulier, les actions définies par le modèle de génération choisi (commandes d'émission de code) et des demandes d'allocation et de libération de ressources (registres, temporaires, etc).

Si l'évaluation échoue, la compilation du programme est arrêtée et un certain nombre d'informations sur l'état du générateur sont imprimées. L'évaluateur n'a pas pu trouver une séquence d'instructions pour mettre en oeuvre l'opération demandée, même après avoir tenté de transformer les opérandes ; dans la plupart des cas, cet échec est dû à un oubli dans la description des instructions ou à des opérations n'ayant pas été décrites (en général, des affectations). Une autre possibilité peut être une description incorrecte, auquel cas les informations imprimées seront très utiles pour trouver l'erreur.

L'algorithme d'évaluation analyse toutes les possibilités d'une opération et tous les modes d'adressage associés à chaque possibilité ; il est donc fortement combinatoire, comme on peut le constater dans l'annexe E. Cependant, deux facteurs réduisent beaucoup le nombre de combinaisons possibles :

a) Dans une machine donnée, on ne dispose le plus souvent que d'une seule instruction pour réaliser <u>une</u> opération.
b) Le nombre de modes d'adressage permis par les instructions d'une machine ; les plus riches ne dépassent jamais 8 modes d'adressage.

De plus, nous avons écrit les algorithmes de façon à réduire les combinaisons et à obtenir les meilleures performances possibles. En particulier :

a) L'algorithme de reconnaissance de formes abandonne l'évaluation d'une possibilité dès qu'il trouve une incompatibilité ou un coût supérieur à celui du choix courant.
b) Un mode d'adressage n'est évalué qu'une seule fois pour un opérande donné ; sachant qu'un mode est généralement utilisable dans plusieurs instructions, ceci nous permet de gagner beaucoup de temps, surtout lorsque le jeu d'instructions de la machine est très riche.
c) Les informations obtenues dans une passe sont toujours gardées pour les passes suivantes ; ainsi,on ne refait jamais 2 fois un même calcul.

L'annexe E présente plusieurs exemples d'évaluation d'opérations ; nous les analysons en détail de façon à montrer le fonctionnement de l'évaluateur. Dans ce qui suit nous développons les 2 étapes correspondant à l'évaluation d'une opération.

1.1.1. Evaluation d'une conformité parfaite.

L'algorithme d'évaluation fait une première passe sur la liste de possibilités de l'opération traitée, à la recherche d'une conformité parfaite. Il vérifie donc, pour chaque possibilité, si les opérandes sont conformes aux modèles de génération de celle-ci et si la ressource nécessaire au résultat est disponible ; s'il y a plusieurs conformités parfaites, il retient la moins coûteuse. L'algorithme de cette recherche est présenté dans la figure VII.3.

Nous pouvons remarquer que l'algorithme alloue les ressources nécessaires au résultat de l'opération. En général, aucune allocation n'est nécessaire car la plupart des instructions machine mettent le résultat dans l'un des opérandes ; cependant, certaines instructions peuvent le faire dans une partie de la

Algorithme 2 : Recherche Conformité(Opération,Op1,Op2)

Pas 1 :
Pour toute possibilité de réaliser l'opération :
a) Evaluer l'allocation des ressources nécessaires au résultat.
b) Confronter l'opérande gauche avec les modes d'adressage gauches de la possibilité de réalisation. En cas de conformité, choisir le moins coûteux.
c) Appliquer le même traitement à l'opérande droit.
d) Si a), b) et c) ont réussi, retenir cette possibilité.

Pas 2 :
Choisir la possibilité la moins coûteuse parmi celles qui ont été retenues et retourner son chemin de génération.

Figure VII.3. Recherche d'une conformité parfaite.

ressource occupée par l'opérande (par exemple, un sous-registre pour la division) ou bien dans une ressource englobante (par exemple, un double registre pour la multiplication) ; de plus, il peut aussi être obtenu dans une toute autre ressource, ce qui est le cas des instructions de comparaison qui positionnent un code de condition. L'algorithme identifie et réserve, suivant le cas, les ressources qui seront utilisées par le résultat ; il faut remarquer que l'allocation ne se produit qu'à la génération effective, ici elles sont seulement réservées ou mises de côté pendant l'analyse de la possibilité correspondante.

L'algorithme exploite la propriété de commutativité de certaines opérations ; en effet, si l'opération est commutative, le pas 1 (figure VII.3) est exécuté une deuxième fois en inversant le rôle des opérandes : le gauche devient le droit et réciproquement. C'est seulement après que le choix est fait, ce qui nous permet de toujours obtenir le meilleur code pour l'opération.

Nous devons aussi ajouter que l'algorithme construit, durant l'analyse, une structure de données décrivant la raison des échecs, là où il y en a eu. Si aucune conformité parfaite n'est trouvée, ces informations serviront de guide à l'algorithme de transformation qui est exécuté à la suite ; de même, nous économisons un certain nombre de calculs déjà faits.

1.1.2. Transformation d'une opération.

Si la recherche d'une conformité parfaite échoue, cela signifie que l'opération ne peut être réalisée directement par aucune des possibilités décrites. Il est nécessaire alors d'adapter les opérandes pour pouvoir réaliser l'opération.

Nous avons vu que toute possibilité accepte un ensemble de modèles pour chaque opérande, représentant les modes d'adressage qui peuvent être utilisés pour l'atteindre (l'adresser). Dans notre cas, l'un des opérandes au moins est incompatible avec les modes d'adressage qui lui correspondent, à l'intérieur de chaque possibilité ; nous devons donc transformer le (ou les) opérande(s) concerné(s) (figure VII.2, pas 3). Une transformation consiste à trouver une suite d'instructions qui chargent la valeur de l'opérande dans une ressource compatible avec le mode d'adressage désiré (figure VII.4).

MODES ADRESSAGE	TYPE OPERANDE		
	registre	constante	mémoire
registre	(1)	(2)	(3)
immédiat	impossible	impossible	impossible
mémoire	(4)	(5)	(6)

(1) Allouer un autre registre
Y transférer l'opérande

(2) Allouer un registre
Y charger la constante

(3) Allouer un registre
Y charger l'opérande

(4) Allouer un temporaire
Y ranger l'opérande

(5) Allouer une const. non immédiate.
L'initialiser.

(6) Transformer base et index s'ils ne sont pas conformes
Si depl. non conforme, impossible

Figure VII.4. Transformations pour un opérande ·

Si l'opérande n'est pas conforme, une transformation est essayée pour chaque mode d'adressage de la possibilité analysée ; ainsi, nous pouvons les évaluer et choisir la moins

chère. Si aucune transformation n'est réussie, la possibilité est abandonnée. Comme dans l'algorithme d'évaluation d'une conformité parfaite, nous profitons ici aussi de la propriété de commutativité de certaines opérations.

Si l'opération est une affectation, les transformations sont appliquées avec certaines restrictions, pour 2 raisons :

a) Il ne faut jamais transformer l'opérande gauche (sinon l'affectation ne serait pas faite à la variable désirée!!).
b) Elles peuvent faire boucler l'algorithme.
   Par exemple, supposons qu'on veuille affecter la valeur de I (une position mémoire) à R1 (un registre) et qu'il existe une instruction "LR Ri,Rj". Si l'on essaye d'appliquer cette instruction à notre affectation, on trouvera que R1 est conforme à Ri mais que I ne l'est pas à Rj ; donc, I doit être transformé (chargé dans un registre). Si aucun contrôle n'est fait, l'algorithme peut essayer d'utiliser LR pour faire ce deuxième chargement, se retrouvant avec exactement le même problème qu'au début.

Les restrictions nécessaires sont mises en oeuvre par l'algorithme de transformation.

1.2. Génération effective de code.

Une fois l'évaluation faite, la séquence d'instructions et les modes d'adressage les plus appropriés à l'opération et à ses opérandes, ont été déterminés et décrits dans le chemin de génération. Le module Générer a pour but d'exécuter les actions décrites dans ce chemin, générant ainsi le code correspondant à l'opération l'algorithme est présenté dans la figure VII.5.

Le résultat de Générer est mis dans 2 tables, utilisées comme entrée de la phase suivante (optimiseur final). La première (CodeSymb) contient toutes les instructions machine générées : chaque élémentrespond à une instruction et à ses opérandes ; un compteur d'assemblage pointe toujours vers la dernière instruction générée et indique par ce fait la taille de la table. Une entrée est composée par les informations suivantes :

a) Type d'opération.

b) Espace occupé par l'instruction.
c) Nom symbolique de l'instruction.
d) Description de l'opérande gauche (mode d'adressage et descripteur).
e) Description de l'opérande droit (mode d'adressage et descripteur).

Algorithme 3 : Générer(CheminGen, Op1, Op2)

Pas 1 :
Si l'opération et les opérandes ne sont pas du même type, générer les instructions de conversion correspondantes.

Pas 2 :
Allouer les ressources nécessaires au résultat de l'opération.

Pas 3 :
Transformer l'opérande gauche, si nécessaire, en générant les instructions de transfert correspondantes.

Pas 4 :
Transformer l'opérande droit, si nécessaire.

Pas 5 :
Emettre le code correspondant à la possibilité d'opération choisie.

Pas 6 :
Libérer les ressources qui ne sont plus nécessaires.

Figure VII.5. Génération de code pour une opération.

Nous remarquons que toute instruction a un type qui nous permet de distinguer les différents catégories d'instructions, en particulier les branchements qui nécessitent un traitement spécifique. En effet, un branchement n'est pas analysé par la chaine Evaluer-Générer mais plutôt par des fonctions spéciales du module Générer ; pour déterminer l'instruction exacte qui réalisera le branchement, nous devons connaitre la distance entre elle et sa cible (du fait des instructions de branchement court ou long) ; or, ceci n'est possible qu'une fois tout le code généré. En conséquence, nous nous limitons, au niveau Générer, à préparer les opérandes et à conserver les informations appropriées dans la table "CodeSymb" ; c'est l'optimiseur final qui se chargera ensuite de déterminer la meilleure instruction correspondant au branchement.

La deuxième table produite par Générer contient les

étiquettes associées aux instructions ; chaque entrée correspond à une étiquette et est constituée par 2 informations :
- le numéro de l'instruction à laquelle est associée l'étiquette,
- les reférences qui lui sont faites.

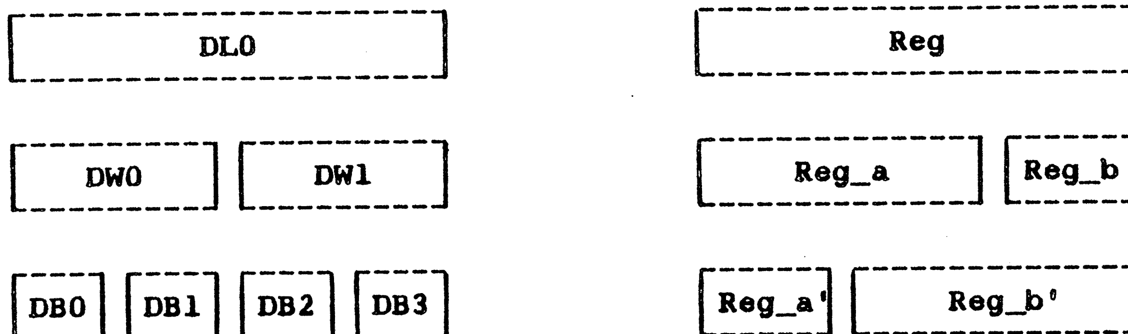
Une fonction du module Générer (ImprimerCodeGénéré) permet d'imprimer ces tables sous une forme symbolique standard, du niveau d'un langage d'assemblage ; un exemple de la sortie est présenté plus loin (cf. VII.1.6).

Nous pouvons remarquer que l'algorithme se charge d'allouer les ressources qui seront occupées par le résultat ; si l'une des ressources est occupée, la valeur qu'elle contient est sauvegardée, avant de la réallouer. D'autre part, l'algorithme libère les ressources qui ne sont plus nécessaires à la fin du traitement de l'opération ; ces ressources correspondent soit aux valeurs intermédiaires calculées pour réaliser l'opération (ou une opération précédente), soit une valeur qui n'est plus nécessaire à l'appelant et qu'il nous demande de libérer.

1.3. Gestion des registres.

Lorsqu'un constructeur présente les registres d'une machine, il indique qu'elle possède un certain nombre de registres élémentaires R0,R1,...Rn. Mais, il arrive très souvent que l'on puisse manipuler comme un registre :
- la concaténation de plusieurs registres (par exemple, le résultat d'une multiplication se trouve dans 2 registres),
- une partie d'un registre (par exemple, on peut charger un octet dans les 8 bits poids fort d'un registre 16 bits).

Le modèle que nous avons choisi pour représenter l'ensemble des registres, ayant des bits en commun, est une arborescence ; nous aurons, par exemple, la décomposition du registre DL0 du 68000 (‹Motorola80›), illustrée par la figure VII.6.a. Nous supposons donc qu'il n'y a pas 2 partitions possibles pour un registre et qu'en particulier, le découpage de la figure VII.6.b est interdit.



a) Registre donnée du 68000. b) Découpage interdit.

Figure VII.6. Modèle de représentation des registres.

1.3.1. Structure de données utilisée.

Notre gestion de registres impose essentiellement les contraintes suivantes :

a) Il faut à tout moment savoir si un registre est libre et, s'il ne l'est pas, être capable de le libérer. Nous allons montrer sur un exemple ce que cela implique.

Soient les registres décrits dans la figure VIII.7 ; les valeurs V1,V2,V3 sont contenues respectivement dans les registres DB2, DB3 et DW0. Quand on considère l'état des registres, il faut bien sûr voir que DW0, DB2 et DB3 sont occupés et qu'en conséquence, DL0, DW1, DB0 et DB1 sont aussi occupés. Donc, si on veut libérer DL0, il faut voir qu'il faut libérer DW0, DB2 et DB3 ; et si l'on veut libérer DW1, qu'il faut libérer DB2 et DB3.

b) Problèmes posés par la division et la multiplication.
Reprenons l'exemple précédent. Considérons une opération de multiplication avec un opérande dans DW1 et résultat dans DL0 ; il faut se rendre compte qu'avant de faire la multiplication, on doit libérer DW0 car il sera écrasé. Pour une division, il faudra faire l'action contraire, c'est à dire, libérer le sous-registre DW0 car il n'est plus utilisé après la division.

Nous avons donc mis en oeuvre une structure de données adaptée à ces contraintes.

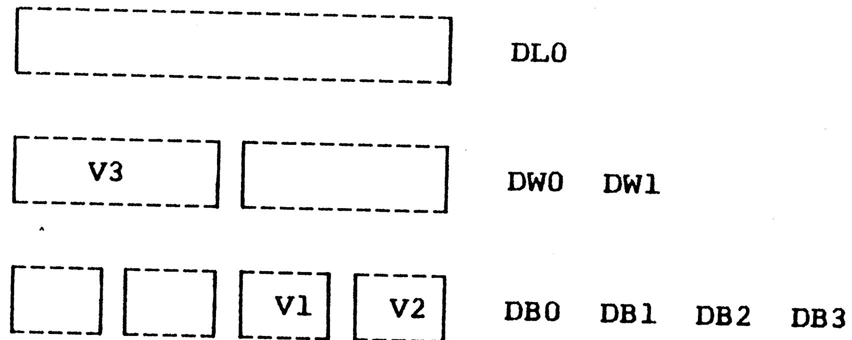


Figure VII.7. Exemple d'occupation de registres.

Notre table de registres contient l'ensemble des registres qui ont été décrits ; chaque entrée correspond à un registre et est formée par les informations suivantes :

- SousReg : liste des sous-registres du registre concerné ; un sous-registre est décrit par le triplet : nom du sous-registre, numéro du premier bit et numéro du dernier bit.
- SurReg : le père du registre concerné.
  Ces 2 informations mettent en oeuvre le modèle arborescent que nous avons choisi.
- Bornes : les numéros du premier et du dernier bit du registre.
- Etat : indique l'état courant du registre. Il peut prendre les valeurs Libre, Alloué ou Occupé.
- RefDesc : donne accès au descripteur de la valeur contenue dans le registre.
- RefReg : le registre concerné peut être occupé sans qu'il n'ait été directement alloué ; ceci est du à l'allocation effective d'un ou plusieurs de ses ascendants ou descendants ; RefReg contient la liste de ces registres.
- Eval : booléen qui indique si le registre est actuellement utilisé ou non en évaluation.

En ce qui concerne l'attribut Etat, voici la signification de ses 3 valeurs possibles :

- Libre : le registre peut etre alloué.
- Alloué : le registre a été réservé pour un certain usage. Le descripteur de la valeur qu'il contient, n'est pas connu par le module de gestion de registres ; en conséquence, elle ne pourra pas être sauvegardée, en cas de besoin, pour nous

permettre de réallouer le registre.

- Occupé : le registre contient une valeur dont le module de gestion connait le descripteur. Si le registre est nécessaire pour un autre usage, il peut donc être libéré, en sauvegardant sa valeur dans une mémoire temporaire.

1.3.2. Les algorithmes.

Nous avons vu que l'algorithme de génération de code travaille en 2 étapes. Les besoins de ces 2 étapes, en ce qui concerne la gestion de registres, sont tout à fait différents, ce qui nous conduit à séparer cette gestion en 2 parties :

- EvaluerReg, qui fait la gestion pour l'étape d'évaluation, et
- GénérerReg, qui le fait pour l'étape de génération effective de code.

La fonction principale de la première partie est EvalAllouerReg ; elle est chargée de choisir, parmi une classe donnée de registres (fixée par l'appelant), celui dont le coût d'allocation est le plus faible ; ainsi, son choix portera sur un registre à l'état Libre plutôt que sur un autre qui est Occupé. La fonction retourne comme résultat le coût d'allocation du registre choisi ainsi qu'un chemin de génération, si sa libération est nécessaire. En plus, elle marque le registre comme étant réservé, sans modifier son état précédent ; il ne sera pas choisi à nouveau par la fonction tant que l'évaluateur n'aura pas indiqué qu'il n'est plus nécessaire (par l'intermédiaire de EvalDésallouerReg) ; cette mise en disponibilité correspond soit à l'abandon (retour en arrière), soit à la fin de l'évaluation d'une possibilité.

La deuxième partie travaille sur l'état des registres, nous permettant de les faire évoluer en fonction des opérations traitées ; elle ne fait aucun choix de registre car la première partie a été appelée avant pour le faire. Elle est formée essentiellement par :

a) AllouerReg, qui permet d'allouer un registre (l'état devient Alloué).

b) SetUtilisation, qui permet de faire le lien entre un registre

et le descripteur de la valeur qu'il contient (le registre passe donc à l'état Occupé).

c) DésallouerReg, qui permet de libérer un registre (nouvel état : Libre).

1.4. Gestion des temporaires et des constantes.

Ce module gère 2 types de ressources, les temporaires et les constantes non immédiates, toutes deux implantées en mémoire.

Dans cette gestion, nous retrouvons encore une fois les étapes d'évaluation et de génération effective. Par contre, la première étape n'a plus comme objectif l'évaluation d'un coût ni le choix de quelque chose, en fonction de ce coût ; en effet, l'allocation d'une constante ou d'un temporaire a toujours un coût nul, quelle que soit la situation de l'évaluateur. La première étape se limite donc à fournir le descripteur d'un temporaire (ou d'une constante) quelconque.

C'est l'étape génération effective qui se charge complètement de l'allocation et de la libération de ces 2 types de ressources.

1.4.1. Les temporaires.

En général, une gestion en pile est suffisante pour les temporaires (‹Aho79›, ‹Gries71›). Dans notre cas, il était impossible de l'utiliser car nous avons prévu d'évaluer une seule fois les sous-expressions communes (expressions identiques qui sont à différents endroits du programme mais qui s'évaluent à la même valeur).

L'identification de ces sous-expressions est faite dans une phase précédente. Ainsi, lorsque le générateur en trouve une, il regarde d'abord si elle n'a pas encore été évaluée ; si c'est le cas, il l'évalue et bloque la ressource qui contiendra le résultat à l'exécution ; si plus tard, il retrouve ailleurs cette même expression, il ira prendre directement la valeur dans cette ressource, sans réévaluer l'expression.

Le résultat d'une sous-expression commune peut évidemment être stocké dans un temporaire ; de ce fait, la gestion des temporaires ne peut pas être faite en pile : les temporaires empilés en dessous d'une sous expression commune ne pourraient être libérés qu'après avoir traité toutes les occurrences de la sous expression. Nous avons donc choisi un autre type de gestion.

Notre gestion s'articule autour d'une table d'occupation qui indique pour chaque élément si le granule d'allocation correspondant est libre ou occupé ; un granule est une information de base de la machine (en général, un octet). Un temporaire peut être constitué d'un ou plusieurs granules, en fonction de son type.

Une table d'occupation est nécessaire pour chaque niveau de procédure. C'est l'algorithme de parcours de l'arbre syntaxique qui doit pourvoir ces tables ; il doit donc activer et initialiser la table correspondante, au début du traitement de chaque procédure, et restaurer celle du niveau inférieur, à la fin de ce traitement.

A toute table d'occupation est associée une table de temporaires, qui est la réalisation des temporaires à l'exécution du programme compilé ; cette table est placée dans la pile d'exécution du programme. Ces 2 tables sont de la même taille, donnée en paramètre dans la description de la machine cible.

L'algorithme d'allocation doit respecter essentiellement 2 contraintes lorsqu'il cherche à allouer un temporaire :

a) Que son premier granule soit compatible avec l'alignement du type désiré.

b) Que tous les granules qui le composent soient contigus.

Pour la réalisation de la première de ces contraintes, nous forçons le premier granule de la table à occuper une position qui respecte les alignements de tous les types de la machine. L'algorithme d'allocation est donné dans la figure VII.8.

Algorithme 4 : AllouerTemporaire(Type)

Pas 1 :
Calculer nb et align (respectivement le nombre de granules et l'alignement), correspondants au type de temporaire désiré.

Pas 2 :
Tant que l'allocation n'est pas faite et qu'il reste des éléments de la table à examiner, faire :
a) Chercher un granule libre dont la position est compatible avec align.
b) Vérifier si les nb-1 granules suivants sont libres.
c) Si oui, allouer ces nb granules, allocation, ainsi que celle de lib, en les marquant comme étant occupés.

Pas 3 :
Retourner le descripteur du temporaire alloué.

Figure VII.8. Allocation d'un temporaire.

Il faut remarquer qu'une demande d'allocation peut échouer, faute de place pour le temporaire demandé ; dans ce cas, la génération de code est abandonnée car les demandes se font toujours au niveau génération effective et aucun retour en arrière n'est possible. Pour résoudre ce problème, on ne peut que reconstruire le générateur de code, en prenant une taille plus grande pour la table de temporaires.

La libération d'un temporaire est très simple ; elle consiste à faire passer à l'état libre tous les granules qui forment le temporaire.

1.4.2. Les constantes non immédiates.

Toutes les constantes du programme compilé, qui ne peuvent pas être utilisées en tant qu'opérandes immédiats, sont mises par le générateur dans une table dite des constantes, par l'intermédiaire du module décrit ci-dessous.

La table des constantes est unique ; elle aussi est placée dans la pile d'exécution du programme juste après les variables et les temporaires de l'environ global, ce qui nous permet de ne pas limiter sa taille.

La gestion de cette table est très simple étant donné qu'il n'y a jamais de libérations de constantes. Il nous suffit donc d'avoir une variable qui indique en permanence la première position libre dans la table ; une allocation consistera alors à réserver la place mémoire, nécessaire à la constante, à l'endroit pointé par cette variable et à mettre à jour ce pointeur. Nous disposons en outre d'un modèle de descripteur de la première position de la table, qui sert à former les descripteurs des constantes allouées.

Dans cette gestion, comme dans celle des temporaires, nous assurons toujours l'alignement correspondant au type de la constante traitée. Il peut donc rester des "trous" dans la table de constantes ; l'espace perdu est cependant très faible en raison du petit nombre de constantes non immédiates rencontré dans un programme.

En plus de l'allocation, le module réalise une deuxième fonction : l'initialisation des constantes allouées. Comme nous ne générons pas du code binaire, nous nous contentons de remplir une table interne avec la valeur et le type de la constante ; ces informations permettent à une phase postérieure (d'assemblage) de faire la réservation et les initialisations correspondantes.

1.5. La gestion des descripteurs.

Notre méthode de génération de code est basée sur la notion de descripteur de valeur ; toute valeur du programme compilé est donc associée à un descripteur, avant d'être manipulée par le générateur. C'est l'allocateur des variables qui est chargée de faire cette association pour les variables et les constantes du programme ; d'autre part, le générateur construit toujours un descripteur pour le résultat de l'opération traitée.

Le module de gestion de descripteurs fournit un ensemble de fonctions qui permettent une manipulation aisée de ces descripteurs ; en particulier, il facilite la communication entre les différents modules du générateur et offre un système

de verrouillage des descripteurs.

Durant le parcours de l'arbre syntaxique du programme, tout descripteur rencontré est rangé dans la table des descripteurs avant que toute génération l'utilisant ne soit lancée. A cet instant, un numéro, qui sert à le désigner à l'intérieur du générateur, lui est assigné ; il est conservé dans la table jusqu'à ce que sa valeur ne soit plus nécéssaire.

Pour les mêmes raisons que dans la gestion des temporaires, la table des descripteurs ne peut être gérée en pile. Nous avons là aussi choisi d'utiliser une table d'occupation, où chaque élément nous indique si l'entrée de même indice dans la table des descripteurs est libre ou occupée ; une allocation revient donc à faire un parcours linéaire de cette table d'occupation et à prendre la première position libre.

Les informations stockées dans la table de descripteurs sont :

a) Le descripteur lui-même.
b) Les ressources qu'il occupe et qui n'ont été allouées que pour lui.
c) Un verrou.

Cette dernière information permet d'interdire la libération du descripteur et de ses ressources par d'autres modules, ainsi que la modification de sa valeur ; c'est celui qui fait le verrouillage qui levera la restriction lorsqu'elle ne sera plus nécessaire. Ce mécanisme nous permet de traiter assez facilement une sous-expression commune ; en effet, après avoir traité sa première occurrence, nous verrouillons le descripteur de son résultat jusqu'à ce que l'on traite sa dernière occurrence.

Dans la plupart des cas, une entrée de la table n'est que consultée. Cependant, dans certains cas nous avons besoin de modifier son contenu ; ceci est nécessaire, par exemple, lorsqu'une préemption de registre est faite par le module correspondant. Ce type d'opération est réalisé par une fonction qui permet de remplacer le descripteur correspondant ainsi que ses ressources.

1.6. Exemple d'utilisation du noyau.

Nous faisons un ensemble d'hypothèses sur l'environnement qui entoure le noyau, en vue de simplifier la présentation et la compréhension de l'exemple. Ces hypothèses sont les suivantes :

a) L'allocation des variables a déjà été faite ; en conséquence, une décision concernant l'adressage de ces variables a été prise et on dispose de leurs descripteurs.

b) Nous effectuons un parcours classique en compilation de l'arbre du programme (‹Gries71›) ; nous traitons d'abord le fils droit, ensuite le fils gauche et finalement le père.

c) Le noyau est appelé seulement lorsque tous les opérandes du noeud visité sont réduits à des feuilles.

d) La machine cible utilisée pour les exemples est le 68000 dont la description est donnée dans l'annexe B.

e) D'autres hypothèses moins importantes concernant l'adressage des temporaires, la libération des ressources, la gestion des résultats intermédiaires.

Dans la suite nous présentons la façon dont est traitée une expression PASCAL par le générateur ainsi que les résultats obtenus.

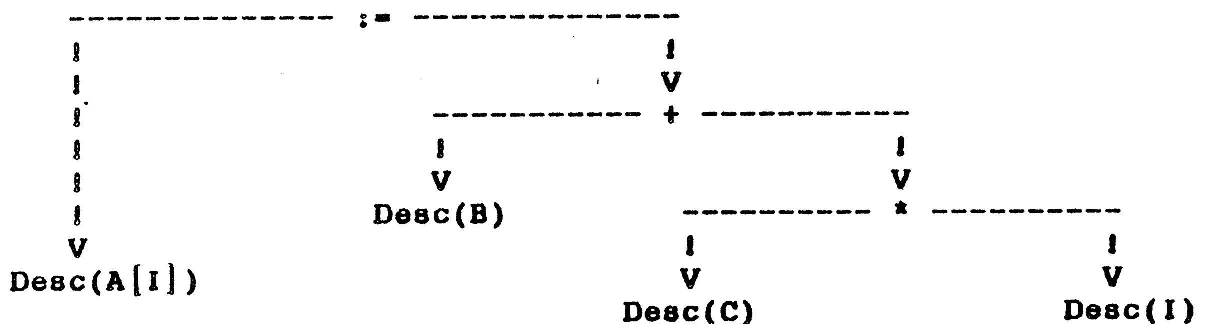


Figure VII.9. Représentation intermédiaire d'une instruction.

Soit l'instruction Pascal

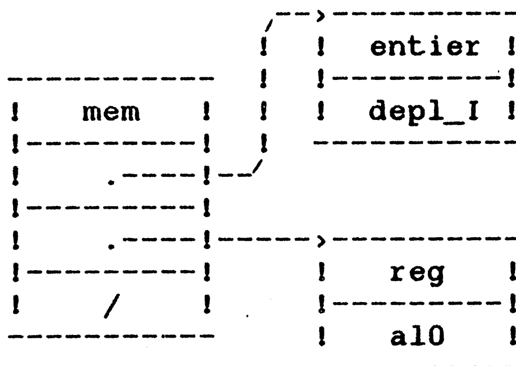
A[I] := B + C * I

où I, B et C sont des variables entières et A un tableau d'entiers ; tous ces objets sont locaux à l'environ courant et

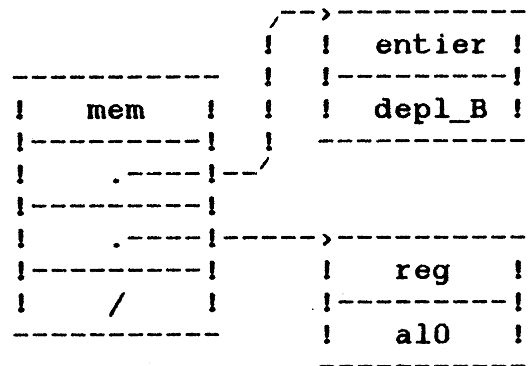
sont basés par le registre a0. Les descripteurs de ces variables sont ceux de la figure VII.10 ; d'autre part, la représentation interne de l'instruction est illustrée par la figure VII.9.

Suivant le parcours choisi, la première opération à générer est la multiplication entre C et I. Avec le descripteur résultat qui est rendu par le noyau, nous traitons l'addition entre celui-ci et B, et ainsi de suite. Les appels du noyau sont les suivants :

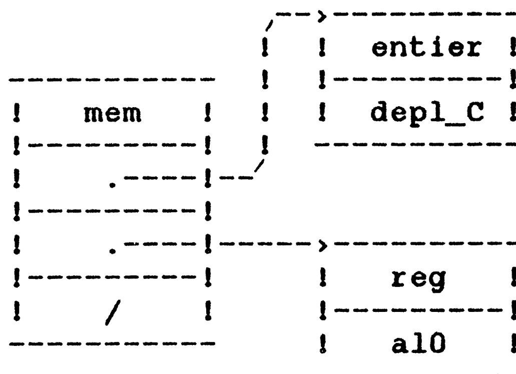
a) Génération('*', Desc(C), Desc(I))
b) Génération('+', Desc(B), Desc(résultat a))
c) Génération(':=', Desc(A[I]), Desc(résultat b))



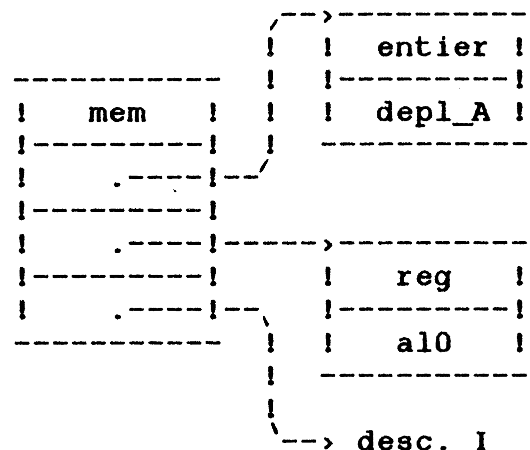
a) Descripteur I



b) Descripteur B



c) Descripteur C.



d) Descripteur A[I].

Figure VII.10. Descripteurs de variables Pascal.

Une fois terminés tous ces appels et en supposant qu'il s'agit des premiers appels, le résultat de ImprimerCodeGénéré serait :

```
‹ multiplication
‹
          move  d0,depl_C(a0)
          muls  d0,depl_I(a0)
‹ le résultat est dans  le registre long d0
‹
‹ ensuite on fait l'addition
          add   d0,depl_B(a0)
‹ et puis on fait l'affectation
          move  a1,depl_I(a0)
          move  depl_A(a0,a1),d0
```

Dans cette séquence, nous pouvons remarquer plusieurs choses :

a) Le code généré pour chaque opération est optimal ; en effet, les instructions utilisées sont toujours les moins coûteuses, de même que les modes d'adressage.

b) Pour réaliser la multiplication, le noyau n'a pas trouvé une instruction qui réalise cette opération pour 2 opérandes mémoire ; elle a donc transformé l'un des opérandes. La transformation retenue c'est un chargement dans un registre, lequel servira également à garder le résultat.

c) Le registre d0 a dû être alloué pour faire la transformation mentionnée en b) ; il était libre avant, d'où aucune séquence de sauvegarde de sa valeur.

d) Le résultat de la multiplication est obtenu dans le registre long d0. Ce résultat doit être utilisé pour faire l'addition mais il s'agit d'un opérande 32 bits alors que l'opération ne travaille qu'avec des opérandes 16 bits ; le noyau a donc fait une conversion 32 -› 16 bits qui, dans notre cas, revient à utiliser la partie poids faible du registre.

e) Pour faire l'addition, la propriété de commutativité a été utilisée ; en effet, B qui était l'opérande gauche a été utilisé comme opérande droit.

f) L'indice I du tableau a été chargé dans le registre a1 pour mettre en oeuvre l'accès au tableau par l'intermédiaire d'un mode d'adressage de la machine (l'indexation). Remarquons que seul les registres ai peuvent être utilisés comme index ; a0 n'a pas été pris parce qu'il était déjà occupé (base de notre

environ).

2. Compilation des expressions.

Cette section traite de la compilation des différents types d'expression Pascal. Nous présentons d'abord le traitement des expressions d'accès à une donnée du programme ; nous décrivons ensuite la traduction des opérations arithmétiques et ensemblistes, que nous regroupons sous le nom d'expressions arithmétiques ; nous exposons enfin le traitement des expressions logiques.

2.1. Les expressions d'accès.

Une expression d'accès correspond à la désignation d'un objet ou d'un composant d'un objet. Elle est représentée par un sous-arbre dont les noeuds sont soit des constantes, soit des identificateurs (de variable ou de constante), soit des opérateurs d'accès à un élément d'une structure :
- indirection à travers un pointeur (↑),
- champ d'un enregistrement (.),
- élément d'un tableau ([ ]).

La figure VII.11 donne quelques exemples d'expressions d'accès.

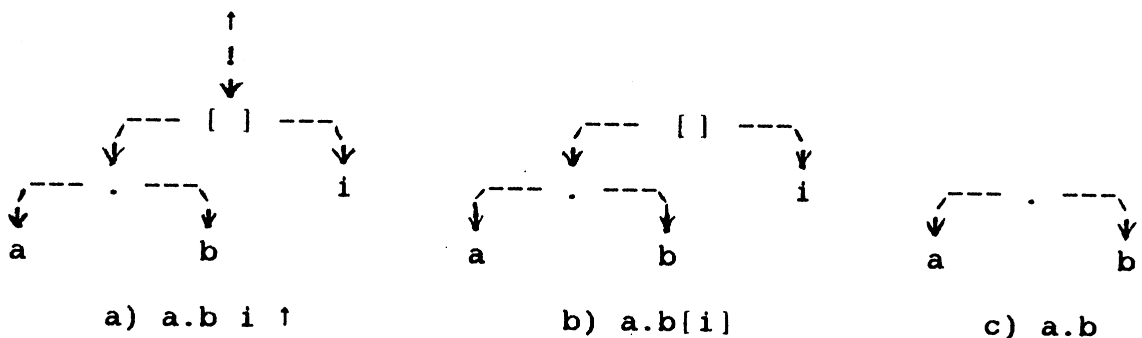


Figure VII.11 Exemples d'expression d'accès.

Le traitement d'une expression d'accès consiste à construire le descripteur correspondant à la donnée désignée. La génération des instructions d'accès à cette donnée est différée jusqu'au traitement de l'opération qui l'utilise ; de cette façon, le noyau peut choisir le mode d'adressage qui convient le mieux à

cette opération. Aucune vérification dynamique n'est générée en ce qui concerne les types simples, les parties variantes des enregistrements et les tableaux.

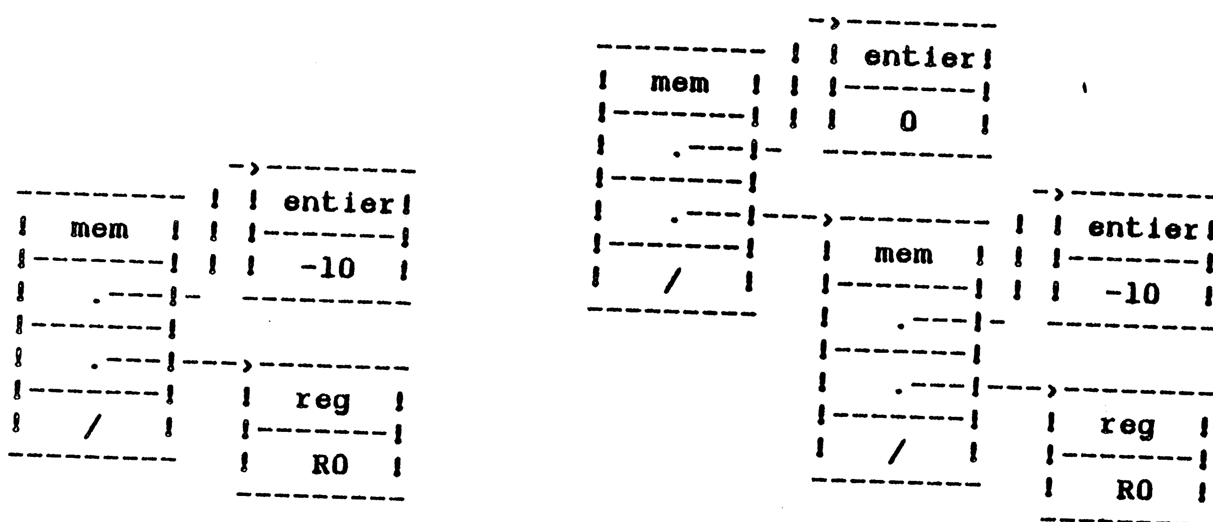
2.1.1. Accès simple.

Un accès simple correspond à la désignation d'un objet en entier ; celui-ci peut être aussi bien un objet simple qu'un objet structuré.

La construction du descripteur correspondant est triviale. S'il s'agit d'une variable, nous adaptons le descripteur qui lui a été associé par l'allocateur au contexte d'adressage de la procédure compilée. S'il s'agit d'une constante, nous lui associons un descripteur de constante immédiate ; c'est le noyau qui fera la transformation en position mémoire si cela s'avère nécessaire.

2.1.2. Indirection à travers un pointeur.

Cette opération correspond à l'accès à une donnée du tas, par l'intermédiaire d'un pointeur ; ce dernier constitue l'opérande unique de l'expression et est lui-même désigné par une expression d'accès.



a) Descripteur de p

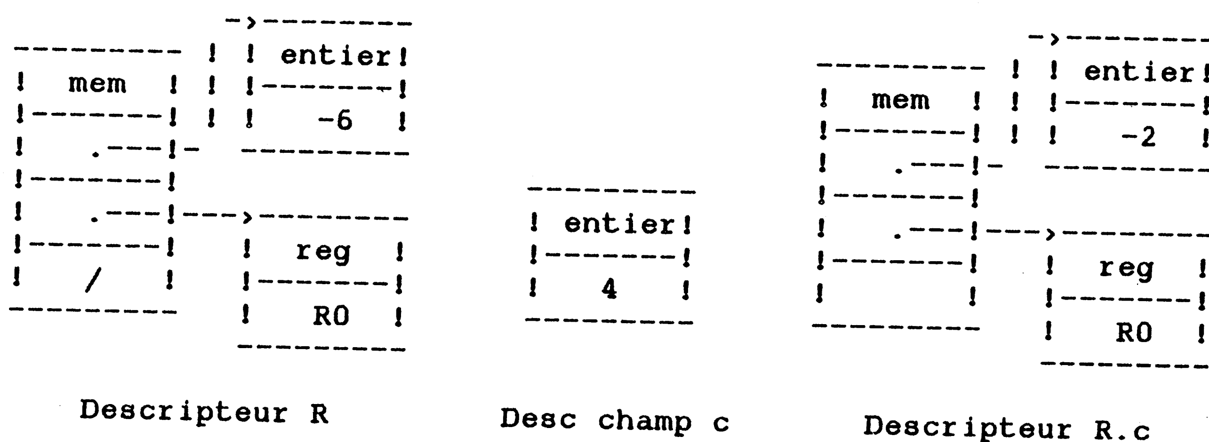
b) Descripteur de p↑

Figure VII.12 Traitement de l'indirection à travers un pointeur.

Le traitement de cette opération se déroule en 2 étapes. D'abord, nous compilons son opérande façon à obtenir le descripteur du pointeur correspondant. Ensuite, nous construisons un deuxième descripteur qui représente l'indirection via ce pointeur ; il s'agit d'un descripteur mémoire dont la base est le pointeur et dont le déplacement et l'index sont nuls. La figure VII.12 donne un exemple de ce traitement.

2.1.3. Champ d'un enregistrement.

Le traitement de cette opération s'effectue en 2 étapes. D'abord, on évalue ses 2 opérandes, de façon à obtenir leurs descripteurs ; le premier correspond au descripteur de l'enregistrement tandis que le deuxième représente le déplacement du champ par rapport au début de l'enregistrement. Nous faisons ensuite l'union de ces descripteurs ; d'une part, nous rajoutons le déplacement du champ à celui de l'enregistrement et, d'autre part, nous conservons la base et le déplacement de ce dernier. Ce traitement est illustré par la figure VII.13.



Descripteur R   Desc champ c   Descripteur R.c

Figure VII.13 Traitement de l'accès à un champ d'enregistrement.

2.1.4. Elément d'un tableau.

Cette opération correspond à l'accès à un élément de tableau, au moyen de l'indexation. Son premier opérande est une

expression d'accès qui désigne le tableau ; tous les autres opérandes sont des expressions arithmétiques correspondant aux indices de l'élément adressé.

Le traitement de cette opération est réalisé en 3 étapes. En premier lieu, nous évaluons le premier opérande de façon à recupérer le descripteur du tableau ; ce descripteur contient entre autres, l'origine virtuelle du tableau et l'enjambée de chacune de ses dimensions. Nous calculons ensuite la distance de l'élément par rapport à l'origine virtuelle du tableau (son index), distance définie par la formule suivante :

$$\sum_{dim=1}^{nb.\ indices} enjamb\acute{e}e(dim) * indice(dim)$$

Pour ce faire, nous évaluons d'abord les expressions qui correspondent aux indices ; puis, nous générons les instructions nécessaires au calcul <u>dynamique</u> de cette formule. Si un indice est une constante, les calculs correspondants sont effectués statiquement ; en conséquence, si tous les indices sont des constantes, aucune instruction n'est générée.

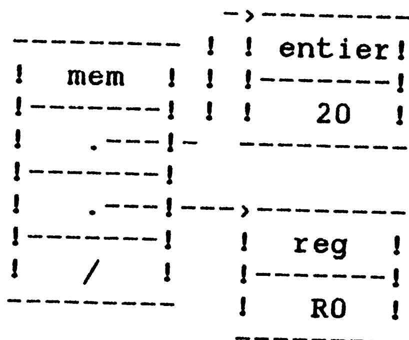
L'étape suivante consiste à construire le descripteur de l'élément désigné, en fonction des informations calculées :
- sa base est celle du tableau,
- son déplacement est obtenu en retranchant l'origine virtuelle du déplacement du tableau,
- son index est le résultat du calcul décrit ci-dessus ; toutefois, si le descripteur du tableau contient lui-même un index (conséquence d'une évaluation antérieure), celui-ci est rajouté à ce résultat,
- son type est celui des éléments du tableau.

La désignation d'un élément de tableau est la seule opération d'accès qui peut donner lieu à une génération de code immédiate. Les instructions générées n'ont toutefois aucune incidence sur le choix du mode d'adressage qui sera utilisé pour adresser la donnée ; c'est toujours le noyau qui fait ce choix et qui génère les instructions d'accès à la donnée.

La figure VII.14 donne un exemple du traitement de cette opération pour une machine hypothétique. La déclaration et l'opération d'accès à la donnée sont respectivement :

T : array[4..10] of integer ; ... T[i]

On a supposé dans l'exemple qu'un entier occupe 2 octets et qu'il n'existe aucun problème d'alignement ; d'autre part, nous n'avons pas utilisé de décalages pour réaliser la multiplication, dans un souci de clarté.

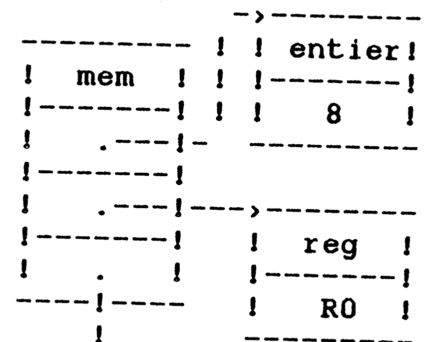


orig. virtuelle = 12
enjambée dim. 1 = 2

Descripteur T

Instructions générées

```
< Charger enjambée
   MOVE  R1, 2
< Calculer index
   MULT  R1,i
```




```
   !
   !
   V
---------
! reg   !
!-------!
!  R1   !
---------
```

Descripteur T[i]

Figure VII.14 Traitement de l'accès à un élément de tableau.

2.2. Les expressions arithmétiques.

Nous regroupons dans ce paragraphe les expressions arithmétiques et ensemblistes pour des raisons pragmatiques, liées essentiellement aux similitudes entre leurs traitements. Dans notre représentation, une expression arithmétique est un sous-arbre dont la racine est un opérateur arithmétique et les fils les opérandes correspondants. Un fils de ces opérations peut être :

- une autre expression arithmétique,
- une expression d'accès,
- ou un appel de fonction.

Dans ce qui suit, nous présentons le traitement réalisé sur

ces expressions.

2.2.1. Traitement d'une sous-expression commune.

Une sous-expression commune possède un attribut qui lui a été rajouté par l'optimiseur global lors de son identification ; cet attribut permet sa reconnaissance et le relie à toutes les occurrences de la même sous-expression.

Durant le parcours de génération, la rencontre d'une sous-expression commune provoque un traitement particulier. S'il s'agit de sa première occurrence, elle est compilée comme une expression quelconque mais le descripteur du résultat est sauvegardé et verrouillé, en vue d'une réutilisation postérieure. Toutes les autres occurrences sont compilées comme un simple accès à cette valeur ; le descripteur est déverrouillé, et les ressources correspondantes libérées, après traitement de la dernière occurrence.

Le résultat d'une sous-expression commune peut avoir été rangé dans un registre ; dans ce cas, si l'on a besoin du registre avant que toutes les occurrences aient été traitées, celui-ci peut être récupéré ; son contenu est alors transféré dans un temporaire et le descripteur remplacé en conséquence.

2.2.2. Les opérateurs arithmétiques.

Nous traitons toujours ces opérateurs comme s'ils étaient binaires, pour uniformiser leur traitement ; si l'opérateur est unaire, un opérande nul lui est alors rajouté.

Le traitement d'une opération est effectué en 2 étapes. Nous évaluons d'abord les opérandes, de façon à obtenir leurs descripteurs ; l'opérande droit est traité le premier, dans le but de minimiser le nombre de temporaires nécessaires au calcul de l'expression (‹Gries71›). Ensuite, une demande de génération, constituée de ces descripteurs et de l'opération, est soumise au noyau ; celui-ci génère le code correspondant et rend le descripteur du résultat.

Si le résultat est dans un registre, nous indiquons à la gestion de registres où se trouve le descripteur correspondant ; de cette façon, si le registre est nécessaire ailleurs, il pourra être recupéré après que l'on ait sauvegardé sa valeur dans un temporaire.

2.2.3. Les opérateurs ensemblistes.

Ces opérateurs sont au nombre de 4 : l'union ("+"), la différence ("-"), l'intersection ("*") et l'affectation. La représentation choisie pour les ensembles pose un certain nombre de problèmes dans le traitement de ces opérateurs ; en particulier :

a) Un ensemble peut occuper plusieurs mots machine ; par contre, les instructions logiques, utilisées pour traduire ces opérations, ne peuvent pas en général être appliquées au delà du mot. Donc, il est parfois nécessaire de découper l'ensemble et d'appliquer l'opération à chacune de ses parties.
b) La représentation de 2 ensembles <u>compatibles</u> n'est pas la même, s'ils sont définis sur des intervalles différents (figure VII.15) ; ceci est dû au fait que nous associons toujours le bit 0 de cette représentation à la valeur inférieure de l'intervalle de définition. Par conséquent, une opération entre ces ensembles implique le décalage de l'un des deux, de façon à aligner leurs représentations ; c'est seulement après ce décalage que l'opération pourra être réalisée.

Le traitement de ces opérations n'est par conséquent pas très simple, surtout si l'on considère que nous voulons générer du code optimisé. Nous avons développé un algorithme pour résoudre ces problèmes ; cet algorithme est présenté dans ‹Santana83›.

2.2.4. Le constructeur d'ensembles.

Cet opérateur permet de construire et d'initialiser un ensemble. Ses opérandes sont les valeurs qui serviront à cette

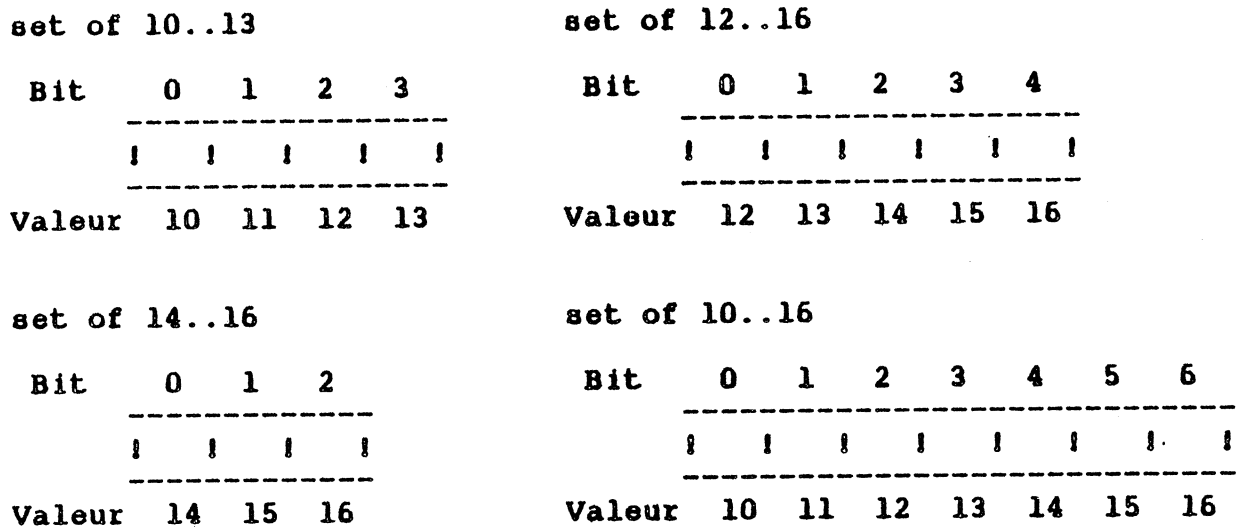


Figure VII.15 Représentations de 4 ensembles compatibles.

initialisation ; ils seront par conséquent compilés avant de traiter l'opérateur.

La construction de l'ensemble est faite statiquement si toutes les valeurs initiales sont des constantes ; l'opérateur est alors compilé comme une référence à la constante correspondante. Autrement, la construction est réalisée dynamiquement ; pour ce faire, nous allouons un temporaire qui représente l'ensemble, et nous générons les instructions nécessaires à son initialisation (instructions du type "mettre à 1 le bit i") ; la partie constante peut être construite statiquement et ensuite faire l'objet d'une union avec la partie dynamique, si cela s'avère moins coûteux.

La représentation utilisée pour l'ensemble est déterminée en fonction du contexte de l'opération. En effet, étant donné que cet ensemble sera utilisé comme opérande d'une autre opération ensembliste, nous essayons de lui associer la représentation la plus appropriée à cette opération ; ainsi, aucun décalage ne sera nécessaire pour l'aligner avec le deuxième opérande.

Le contexte de l'opération peut également donner lieu à une certaine classe d'optimisations. En effet, dans certains cas, il est possible de remplacer la construction de l'ensemble <u>et</u> l'opération qui utilise son résultat par une opération équivalente ; par exemple, l'expression

EnsembleX + i

peut être remplacée par l'opération "mettre à 1 le bit i de EnsembleX". Dans ce cas, nous évaluons les deux possibilités et nous utilisons la moins coûteuse.

2.3. Les expressions logiques.

Une expression logique est un sous-arbre dont l'évaluation donne comme résultat une valeur booléenne ; la racine d'une telle expression est par conséquent une comparaison, une opération logique ("and", "or" ou "not") ou bien une référence à un objet de type booléen.

Une caractéristique de ce type d'expressions est que leur valeur peut être obtenue sans avoir évalué toute l'expression ; par exemple, si l'évaluation du premier opérande d'un "or" nous donne la valeur "vrai", nous savons d'ores et déjà que le résultat de l'opération sera lui-aussi "vrai". La définition de Pascal permet d'exploiter cette caractéristique ; en effet, elle spécifie que les opérandes d'une expression peuvent ne pas être tous évalués.

Notre réalisation tire profit de cette propriété ; en effet, dès que la valeur d'une expression est connue, cette valeur est rendue sans que l'on évalue le reste de l'expression. Un effet secondaire de cette approche est qu'une expression logique ne peut jamais être considérée en tant que sous-expression commune.

Nous présentons dans ce qui suit les différents aspects du traitement d'une expression logique.

2.3.1. Contextes d'évaluation.

Une expression logique peut apparaître dans 2 contextes différents : en partie droite d'une affectation ou dans la partie condition d'une instruction de contrôle. La traduction de l'expression dépend de ce contexte ; dans le premier cas, il faut obtenir une valeur tandis que dans le second, il faut positionner un code de condition puis générer un branchement en

fonction de la valeur du code.

Le résultat nécessaire au contexte ne peut pas toujours être obtenu directement, avec les instructions disponibles sur la machine cible ; par exemple, pour l'instruction Pascal

Positif := Valeur > 0 ;

la comparaison sera en général réalisée par une instruction qui positionne le code de condition, alors que nous avons besoin d'une valeur booléenne. Il est donc nécessaire dans ces cas de transformer le résultat ; ainsi, dans notre exemple, il faudra allouer un registre ou un temporaire et lui affecter la valeur "vrai" ou "faux", suivant le contenu du code de condition.

Etant donné que ces transformations peuvent se produire assez souvent, il est intéressant de les réduire au strict minimum ; pour ce faire, nous effectuons une analyse globale de l'expression avant de faire sa traduction.

2.3.2. Analyse d'une expression.

Cette analyse nous permet de déterminer le meilleur ordre d'évaluation d'une expression logique, pour un contexte donné, et de minimiser ainsi le nombre de transformations de résultats intermédiaires.

L'analyse est effectuée en un seul parcours postfixé de l'expression. Durant ce parcours, nous calculons pour chaque noeud le coût de son évaluation dans chacun des 2 contextes ainsi que le coût de son éventuelle transformation, compte tenu de son contexte effectif ; nous construisons parallèlement un arbre décrivant les choix réalisés ainsi que toutes les transformations nécessaires.

2.3.3. Traitement des comparaisons.

Les comparaisons sont traitées suivant le type de leurs opérandes ; nous distinguons à ce niveau 3 classes de types :

- simples (types prédéfinis, scalaires, intervalles et pointeurs),

- ensembles,
- structurés (chaînes de caractères).

Dans tous les cas, les opérandes sont compilés avant l'opération elle-même. Comme il s'agit d'expressions arithmétiques, cette compilation est réalisée par le module correspondant à ces expressions.

Le traitement d'une comparaison simple se déroule en 2 étapes. Dans la première étape, on demande au noyau de générer les instructions qui correspondent à l'opération elle-même ; puis, on génère un branchement conditionnel qui exploite le résultat de la comparaison. Dans la deuxième étape, on génère les instructions de transformation du code de condition en valeur booléenne, si cette transformation est nécessaire ; si la machine dispose d'une instruction d'affectation directe du code de condition (par exemple : SCC du 68000), nous utilisons cette instruction pour faire la transformation ; sinon, il faut la simuler à l'aide de branchements conditionnels.

La comparaison d'ensembles pose les mêmes problèmes que les opérations ensemblistes (cf. VII.2.2.3). Il est nécessaire par conséquent de découper les 2 ensembles en parts identiques (en taille aussi bien qu'en disposition de valeurs), pour pouvoir les traiter ; une fois réalisé ce découpage, on applique à chaque partie un traitement similaire à celui des comparaisons simples. L'algorithme utilisé est présenté dans <Santana83>.

En ce qui concerne les valeurs structurées, certaines machines disposent d'instructions qui permettent de les comparer comme un tout. Si c'est le cas de la machine traitée, nous utilisons ces instructions pour traduire leurs comparaisons ; dans le cas contraire, nous générons une boucle de comparaison mot à mot, qui se termine dès que le résultat est connu. Dans les 2 cas, la traduction est complétée par une séquence de transformation du résultat, s'il faut que celui-ci soit une valeur.

2.3.4. Traitement des opérations logiques.

Les opérations logiques ("or", "and" et "not") peuvent être traduites de 2 façons différentes :

a) En utilisant les instructions logiques correspondantes de la machine ; l'évaluation des opérandes doit par conséquent produire des valeurs pouvant être utilisées par ces instrudions. Le résultat de l'opération sera lui-même une valeur, contenue dans une ressource classique.

b) En décomposant l'opération à l'aide de comparaisons et de branchements conditionnels. Ainsi, par exemple, l'expression "a = b or bool" serait décomposée de la façon suivante :

```
          Comparaison entre a et b
          Branch. à EtVrai si a = b
          Comparaison entre bool et "vrai"
          Branch. à EtFaux si bool <> "vrai"
EtVrai:   .                                    < Code à exécuter
          .                                    < si le résultat
          .                                    < est vrai
EtFaux:   .                                    < Code à exécuter
          .                                    < si le résultat
          .                                    < est faux
```

Dans ce cas, aucune valeur n'est obtenue en résultat ; par contre, l'exécution sera aiguillée sur une étiquette (EtVrai ou EtFaux), selon le résultat de l'expression.

C'est la passe d'analyse qui se charge de choisir la traduction la plus appropriée, en fonction du contexte d'évaluation de l'expression. Le traitement d'une opération logique consiste donc à présenter au noyau les demandes de génération correspondant aux choix réalisés par l'analyse.

3. Compilation des instructions.

Dans cette section, nous traitons de la compilation des différentes instructions Pascal. Chaque instruction est traitée de manière ad-hoc, dans l'objectif d'optimiser sa traduction ; nous essayons en particulier d'utiliser les instructions spécialisées de la machine qui peuvent contribuer à cet objectif.

Dans ce qui suit, nous présentons les différents types de traitements effectués. Nous décrivons d'abord celui de l'affectation d'objets simples ainsi que d'objets structurés ; ensuite, nous discutons de la traduction des instructions de contrôle ; nous présentons enfin la traduction de l'appel de procédure ainsi que le prologue et l'épilogue utilisés dans toute procédure.

3.1. L'affectation.

3.1.1. Affectation d'une valeur simple ou d'un ensemble.

Nous traitons ce type d'affectation comme une expression ; nous faisons appel par conséquent au traducteur d'expressions pour effectuer sa compilation. A ce niveau, nous nous limitons à préparer les paramètres nécessaires à l'appel de ce traducteur et à libérer les ressources qu'il a pu allouer durant le traitement.

S'il s'agit d'une affectation booléenne, l'expression logique qui constitue sa partie droite est évaluée dans un contexte valeur et le résultat affecté directement à la variable qui se trouve en partie gauche.

3.1.2. Affectation d'une structure.

La plupart de machines disposent d'une instruction qui permet de recopier toute une zone mémoire ; si cette instruction existe dans la machine traitée, elle peut être utilisée pour traduire l'affectation de structures. Le traitement consiste alors à calculer la taille et les adresses des structures impliquées et à demander au noyau de générer l'instruction ou les instructions correspondantes. Il faut remarquer qu'en général ces instructions effectuent la recopie mot par mot ; en conséquence, si les structures n'occupent pas un nombre entier de mots ou ne sont pas alignées sur une frontière de mot, nous devons générer des instructions complémentaires qui se chargeront du transfert des extrémités des structures.

Si la machine cible ne dispose pas de ce type d'instructions, nous les simulons en générant une boucle d'affectation mot par mot.

3.2. Les instructions de contrôle.

Contrairement aux affectations, on trouve rarement une instruction machine permettant de compiler directement une instruction de contrôle ; il est par conséquent nécessaire de décomposer l'instruction à l'aide d'étiquettes et de branchements, lors de sa compilation.

D'autre part, les instructions machine destinées à faciliter cette décomposition sont difficilement utilisables par un générateur comme le notre ; en effet, ces instructions varient fortement d'une machine à l'autre, ce qui fait de chacune de ces instructions un cas particulier à analyser. De ce fait, nous avons décidé de ne prévoir que le traitement des instructions machine les plus répandues ; toutes les autres instructions sont ignorées.

Nous présentons ci-après la décomposition effectuée pour chaque type d'instruction de contrôle.

3.2.1. Instructions conditionnelles.

Nous nous intéressons ici essentiellement à l'instruction "case" car son traitement peut donner lieu à de multiples optimisations ; pour sa part, l'instruction "if" demande un traitement très simple, illustré par la figure VII.16.

L'instruction "case" est composée de 2 parties distinctes. La première correspond aux instructions qui font l'objet du choix (les actions) tandis que la deuxième concerne le choix lui-même (la sélection). Les actions sont compilées en fonction de leur nature et sont toujours suivies par un branchement inconditionnel vers l'instruction qui suit le "case" ; leur traduction ne pose donc aucun problème particulier. Par contre,

```
if cond then instr1
else instr2

   évaluer cond
   branch. à Et2 si faux
Et1: évaluer instr1
   branch. à Et3
Et2: évaluer instr2
Et3: ...
```

a) if_then_else

```
if cond then instr

   évaluer cond
   branch. à Et2 si faux
Et1: évaluer instr
Et2: ...
```

b) if_then

```
while cond do instr

Et1: évaluer cond
     branch. à Et2 si faux
     évaluer instr
     branch. à Et1
Et2: ...
```

c) while

```
repeat instr until cond

Et1: évaluer instr
     évaluer cond
     branch. à Et1 si faux
     ...
```

d) repeat

Figure VII.16 Schéma de traduction de quelques instructions Pascal

la sélection peut être traduite de façon très différente :

a) En une séquence linéaire de comparaisons entre l'indice et les constantes du "case" ; chaque comparaison est alors suivie d'un branchement conditionnel vers l'action associée à la constante.
b) En un branchement indexé par une table de branchements, dans laquelle chaque branchement a pour cible l'action correspondant à son indice dans la table.
c) En un ensemble de comparaisons, organisé de façon arborescente ; chaque comparaison est un noeud de l'arbre permettant de choisir entre 2 intervalles de valeurs. Cette technique est similaire à celle des B-arbres.
d) En utilisant une combinaison des 2 techniques précédentes.
e) En organisant les constantes du "case" et les adresses de leurs actions respectives, en une table qui sera parcourue par un algorithme de recherche approprié.

L'espace mémoire et le temps d'exécution demandés par chacune de ces méthodes peuvent être très différents, suivant le nombre et la dispersion des constantes du "case" ; chaque méthode s'adapte mieux à certaines circonstances (<Sale81>).

Dans notre générateur, nous essayons de traduire chaque "case" par la méthode qui lui convient le mieux ; le choix

s'effectue entre les 4 premières méthodes présentées ci-dessus en vue de minimiser l'espace occupé par le code généré. Pour faire ce choix, nous utilisons un algorithme basé sur celui de Hennessy et Mendelsohn (‹Hennessy82›), adapté de façon à prendre en compte la première méthode ; d'autre part, si le type de l'indice du "case" n'est composé que de 2 valeurs ("boolean", par exemple), l'instruction est traduite comme un "if" (‹Atkinson82›).

3.2.2. Instructions repétitives.

La traduction des instructions "while" et "repeat" est assez simple et ne donne lieu à aucune optimisation ; les schémas de traduction correspondants sont présentés dans la figure VII.16.

```
      évaluer expi
      var := valeur expi          < init. variable de la boucle
      évaluer expf
      tempo := valeur expf        < garder la valeur de expf
                                  <   dans un temporaire
Et1:  comparer var et tempo
      branch à Et2 si var>tempo   < test fin de boucle

      évaluer instr               < corps de la boucle

      incrémenter var de 1        < incrémenter la variable
      branch à Et1                < et recommencer la boucle
Et2:  ...
```

Figure VII.17 Schéma de traduction de l'instruction "for"

L'instruction "for" est par contre plus intéressante car elle peut être traduite de multiples façons ; la figure VII.17 montre le schéma général de traduction de cette instruction, pour une itération croissante ("for-to"). Plusieurs opportunités d'optimisation se présentent quand l'expression initiale ou l'expression finale d'un "for" ont des valeurs constantes :

a) Si toutes les 2 sont constantes et si la boucle s'exécute au moins une fois, il n'est pas nécessaire de faire la comparaison et le test de fin de boucle la première fois. Dans ce cas, nous générons ces 2 opérations en fin de boucle, à la place du branchement inconditionnel ; l'étiquette du branchement conditionnel devient alors celle qui suit la

boucle.

b) Si les 2 valeurs sont constantes et égales, la boucle ne s'exécutera qu'une seule fois ; dans ce cas, il suffit de générer l'initialisation de la variable ainsi que le corps de la boucle. Si la boucle ne s'exécute jamais, elle est supprimée intégralement. Ces 2 cas peuvent apparaître comme une conséquence d'autres optimisations, en général globales.

c) Si la valeur finale est constante et égale à zéro, il est possible d'utiliser les instructions de type "décrémenter (ou incrémenter) et branchement si zéro" (par exemple, JDX et JIX du Solar16). Si la machine possède ce type d'instructions, la traduction devient alors :

```
        évaluer expi
        var := valeur expi
   Etl: évaluer instr
        incrémenter var et branch. à Etl si var=0
```

En général, la variable de la boucle sera placée dans un registre car ces instructions ne peuvent opérer que sur des registres.

3.2.3. Autres instructions.

L'instruction "goto" est traduite par un simple branchement, à moins que l'étiquette cible n'appartienne à un autre bloc ; dans ce dernier cas, la pile d'exécution doit être mise à jour avant d'effectuer le branchement pour permettre à la procédure contenant l'étiquette d'accéder à ses données. Etant donné que l'étiquette appartient nécessairement à une procédure englobante, l'environ de celle-ci est déjà présent dans la pile ; la mise à jour consiste par conséquent à dépiler tous les environs situés au-dessus de lui et à restaurer les registres "display" qui lui sont nécessaires.

La dernière instruction que nous traitons ici est le "with", bien qu'il ne s'agisse pas tout à fait d'une instruction de contrôle. Pour faciliter la traduction de cette instruction, nous nous servons de notre mécanisme d'évaluation de sous-expressions communes ; en effet, les expressions du "with" correspondent parfaitement à cette notion car elles ne doivent

être évaluées qu'avant le "with" et leurs valeurs réutilisées partout où les expressions apparaissent dans le corps du "with". La traduction de cette instruction se réduit donc à enchainer la traduction de ses expressions et de son corps.

3.3. Appel de procédure.

L'appel de procédure est l'une des 2 instructions Pascal les plus utilisées, d'après les statistiques réalisées sur ce langage (‹Fortier75›, ‹Shimasaki80›) ; d'autre part, un appel implique l'activation d'un nouvel environ de procédure, ce qui en fait une opération particulièrement complexe. Il est par conséquent très intéressant d'optimiser la traduction de cette instruction.

Un appel de procédure n'est que partiellement traité à l'endroit où il est effectué ; en effet, une partie de l'appel est en général réalisé à l'entrée de la procédure, ce qui permet de partager cet "en-tête" entre tous les appels et de réduire globalement le code généré. Une première optimisation consiste donc à déterminer où placer les différentes opérations composant un appel ; il faut remarquer que les informations dont on dispose avant l'appel ou à l'entrée de la procédure ne sont pas les mêmes ; réaliser une opération dans l'en-tête de la procédure peut par conséquent être plus coûteux, voire même impossible, par manque d'informations.

La figure VII.18 illustre les choix que nous avons faits à ce propos. On peut remarquer que le résultat d'une fonction est toujours rangé dans un registre, pour des raisons d'efficacité. On peut remarquer également que la sauvegarde de registres est réalisée avant l'appel effectif ; nous pensons en effet que c'est le plus efficace, pour 2 raisons principales :

a) Le nombre de registres à sauvegarder dans ce cas est très souvent égal à zéro car il s'agit de registres occupés lors de l'appel. Si la sauvegarde était faite à l'entrée de la procédure, il faudrait par contre sauvegarder tous les registres modifiés par la procédure.
b) La compilation est plus simple car on sait quels registres

```
sauvegarde reg. occupés
évaluation paramètres
empilement paramètres

appel effectif              ---->   initialiser lien dynamique
                                    charger registres display
                                    mise à jour sommet pile
                                    initialiser nouveau display

                                    < corps de la procédure >

                                    ( résultat => reg. "fonction" )
                                    (  s'il s'agit d'une fonction )
                                    restaurer ancien sommet pile
                                    restaurer reg. base appelant
restaurer reg. display      <----   retour effectif
désempiler paramètres
```

Figure VII.18 Schéma de décomposition d'un appel de procédure

sont occupés au moment du traitement de la sauvegarde. Dans l'autre cas, il faudrait attendre la fin de la compilation de la procédure pour connaître les registres qu'elle modifie, pour ensuite revenir en arrière afin de générer les instructions de sauvegarde.

Toutefois, il existe une exception ; lorsqu'il s'agit d'une procédure prédéfinie, nous effectuons cette sauvegarde à l'entrée de la procédure.

Le deuxième type d'optimisation consiste à tirer profit des mécanismes de gestion de pile de la machine, s'ils existent ; dans ce cas, nous utilisons les instructions d'empilement et de désempilement pour effectuer la sauvegarde de registres et de paramètres ; par ailleurs, nous n'avons plus besoin de sauvegarder l'adresse de retour de la procédure car ceci est fait automatiquement par l'instruction d'appel de procédure. En contrepartie, il faut générer des instructions complémentaires pour maintenir à jour le registre sommet de cette pile.

Ces 2 types d'optimisations s'appliquent également au retour de la procédure dont le schéma de traduction est présenté dans la figure VII.18.

VIII. L'OPTIMISEUR FINAL.

Il est chargé d'optimiser les instructions générées par le codeur et de réaliser la sélection des instructions de branchement. Il est composé de 2 phases :

a) La première phase applique les règles d'optimisation décrites dans le paragraphe IV.2.

b) La seconde a pour rôle de déterminer l'instruction machine la mieux adaptée à chaque branchement du programme ; ceci est nécessaire car, dans la plupart des machines, on trouve plusieurs sortes d'instructions de branchement, avec des possibilités d'adressage et des coûts différents.

Dans ce qui suit, nous présentons les 2 phases qui composent l'optimiseur. Nous présentons également la structure de données qui sert d'interface entre un générateur GEMME et l'optimiseur ainsi que les structures de données utilisées en interne par ce dernier.

1. Interface avec le générateur.

Le programme objet produit par la phase de génération joue le rôle d'interface entre le générateur et l'optimiseur final ; ce programme contient des nombreuses informations qui sont communiquées explicitement à l'optimiseur.

En effet, lorsque le générateur fait l'élaboration du code objet, il a besoin de consulter la description de la machine cible et de calculer d'autres informations relatives aux instructions qu'il doit générer ; il dispose par conséquent d'un nombre important d'informations sur chaque instruction qui compose le programme objet. Certaines de ces informations sont également nécessaires à l'optimiseur final car il travaille au niveau des instructions machine ; pour ne pas les reconsulter ou

les recalculer, le générateur les lui transmet avec le code généré. Ainsi chaque instruction générée contient un certain nombre d'informations destinées à l'optimiseur :

a) Le type d'instruction : branchement, affectation, opération arithmétique-logique ou opération sans effet direct. Ce dernier type correspond aux instructions qui ne modifient pas leurs opérandes et est constitué principalement par les instructions de test et de comparaison.
b) La taille de l'instruction ; la taille d'un branchement est indéterminée car l'instruction correspondante ne sera connue qu'après la deuxième phase de l'optimiseur.
c) Le code opération de l'instruction ; ce code est indéterminé dans le cas d'une instruction de branchement et sera rajouté par la deuxième phase de l'optimiseur.
d) Le premier opérande ; pour une instruction de branchement, il contient le type de condition à tester (égal, plus grand, ..., sans condition).
e) Le deuxième opérande ; pour une instruction de branchement, il contient l'étiquette cible.

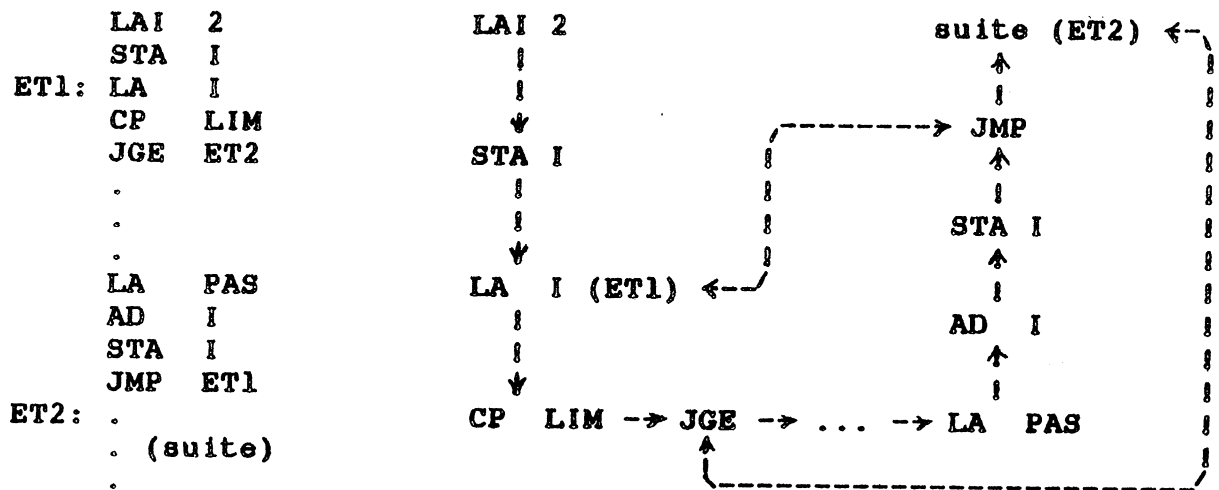
D'autre part, le générateur produit une table d'étiquettes, destinée en exclusivité à l'optimiseur. Ces étiquettes sont donc éliminées en sortie de l'optimiseur et leurs utilisations remplacées par les adresses correspondantes.

2. Structures de données internes.

Le programme objet est représenté sous la forme d'une liste circulaire doublement chaînée, où chaque élément représente une instruction du programme. Le choix d'une liste doublement chaînée nous a été imposé par le type de traitement effectué qui nécessite dans certains cas un parcours en arrière ; par ailleurs, les opérations de rajout et de suppression d'un élément sont plus simples avec une telle structure. Le choix d'une liste circulaire, moins impératif, a été réalisé pour des raisons pragmatiques :

1) le traitement effectué correspond tout à fait à un balayage circulaire et itératif du programme,

2) il nous permet de banaliser les opérations en début et en fin de liste.



a) Code objet. b) Liste d'instructions générées.

Figure VIII.1. Représentation du programme objet.

La figure VIII.1 présente un exemple de cette structure ; le programme source correspondant est celui de la figure IV.1. On peut remarquer que les étiquettes ne sont pas des éléments de la liste, comme c'est le cas dans le compilateur Bliss/11 (‹Wulf75›) ; par contre, chaque instruction possède un champ indiquant les reférences qui lui sont faites.

La deuxième structure de données utilisée par l'optimiseur est une liste de blocs d'instructions, que nous appelons la liste d'évaluation des branchements. Chaque bloc de cette liste représente :

a) soit une suite d'instructions non reférencées par des branchements (à l'exception de la première) et qui ne contient aucun branchement,
b) soit une instruction de branchement.

Cette liste est elle-aussi circulaire et doublement chainée ; elle est utilisée à la place de la liste d'instructions par la deuxième phase de l'optimiseur, pour 2 raisons principales :

a) le parcours est plus rapide car cette phase ne s'intéresse qu'aux instructions de branchement,
b) le calcul de la distance entre un branchement et sa cible est simplifié ; en effet, la taille des blocs sans branchements

n'est calculée qu'une seule fois.
La liste d'évaluation correspondant à l'exemple précédent est présentée dans la figure VIII.2.

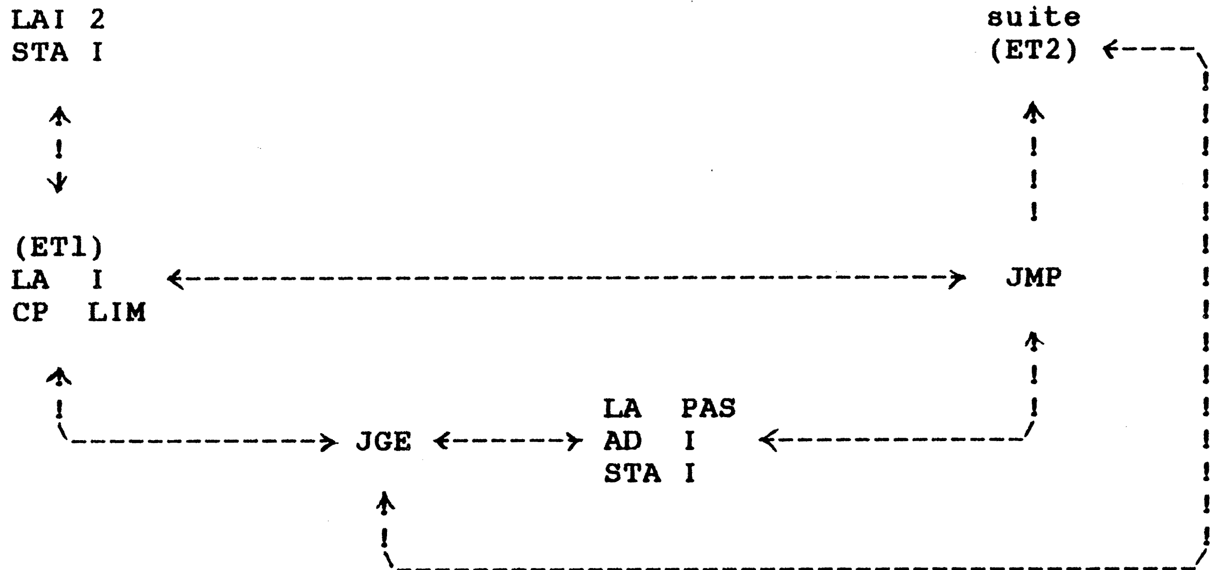


Figure VIII.2. Liste d'évaluation des branchements.

Enfin, l'optimiseur utilise une troisième structure de données ; il s'agit de la description des instructions de branchement de la machine cible, présentée au chapitre V.

3. Phase d'optimisation.

Au cours de cette phase, la liste d'instructions générées est parcourue de façon circulaire jusqu'à ce qu'aucune optimisation ne puisse être appliquée pendant un tour complet, à compter de la dernière instruction optimisée. Ces passes répétées sont effectuées pour tirer un maximum de profit car une optimisation peut toujours faire apparaître de nouvelles opportunités d'optimisation.

Toutes les instructions sont examinées durant le parcours ; le type d'opération de l'instruction détermine l'ensemble des règles d'optimisation qu'il faut essayer de lui appliquer :

a) Si l'instruction est un branchement inconditionnel, elle peut être suivie de code inaccessible ou bien avoir pour cible l'instruction suivante ; elle peut également être le début

d'une chaine de branchements.

b) Si c'est un branchement conditionnel, il se peut aussi qu'il pointe sur l'instruction suivante ou qu'il donne lieu à une chaine de branchements. Il faut également vérifier s'il est suivi d'un branchement inconditionnel à la même cible ou si l'on peut inverser sa condition.

c) S'il s'agit d'une affectation, nous regardons si elle est redondante. Nous en conservons une trace pour pouvoir analyser les affectations suivantes.

d) Si c'est une opération arithmétique-logique, nous annulons la trace de la dernière affectation éventuelle, faite à la ressource résultat de l'opération.

A la fin du parcours, la liste d'instructions représente le programme optimisé.

4. Phase de résolution des branchements.

Généralement les machines disposent d'au moins 2 types d'instructions de branchement :

a) Un format court (branchements relatifs) dont l'adressage de la cible est relatif au compteur ordinal. Il est plus rapide et occupe moins d'espace mais il est limité à un petit intervalle d'adressage (en général -128..+127).

b) Un format long (branchements non relatifs) qui permet d'adresser toutes les positions mémoire, moyennant un coût supérieur.

Certains microprocesseurs récents proposent même plusieurs catégories de branchements relatifs, avec des intervalles d'adressage plus ou moins grands ; c'est le cas, par exemple, du 68000 (‹Motorola80›) qui permet 2 intervalles : -128..+127 et -32768..+32767.

A la génération du code objet, il est difficile, et parfois impossible, de choisir la meilleure instruction pour chaque branchement du programme. En effet, la détermination de cette instruction dépend de la taille de la partie du programme qui

sépare le branchement de sa cible ; or, cette taille n'est pas toujours connue au moment de la génération, à cause des branchements en avant et de l'entrelacement qui peut exister entre plusieurs branchements. De ce fait, beaucoup de générateurs choisissent d'utiliser systématiquement le format long, sauf pour les cas évidents.

En ce qui nous concerne, nous estimons que les branchements méritent une attention particulière car ils représentent un pourcentage élevé des instructions d'un programme ; nous essayons donc de leur associer le format le plus court, permettant d'adresser leurs cibles. Ce traitement ne peut être réalisé que dans la deuxième phase de l'optimiseur, où l'on dispose d'un programme objet complet (à l'exception des branchements) et qui a déjà été optimisé.

Après une séquence d'initialisation, dont le but est de construire la liste d'évaluation des branchements et de calculer la taille des blocs d'instructions, la deuxième phase se déroule en 3 étapes que nous présentons ci-après.

4.1. Résolution des cas simples.

La distance exacte entre un branchement et sa cible ne peut pas être calculée tant qu'il y a des branchements non résolus dans la séquence d'instructions qui les sépare. Cependant, le calcul de cette distance n'est pas toujours indispensable ; dans certains cas, l'instruction choisie sera la même quel que soit le choix fait pour les branchements intermédiaires.

```
...
if VALEUR > 0
then VALEUR := COEFF-VALEUR
else VALEUR := COEFF+VALEUR ;
...
```

a) Programme PASCAL

```
      CMP  VALEUR,#0
      BGE  ET1
      MOVE R0,COEFF
      SUB  R0,VALEUR
      MOVE VALEUR,R0
      BR   ET2
ET1:  ADD  VALEUR,COEFF
ET2:  ...
```

b) Traduction PDP-11

Figure VIII.3. Un cas simple de résolution de branchement.

Prenons comme exemple le programme de la figure VIII.3 et sa traduction pour le PDP-11 (<Dec75>), où l'on dispose de branchements courts (BR, BEQ, BGE, etc) et de branchements longs (JMP). Dans ce programme, nous pouvons utiliser l'instruction "BGE" pour mettre en oeuvre le branchement conditionnel vers l'étiquette ET1, même si l'instruction "BR ET2" est remplacée par "JMP ET2" ; en effet, dans les 2 cas, la distance appartient à l'intervalle d'adressage des branchements courts (-128..+127). Le cas contraire, où l'on doit utiliser l'instruction "JMP", peut également se présenter.

Cette première étape est chargée de traiter ces cas simples. Pour ce faire, elle parcourt la liste d'évaluation des branchements, essayant de détecter ces cas pour leur assigner une instruction de façon définitive ; si un branchement au moins est résolu, un nouveau parcours est effectué car d'autres branchements ont pu devenir des cas "simples".

Durant un parcours, nous calculons les distances les plus favorable et les plus défavorable pour chaque branchement non résolu ; ce calcul est réalisé en supposant que tous les branchements intermédiaires seront respectivement du format le plus court et le plus long. Si pour un branchement donné, ces 2 distances appartiennent à l'intervalle d'une instruction de format court, nous prenons immédiatement cette instruction ; si toutes deux sont en dehors de l'intervalle, nous essayons un intervalle plus grand. Si tous les branchements relatifs ont été essayés sans succès, nous utilisons l'instruction de format long.

Si l'une de ces 2 distances est située à l'intérieur d'un intervalle et l'autre à l'extérieur, le choix n'est pas encore possible ; l'instruction en question ne peut être utilisée que si un certain nombre de branchements intermédiaires utilisent le format court. Nous laissons donc le branchement à l'état non résolu ; il faut remarquer que la résolution d'un autre branchement, entrelacé avec lui, peut faire disparaître le conflit.

4.2. Résolution des conflits.

A la fin de l'étape précédente, un certain nombre de branchements peuvent rester sans solution ; il s'agit des cas où l'entrelacement est tel qu'il les fait dépendre les uns des autres. Cette étape a pour objectif la résolution de ces conflits.

La méthode employée est un peu brutale : quand nous rencontrons un conflit, nous utilisons systématiquement l'instruction de coût immédiatement supérieur à celle qui provoque le conflit. Notre expérience nous a montré que, grâce à l'utilisation des structures de contrôle évoluées et de la programmation dite structurée, ces conflits sont peu fréquents ; à notre avis, un traitement plus sophistiqué est donc superflu.

Le traitement se fait en un parcours unique de la liste d'évaluation des branchements. Remarquons que l'on peut rencontrer des cas "simples" durant ce parcours car la résolution d'un conflit peut en résoudre d'autres.

4.3. Calcul des adresses de branchement.

Cette dernière étape est chargée de préparer la version finale et complète du programme objet. Pour ce faire, deux parcours de la liste d'instructions sont effectués :

1) le premier calcule le déplacement relatif de chaque instruction par rapport au début du programme,
2) le deuxième reconstruit la représentation linéaire du programme et l'écrit sur le fichier de sortie.

Durant le deuxième parcours, les instructions de branchement sont traitées de façon particulière. En effet, nous disposons depuis la génération de code de toutes les informations concernant les autres instructions, tandis que pour les branchements nous n'avions aucune information ; il nous faut par conséquent construire les divers champs de l'instruction, à partir des différentes informations dont nous disposons.

Ainsi, le choix fait par les 2 étapes précédentes nous permet de retrouver, dans la description de la machine, le code opération de l'instruction ainsi que le mode d'adressage de son opérande. D'autre part, les déplacements relatifs des instructions nous permettent de calculer l'adresse de l'instruction cible du branchement ; cette adresse est transformée ensuite en fonction du mode d'adressage de l'instruction.

5. Un exemple d'optimisation.

Nous présentons ci-après un exemple du traitement réalisé par l'optimiseur. Cet exemple nous permet d'illustrer le fonctionnement de l'optimiseur ainsi que les interactions existantes entre les différentes optimisations.

```
program exemple( input, output ) ;
var car : char ;
    valcar, valeur : integer ;
begin
   (* ignorer les blancs du début *)
   read( car ) ;
   while car = ' ' do read( car ) ;

   (* boucle de traitement *)
   while car <> '.' do
   begin
      (* traitement d'un entier *)
      valeur := ord( car ) - ord( '0' ) ;
      read( car ) ;
      while ( car <> ' ' ) and ( car <> '.' ) do
      begin
         valeur := valeur * 10 ;
         valcar := ord( car ) - ord( '0' ) ;
         valeur := valeur + valcar ;
         read( car )
      end ;
      write( valeur ) ;

      (* ignorer les blancs entre 2 entiers *)
      while car = ' ' do read( car )
   end
end.
```

Figure VIII.4. Programme source.

Le programme Pascal utilisé est présenté dans la figure VIII.4. Ce programme lit une suite de caractères contenant

plusieurs entiers positifs, transforme ces derniers en binaire et imprime leurs valeurs ; la suite de caractères se termine par un ".". Il s'agit d'un programme très simple et court mais qui donne lieu à de nombreuses optimisations.

La figure VIII.5 donne le code produit par un générateur GEMME pour le programme "exemple", avant que les optimisations finales soient appliquées ; la cible utilisée pour l'exemple est le microprocesseur 68000. Nous avons utilisé un langage très proche de l'assembleur 68000 pour des raisons de clarté; des commentaires ont été rajoutés pour illustrer la traduction réalisée. Voici la sémantique de quelques instructions :

```
pea  = empiler l'adresse de l'opérande
jsr  = appel de procédure
addq = addition immédiate (1..8)
cmpi = comparaison immédiate (-128..127)
bne  = branchement conditionnel court (-128..127)
bra  = branchement inconditionnel court (-128..127)
beql = branchement conditionnel long (-32768..32767)
ext  = extension du signe
```

La plupart d'instructions ont un suffixe indiquant la longueur des opérandes (b=octet, w=mot, l=double mot).

Il faut remarquer que le code généré est déjà de bonne qualité. Toutefois, de nombreuses optimisations sont encore possibles et seront effectuées par notre optimiseur :

a) L'étiquette et0 est le point de confluence de 2 chemins d'exécution dont les dernières instructions sont identiques ("cross jumping") ; il s'agit respectivement des séquences qui précèdent l'étiquette et0 et l'instruction "bra et0". Une de ces séquences peut donc être supprimée ; après cette optimisation, le début du programme devient

```
et8: pea    _input
     jsr    _readc
     moveb  d7,car(a0)
     addql  #4,a7
     cmpib  car(a0),#' '
     bne    et1
```

```
< a0 est utilisé comme base des données du programme
< a7 est utilisé comme sommet de pile
< d7 est utilisé pour rendre le résultat des fonctions
      pea    _input             < begin
      jsr    _readc             <
      moveb  d7,car(a0)         <
      addql  #4,a7              <     read( car ) ;
et0:  cmpib  car(a0),#' '       <     while
      bne    et1                <        car = ' '
      pea    _input             <     do
      jsr    _readc             <
      moveb  d7,car(a0)         <
      addql  #4,a7              <        read( car ) ;
      bra    et0                <
et1:  equ    $                  <
et2:  cmpib  car(a0),#'.'       <     while
      beql   et3                <        car <> '.' do
      moveb  car(a0),d0         <     begin
      subib  #'0',d0            <        d0:=car-'0'
      extw   d0                 <
      extl   d0                 <
      movel  d0,valeur(a0)      <          valeur := d0 ;
      pea    _input             <
      jsr    _readc             <
      moveb  d7,car(a0)         <
      addql  #4,a7              <        read( car ) ;
et4:  cmpib  car(a0),#' '       <        while
      beq    et5                <           car <> ' '
      cmpib  car(a0),#'.'       <              and
      beq    et5                <           car <> '.' do
      pea    10                 <        begin
      movel  valeur(a0),(a7)-   <
      jsr    _multi             <
      addql  #8,a7              <
      movel  d7,valeur(a0)      <           valeur := valeur*10 ;
      moveb  car(a0),d0         <
      subib  #'0',d0            <           d0 := car-'0'
      extw   d0                 <
      extl   d0                 <
      movel  d0,valcar(a0)      <           valcar := d0 ;
      movel  valcar(a0),d0      <
      addl   d0,valeur(a0)      <           valeur := valeur+valcar;
      pea    _input             <
      jsr    _readc             <
      moveb  d7,car(a0)         <
      addql  #4,a7              <           read( car )
      bra    et4                <        end ;
et5:  pea    _output            <
      movel  valeur(a0),(a7)-   <
      pea    _formati           <
      jsr    _writei            <
      addl   #12,a7             <        write( valeur ) ;
et6:  cmpib  car(a0),#' '       <        while
      bne    et7                <           car = ' '
      pea    _input             <        do
      jsr    _readc             <
      moveb  d7,car(a0)         <
      addql  #4,a7              <           read( car )
      bra    et6                <
et7:  bral   et2                <     end
et3:  <fin   prog.>             < end.
```

Figure VIII.5. Code généré avant optimisation.

```
        bra   et8
et1:    equ   $
```

b) L'optimisation précédente crée une nouvelle opportunité d'optimisation ; en effet, la séquence

```
        bne   et1
        bra   et8
et1:    equ   $
```

peut être remplacée tout simplement par

```
        beq   et8
```

c) L'étiquette et4 est elle-aussi le point de rencontre de 2 séquences identiques ; on peut donc appliquer la même optimisation qu'en a) et obtenir

```
et9:    pea   _input
        jsr   _readc
        moveb d7,car(a0)
        addql #4,a7
        cmpib car(a0),#' '
        beq   et5
        ...
        addl  d0,valeur(a0)
        bra   et9
et5:    ...
```

d) La deuxième instruction de la séquence

```
        movel d0,valcar(a0)
        movel valcar(a0),d0
        addl  d0,valeur(a0)
```

est une affectation redondante ; elle sera par conséquent supprimée.

e) Le branchement "bne et7" a pour cible un branchement inconditionnel ("bra et2") ; nous pouvons donc remplacer son étiquette par celle de sa cible finale. L'instruction devient alors

```
        bne   et2
```

f) Après l'optimisation précédente, l'étiquette et7 n'est plus

```
et8: pea   _input
     jsr   _readc
     moveb d7,car(a0)
     addql #4,a7
     cmpib car(a0),#' '
     beq   et8
et2: cmpib car(a0),#'.'
     beq   et3
     moveb car(a0),d0
     subib #'0',d0
     extw  d0
     extl  d0
     movel d0,valeur(a0)
et9: pea   _input
     jsr   _readc
     moveb d7,car(a0)
     addql #4,a7
     cmpib car(a0),#' '
     beq   et5
     cmpib car(a0),#'.'
     beq   et5
     pea   10
     movel valeur(a0),(a7)-
     jsr   _multi
     movel d7,valeur(a0)
     addql #8,a7
     moveb car(a0),d0
     subib #'0',d0
     extw  d0
     extl  d0
     movel d0,valcar(a0)
     addl  d0,valeur(a0)
     bra   et9
et5: pea   _output
     movel valeur(a0),(a7)-
     pea   _formati
     jsr   _writei
     addl  #12,a7
et6: cmpib car(a0),#' '
     bne   et2
     pea   _input
     jsr   _readc
     moveb d7,car(a0)
     addql #4,a7
     bra   et6
et3: <fin  prog.>
```

Figure VIII.6 Code généré après optimisation.

référencée et sera par conséquent supprimée ; l'instruction "bra et2" devient alors inaccessible et sera elle-aussi supprimée.

g) Ensuite, l'optimiseur fait la sélection des instructions de branchement. Dans l'exemple, il a partout choisi des

branchements courts car les distances entre branchement et cible le permettaient ; ce choix n'est réalisé qu'après les optimisations mais nous avons préféré l'expliciter dès le début pour des raisons de clarté. On peut remarquer qu'un branchement long ("beql et3") a été transformé en court, suite aux optimisations réalisées.

Le résultat final est donné dans la figure VIII.6. La taille du programme est de 200 octets avant optimisation et de 160 octets après ; une réduction d'au moins 20 pour cent a par conséquent été obtenue.

RESULTATS

ET

CONCLUSION

IX. RESULTATS ET CONCLUSION.

1. Réalisation d'un prototype du système GEMME.

Nous avons réalisé un prototype du système GEMME sur l'ordinateur CII-HB 68 DPS-3 sous le système Multics. Nous analysons ci-après les aspects principaux de cette réalisation ; nous présentons d'abord les raisons de notre choix du langage de développement, puis l'état actuel du prototype et enfin les résultats des expériences effectuées.

1.1. Choix du langage de développement.

Au départ, nous avions envisagé de réaliser le prototype en Prolog sous Multics. Nous avons donc réalisé une petite maquette, comportant les mécanismes de base de notre méthode. Cette expérience a mis en évidence certains points favorables à l'utilisation d'un tel langage ; nous citerons, en particulier, les facilités pour la représentation de nos structures de données ainsi que pour la mise en oeuvre de la reconnaissance de formes et du retour arrière. Malheureusement, elle a également révélé un certain nombre de lacunes suffisantes pour nous dissuader de poursuivre dans cette voie ; les lacunes principales, qui tiennent pour une bonne part à la version de Prolog disponible, sont :

- le manque d'un environnement de mise au point,
- l'absence de variables globales,
- le coût prohibitif en temps machine.

Suite à ces inconvénients, nous nous sommes tournés vers d'autres langages : Rlisp que nous avons préféré à Maclisp car sa syntaxe, proche de celle de Pascal, offre une meilleure lisibilité et une plus grande souplesse d'écriture, et Pascal. Pour sélectionner le langage qui nous conviendrait le mieux, nous avons donc programmé en Pascal et en Rlisp un embryon de

notre prototype. Cette expérience nous a permis de constater la supériorité de Rlisp pour l'expression de notre problème ; les principaux atouts sont :

a) Le traitement et l'initialisation de données organisées en listes et en arbres est beaucoup plus facile ; il faut indiquer que Rlisp fonctionne sous l'environnement Maclisp, ce qui lui permet de bénéficier de la puissance d'expression de ce dernier.
b) Toutes les fonctions écrites sont appelables au niveau commande de l'environnement, ce qui facilite l'écriture de programmes de test.
c) Maclisp offre un environnement de développement comprenant un interpréteur et un certain nombre de facilités de mise au point.
d) Un programme Rlisp compilé est aussi performant qu'un programme Pascal équivalent.

Nous avons donc choisi Rlisp pour développer notre prototype. Toutefois, nous ne nous sommes servis de ce langage qu'en tant que pré-processeur de Maclisp.

1.2. Etat actuel.

Le prototype GEMME que nous avons réalisé est actuellement incomplet mais suffisant pour nous permettre de valider nos idées. Dans son état actuel, le prototype comprend :

a) Un noyau de génération (cf VII.1) chargé de traduire les opérations simples et de gérer les ressources de la machine cible (registres, temporaires et constantes).
b) Une couche de traduction des expressions et des instructions Pascal (cf VII.2 et VII.3) ; la traduction se fait en décomposant celles-ci en opérations élémentaires qui sont elles-mêmes soumises au noyau de génération. Cette couche est chargée en outre du traitement des sous-expressions communes et de la gestion des étiquettes.
c) Un allocateur des variables (cf VI) dont la tâche est de trouver une représentation pour chaque type défini par l'utilisateur et d'associer une adresse mémoire et un descripteur à chaque variable du programme.

d) Un optimiseur final (cf VIII) chargé d'optimiser localement le code généré et de faire le choix des instructions machine pour chaque branchement.

L'ensemble de ces quatre composants constitue le générateur de code indépendant des machines. Par ailleurs, nous avons réalisé un analyseur syntaxique et sémantique qui nous permet d'alimenter le générateur à partir de programmes source ; cet analyseur nous permet de compiler tout programme Pascal mais est très limité en ce qui concerne la récupération après une erreur (nous ne concevons l'utilisation de GEMME que pour la traduction de programmes déjà au point, dans le contexte d'un atelier de logiciel).

Le prototype comprend également un compilateur qui construit les tables de génération d'une machine à partir de sa description en LD, notre langage de description de machines.

Enfin, nous avons réalisé à ce jour deux descriptions de machine (Motorola 68000 et PDP-11) que nous utilisons pour nos expériences. Ces deux descriptions reflètent l'état actuel du protoype ; en effet, elles ne décrivent que tout ce qui peut être traité actuellement par celui-ci.

Nous donnons ci-après la taille approximative des différents composants du prototype (en lignes Rlisp) ainsi que des descriptions réalisées (en lignes LD) :

noyau de génération	5200
traducteur expressions-instructions	1400
allocateur des variables	800
optimiseur final	2000
analyseur Pascal	3400
compilateur LD	2400
description MC 68000	900
description PDP-11	650

Soit environ 15200 lignes Rlisp et 1550 lignes LD. Il faut indiquer que les lignes de commentaires sont comprises dans ces tailles ; nous avons observé un rapport d'environ 2 lignes de commentaire pour 3 lignes de code effectif.

D'autres composants du système GEMME sont actuellement en cours de développement ; ces composants sont les suivants :

a) Un optimiseur global chargé de réaliser des transformations relativement simples et indépendantes de la machine sur l'arbre abstrait du programme : identification de sous-expressions communes, évaluation des expressions statiques, propagation de constantes, expansion de certaines opérations (accès aux tableaux, conversions implicites, etc.) et déplacement des calculs constants des boucles.

b) Un assembleur qui traduit le programme symbolique généré par le codeur en langage binaire ; le résultat de cet assembleur est compatible avec le format du module objet correspondant au système de la machine. La traduction est dirigée par les formats de génération figurant dans la description de la machine cible.

1.3. Expérimentation.

Nous avons réalisé un certain nombre d'expériences avec le prototype de notre système. Ces expériences ont porté sur une variété de programmes de petite taille (environ 30 lignes), choisis de façon à tester les différentes constructions du langage Pascal ; les machines cibles utilisées ont été le Motorola 68000 et le PDP-11.

Les expériences réalisées nous ont permis de vérifier que nos objectifs initiaux ont été largement atteints. Nous rappelons que ces objectifs étaient les suivants, dans l'ordre d'importance décroissante :

- qualité du code généré,
- rapidité de production d'un nouveau générateur,
- efficacité (en temps d'exécution) des générateurs construits avec le système.

Nous analysons ci-après les résultats obtenus pour chacun d'eux.

1.3.1. Qualité du code généré.

Les expériences que nous avons pu réaliser jusqu'à présent tendent à montrer que le code produit est de très bonne qualité. Nous avons en particulier effectué une comparaison entre le code généré par notre prototype et le code généré par trois autres compilateurs :

- Pascal Siemens, disponible sur le SM90 (Motorola 68000),
- Pascal Berkeley, disponible sur le Micromega (Motorola 68000),
- OMSI Pascal-1, disponible sur le PDP-11.

Dans tous les cas, la comparaison tourne à l'avantage de notre générateur. Nous illustrons cette comparaison à l'aide du programme "conversion", présenté dans l'annexe F ; nous y donnons le programme source ainsi que les programmes objets générés par chaque compilateur. Pour ce programme, notre générateur permet d'économiser entre 27 et 45% sur la taille du programme objet ; on peut s'attendre à un gain équivalent en temps d'exécution.

1.3.2. Production d'un nouveau générateur.

La construction d'un nouveau générateur comprend les tâches suivantes :

- l'écriture de la description,
- la validation du nouveau générateur, de façon à vérifier que la description est correcte et complète,
- l'écriture des fonctions externes prédéfinies du langage ("run-time").

L'effort de réalisation nécessaire à cette dernière tâche est le même que pour un compilateur classique. En ce qui concerne les deux autres tâches, nous estimons leur réalisation à environ un mois dont une semaine pour l'écriture de la description.

A titre indicatif, la description du PDP-11 nous a demandé environ trois jours ; il faut toutefois indiquer que cette description n'était pas complète au moment de la réalisation.

1.3.3. Vitesse de compilation.

Dans l'état actuel, le prototype génère une moyenne de dix instructions par seconde (pour le Motorola 68000). Toutefois, il faut savoir que ce prototype travaille en mode "mise au point" Maclisp et qu'il effectue de nombreuses vérifications ; on peut par conséquent espérer une amélioration sensible de cette moyenne dans une version de production.

Nous considérons que cette vitesse de compilation est acceptable, surtout dans un environnement comme Adèle où un programme n'est censé être compilé qu'une fois mis au point. Il nous est difficile de présenter des comparaisons significatives avec les compilateurs disponibles sur nos cibles car notre prototype fonctionne actuellement sur une autre machine.

Pour conclure sur cet aspect, nous devons souligner que les résultats obtenus démontrent que le choix d'une recherche exhaustive pour la sélection d'instructions ne s'avère pas déraisonnable.

2. Contributions.

Nous avons présenté dans cette thèse une méthode de génération de code indépendante des machines cibles ainsi qu'un système de production automatique de générateurs basé sur cette méthode. Nous avons également présenté le prototype qui nous a servi à démontrer la faisabilité des propositions avancées. Nous essayons ci-après de dégager les apports de notre travail au problème de la génération de code :

a) L'utilisation d'une recherche exhaustive pour la sélection d'instructions permet de produire du code de bonne qualité. Cette approche garantit en effet l'obtention du meilleur code pour chaque opération du programme à compiler, tout en restant dans des performances acceptables.
b) L'architecture adoptée pour notre générateur permet une distribution fonctionnelle des tâches entre ses différents composants. Ainsi, par exemple, c'est l'optimiseur final et

non pas le codeur qui réalise la sélection des instructions de branchement car il dispose pour ce faire d'informations plus complètes.

c) Les optimisations du code généré ont été modélisées de façon indépendante de la machine cible ; il faut noter que ces optimisations sont en général traitées de façon spécifique pour chaque machine.

d) L'utilisation des descripteurs apporte une grande souplesse au niveau du traitement des opérandes ; en particulier, elle permet la représentation de modes d'adressage complexes.

e) Tous les algorithmes sont totalement indépendants des machines. Tout ce qui dépend de la machine est donné sous une forme purement descriptive, dans un langage approprié ; aucune tâche de la génération n'est donc à la charge de l'écrivain du compilateur.

Nous tenons également à signaler l'utilisation des techniques de l'intelligence artificielle pour la réalisation de notre système. Ces techniques se sont révélées bien adaptées aux problèmes traités, notamment pour la sélection d'instructions.

3. Perspectives.

A partir des expériences réalisées, nous avons pu dégager diverses améliorations et extensions qu'il serait souhaitable d'apporter au système GEMME.

Plusieurs d'entre elles seront introduites dans la version du système qui sera réalisée prochainement pour le projet CONCERTO :

a) Une meilleure utilisation des registres de la machine cible. Nous voulons notamment garder une trace du contenu de chaque registre dans le but de le réutiliser autant que possible.

b) Définition d'un nouveau langage de description. L'évolution de notre système nécessite un certain nombre d'extensions au langage LD (instructions spécialisées, formats de génération, etc.) ; nous voulons également profiter de cette redéfinition pour rajouter la possibilité d'associer une séquence

d'instructions à chaque opération.

c) Traitement "à l'avance" des opérations les plus communes. Notre objectif est de déterminer les opérations les plus fréquentes des programmes Pascal et de simuler leur compilation lors de la construction d'un générateur ; le résultat sera ensuite utilisé directement chaque fois qu'une de ces opérations est rencontrée lors de la compilation d'un programme quelconque.

d) Réalisation d'une évaluation plus complète. La nouvelle version de notre système doit être implantée sur un SM90, ce qui nous permettra de faire une meilleure évaluation ; nous pourrons en particulier comparer la vitesse de notre générateur avec celle des compilateurs disponibles sur cette machine.

D'autre part, il nous paraît intéressant, à plus long terme, de généraliser notre démarche afin d'enrichir les capacités de notre système. Une première possibilité serait l'extension aux machines à trois adresses. L'adaptation à cette classe de machines n'entraîne que des changements mineurs au niveau description ; par contre, elle nécessite une remise en cause de la partie évaluation du résultat d'une opération, au niveau du codeur.

Une autre possibilité serait la prise en compte des effets secondaires dus aux modes d'adressage et aux instructions. Nous pensons en particulier aux modes d'adressage autoincrémentés ainsi qu'au positionnement du code de condition par certaines instructions. Nous considérons que cette extension risque d'augmenter de façon notable la compléxité du système.

Enfin, la perspective la plus intéressante, à notre avis, serait la possibilité de paramétrer le générateur par une description du langage source. Cette possibilité ne devrait entraîner que des changements superficiels au niveau du noyau de génération et des parties en aval du codeur ; par contre, elle remettrait en cause tous les autres composants du générateur.

ANNEXES

ANNEXE A : DEFINITION DU LANGAGE DE DESCRIPTION (LD).

La grammaire du langage LD est donnée ci-après à l'aide du formalisme BNF. Toute ligne commençant par le caractère "-" est un commentaire décrivant la sémantique des règles qui précèdent. L'axiome de la grammaire est <description>.

```
<description> --> DESCRIPTION : <id D>  <équivalences>  <types>
                  <registres> <modèles> <instructions>
                  <affectations> <branchements> FINDESC
```

- <id D> est le nom de la machine décrite.

```
<équivalences> --> EQUIVALENCES ;  <équivalence>*
<équivalence>  --> <id E> : <valeur> ;
```

- Une équivalence définit une constante qui peut ensuite être
- utilisée tout au long de la description.
- <id E> est le nom de la constante et <constante> sa valeur.

```
<types>          --> TYPES ; <types machine> <types Pascal>
<types machine> --> MACHINE ; <type machine>*
<type machine>  --> <id M> : <constantel> , <constante 2> ;
```

- La partie types machine décrit les différents types de base
- de la machine.
- <type machine> est la déclaration d'un type de nom <id M>,
- ayant comme longueur <constante 1> et comme contrainte
- d'alignement <constante 2>.
- La longueur est exprimée en une unité arbitraire,
- généralement celle du type le plus court. La contrainte
- d'alignement porte sur les adresses d'implantation en
- mémoire pour les objets du type.

```
<types Pascal> --> PASCAL ;
                   integer : <id M> ;
                   boolean : <id M> ;
                   char    : <id M> ;
                   real    : <id M> ;
                   pointer : <id M> ;
```

- Cette partie indique comment seront mis en oeuvre les types
- de base du langage Pascal. Chaque identificateur doit être
- le nom d'un type défini dans la rubrique précédente.

```
<registres>          --> REGISTRES ;
                         <reg. de référence> <sous-registres>
<reg. de référence>  --> REFREG ; <ref reg>*
<ref reg>            --> <nom générique R> : <id T> ;
<nom générique>      --> <id R>
<nom générique>      --> <id R> [ <constante 1> , <constante 2> ]
```

- La partie registres décrit les registres de la machine. On
- décrit d'abord les registres de référence (ceux dont la
- capacité est la plus élevée) et ensuite leurs sous-registres.
- On remarquera le mécanisme <nom générique> qui permet de
- raccourcir considérablement la description, lorsque les
- registres ont des noms reguliers tels que R0,R1,...,R15.
- <ref reg> est la déclaration d'un ou plusieurs registres
- de type <id>. Si <nom générique R> est égal à <id R>, il
- s'agit de la déclaration d'un seul registre. Sinon, il s'agit
- de la déclaration de plusieurs registres dont les noms sont :

```
-    <id R><constante 1>
-    <id R><constante 1>+1
-      ...
-    <id R><constante 2>
```

```
<sous-registres> --> SOUSREG ; <sous-reg>*
<sous-reg>       --> <nom générique S> :
                     <nom générique R> ( <constante> , <id T> );
```

- Déclaration des sous-registres :

```
-   <nom générique S> est le nom des sous-registres déclarés
-   <nom générique R> est le nom des registres de référence
-   <constante> spécifie le bit initial de définition
-   <id T> est le type du sous-registre.
```

```
<modèles> --> <déplacements> <constantes> <classes reg>
              <classes mém> <poss adr> <classes pa>
```

- Définition des modèles de descripteur.

```
<déplacements> --> DEPLACEMENTS ; <déplacement>*
<déplacement>  --> <id D> : <id T> , ( <constante 1> ,
                                       <constante 2> ) ;
```

- Modèles de description associés aux déplacements utilisés par
- la machine. Un déplacement est défini par un type de nom
- <id T> et par un intervalle <constante 1> .. <constante 2>.

```
<constantes> --> <entiers> <intervalles> ;
<entiers>    --> ENTIERS ; <entier>*
<entier>     --> <id E> : <id T> , <constante> ;
```

- Modèles de description de constantes entières.
- <entier> donne le nom <id E> à un descripteur de valeur
- <constante> et de type <id T>.

```
<intervalles> --> INTERVALLES ; <intervalle>*
<intervalle>  --> <id I> : <id T> , <constante 1> ,
                            <constante 2> ;
```

- Modèles d'intervalles.
- <intervalle> définit un descripteur pour les constantes de
- l'intervalle <constante 1> .. <constante 2>. Ces valeurs ont
- pour type <id T>.

```
<classes reg> --> CLASSREG ; <classreg>*
<classreg>    --> <id C> : <id T> , <id R>* ;
```

- Cette rubrique permet de regrouper des registres qui jouent
- rôle symétrique dans les instructions (ex : registres de
- base).
- <classreg> donne le nom <id C> à l'ensemble de registres
- <id R>*, et lui associe le type <id T>.

```
<classes mém> --> CLASSMEM ; <classmem>*
<classmem>    --> <id C> : <id T> , <id D> , <id B> , <id I> ;
```

- Description des modes d'adressage de la machine.
- <classmem> donne le nom <id C> à un modèle de descripteur

- mémoire de type <id T> dont les composants sont :
-     <déplacement> = <id D>
-     <base>   = <id B>
-     <index>   = <id I>

<poss adr> --> POSSADR ; <possadr>*
<possadr> --> <id P> : <id M> , Mode = <constante> ,
                Cout = <constante> , Taille = <constante> ;

- Cette rubrique associe un certain nombre d'informations à
- chaque mode d'adressage :
-     mode : code d'identification du mode
-     cout : dû à son utilisation
-     taille : espace occupé dans l'instruction.
- Une <possadr> donne le nom <id C> à l'association d'un modèle
- <id M> et des 3 informations qui lui correspondent.

<classes pa> --> CLASSPA ; <classpa>*
<classpa>   --> <id C> : <id P>* ;

- Définition des classes de possibilités d'adressage.
- <classpa> donne le nom <id C> à la liste des possibilités
- d'adressage définies par <id P>*.

<instructions>   --> INSTRUCTIONS ; <opérateurs>*
<opérateurs>   --> OPERATEURS : <id O>* ; <opérateur type>*
<opérateur type> --> <po>*
<opérateur type> --> <type> ; <po>*
<type>   --> TYPE : <id T1> , <id T2>
<po>   --> <id G> , <id D> , Cout = <constante 1> ,
                     Result = <constante> ,
                     Codop = <constante> ;

- Description des instructions arithmétiques-logiques.
- <opérateurs> définit toutes les manières de réaliser les
- opérateurs Pascal <id O>*.
- <opérateur type> définit ces différentes possibilités pour un
- type donné.
- <type> spécifie le type des opérandes mais peut être
- implicite, en fonction de l'opération.
- <po> est une possibilité de réalisation de ces opérations.

<affectations> --> AFFECTATIONS ; <opérateurs>*

- Définition des instructions d'affectation et de conversion

- disponibles dans la machine. Si le nom de l'opérateur n'est
- pas prédéfini, on suppose qu'il s'agit d'une conversion.

```
<branchements> --> BRANCHEMENTS ; <branchement>*
<branchement>  --> <id B> : <id M> , Codop = <constante> ,
                   Taille = <constante> ,
                   Conditions = <condition>* ;
<condition>    --> eq
<condition>    --> ne
  ...
<condition>    --> ge
<condition>    --> t
<condition>    --> nil
```

- Description des instructions de branchement.
- <branchement> définit un branchement <id B> en lui associant
- les informations suivantes :
-     un modèle décrivant le mode d'adressage qu'il utilise,
-     un code opération,
-     la taille mémoire de l'instruction,
-     une liste de conditions.

ANNEXE B : DESCRIPTION DU PROCESSEUR MOTOROLA 68000.

```
DESCRIPTION: MC68000;

EQUIVALENCES;
    % Tailles mémoire en octets ;
        Taille0 : 0;
        Taille2 : 2;
        Taille4 : 4;
        Taille6 : 6;
    % Intervalles d'entiers ;
        Int3 : (1,8);
        Int8 : (-128,127);
        Int16 : (-32768,32767);
        Int32 : (-2147483648,2147483647);
    % Descripteurs de résultats implicites ;
        ResultImplG : 'implicite , 'g;
        ResultImplD : 'implicite , 'd;
    % Codes des modes d'adressage ;
        ModeRegD : 0;
        ModeRegA : 1;
        ModeRegIndirect : 2;
        ModeDirect : 5;
        ModeIndexe : 6,1;
        ModeAbsolu : 7,1;
        ModeImm16 : 7,4;
        ModeImm8 : 7,4;

TYPES;
    Base : 8; % Nombre de bits de l'unité d'adressage ;
    MACHINE;
        Octet : 1,1;
        DemiMot : 2,2;
        Mot : 4,2;
    PASCAL;
        integer : DemiMot;
        boolean : Octet;
        char : Octet;
        real : Mot;
        pointer : Mot;

REGISTRES;
    REFREG;
        dl[0,7] : Mot;
```

```
        al[0,7]  : Mot;
    SOUSREG;
        % Sous-registres 16 bits (word) ;
        dwh[0,7]  : dl[0,7]  (0,DemiMot); % reg. donnée poids forts ;
        dwl[0,7]  : dl[0,7]  (16,DemiMot); % reg. donnée poids faibles ;
        aw[0,7]  : al[0,7]  (16,DemiMot); % reg. adresse poids faibles;
        % Sous-registres 8 bits (byte) ;
        dbll[0,7]: dwl[0,7]  (8,Octet);
        dblh[0,7]: dwl[0,7]  (0,Octet);
        dbhl[0,7]: dwh[0,7]  (8,Octet);
        dbhh[0,7]: dwh[0,7]  (0,Octet);

MODELES;
    DEPLACEMENTS;
        Depl8  : Octet,Int8;
        Depl16: DemiMot,Int16;
        Depl32: Mot,Int32;
    CONSTANTES;
        ENTIERS;
            % constante zero pour les 3 types machine ;
            Zero  : DemiMot,0;
            ZeroB  : Octet,0;
            ZeroL  : Mot,0;
        INTERVALLES;
            Interv3  : DemiMot,Int3;  % utilisé dans les opérations "quick" ;
            Interv3L  : Mot,Int3;
            Interv8  : DemiMot,Int8;
            Interv8B  : Octet,Int8;
            Interv8L  : Mot,Int8;
            Interv16  : DemiMot,Int16;
            Interv16L  : Mot,Int16;
            Interv32  : Mot,Int32;
    CLASSREG;
        RegDW  : DemiMot,dwl0,dwl1,dwl2,dwl3,dwl4,dwl5,dwl6,dwl7;
        RegDB  : Octet,dbll0,dbll1,dbll2,dbll3,dbll4,dbll5,dbll6,dbll7;
        RegDL  : Mot,dl0,dl1,dl2,dl3,dl4,dl5,dl6,dl7;
        RegAW  : DemiMot,aw0,aw1,aw2,aw3,aw4,aw5,aw6,aw7;
        RegAL  : Mot,al0,al1,al2,al3,al4,al5,al6,al7;
    CLASSMEM;
        % adressage registre indirect ;
        RegIndirect  : DemiMot,Zero,RegAL,nil;
        RegIndirectB  : Octet,Zero,RegAL,nil;
        RegIndirectL  : Mot,Zero,RegAL,nil;
        % adressage basé ;
        Direct  : DemiMot,Depl16,RegAL,nil;
        DirectB  : Octet,Depl8,RegAL,nil;
        DirectL  : Mot,Depl16,RegAL,nil;
        % adressage indexé ;
        Indexe  : DemiMot,Depl8,RegAL,RegAW;
```

```
    IndexeB : Octet.Depl8.RegAL.RegAW;
    IndexeL : Mot.Depl8.RegAL.RegAW;
    % adressage absolu :
    Absolu : DemiMot.Depl32.nil.nil;
POSSADR;
    % Définition des options par défaut :
        Mode := ModeImpl;
        Cout := (0.0);
        Taille := Taille0;
    % Modes d'adresage (Pa) pour les registres :
        PaRegDW : RegDW.Mode=ModeRegD;
        PaRegDB : RegDB.Mode=ModeRegD;
        PaRegDL : RegDL.Mode=ModeRegD;
        PaRegAW : RegAW.Mode=ModeRegA;
        PaRegAB : RegAB.Mode=ModeRegA;
        PaRegAL : RegAL.Mode=ModeRegA;
    % Rédéfinition option taille :
        Taille := Taille2;
    % Pa pour les valeurs immédiates :
        PaImm16 : Interv16.Mode=ModeImm16.Cout=(1.2);
        PaImm16L: Interv16L.Mode=ModeImm16.Cout=(1.2);
        PaImm8 : Interv8B.Mode=ModeImm8.Cout=(1.2);
    % Pa pour les positions mémoire :
        PaRegIndirect : RegIndirect.Mode=ModeRegIndirect.Cout=(0.2);
        PaRegIndirectB: RegIndirectB.Mode=ModeRegIndirect.Cout=(0.2);
        PaRegIndirectL: RegIndirectL.Mode=ModeRegIndirect.Cout=(0.4);
        PaDirect : Direct.Mode=ModeDirect.Cout=(1.4);
        PaDirectB: DirectB.Mode=ModeDirect.Cout=(1.4);
        PaDirectL: DirectL.Mode=ModeDirect.Cout=(1.6);
        PaIndexe : Indexe.Mode=ModeIndexe.Cout=(1.5);
        PaIndexeB: IndexeB.Mode=ModeIndexe.Cout=(1.5);
        PaIndexeL: IndexeL.Mode=ModeIndexe.Cout=(1.7);
        PaAbsolu : Absolu.Mode=ModeAbsolu.Cout=(2.8);
    % Rédéfinition option taille :
        Taille := Taille0;
    % Modes d'adressage implicites :
        % Constantes implicites :
        PaZeroB : ZeroB;
        PaZero : Zero;
        PaZeroL : ZeroL;
        % Immédiats implicites ;
        PaImplImm3 : Interv3;
        PaImplImm3L: Interv3L;
        PaImplImm8 : Interv8;
        PaImplImm8B: Interv8B;
        PaImplImm8L: Interv8L;
        PaImplImm16 : Interv16;
        PaImplImm32 : Interv32;
```

```
        % Registres implicites ;
        PaImplRegDW : RegDW;
        PaImplRegDL : RegDL;
        PaImplRegAW : RegAW;
        PaImplRegAL : RegAL;
        PaImplRegDB : RegDB;
        % Pa nulle, utilisée pour les op. unaires ou sans opérandes ;
        PaNil : nil;
CLASSPA;
    % effective address ;
        EAW : PaRegDW,PaRegAW,PaImm16,PaRegIndirect,PaDirect,PaIndexe;
        EAB : PaRegDB,PaImm8,PaRegIndirectB,PaDirectB,PaIndexeB;
        EAB2: PaRegDB,PaRegAB,PaImm8,PaRegIndirectB,PaDirectB,PaIndexeB;
        EAL  :  PaRegDL,PaRegAL,PaImm16L,PaRegIndirectL,PaDirectL,PaIndexeL;
    % data alterable ;
        DataAltW : PaRegDW,PaRegAW,PaRegIndirect,PaDirect,PaIndexe;
        DataAltB : PaRegDB,PaRegIndirectB,PaDirectB,PaIndexeB;
        DataAltL : PaRegDL,PaRegAL,PaRegIndirectL,PaDirectL,PaIndexeL;
    % alterable memory ;
        AltMemW : PaRegIndirect,PaDirect,PaIndexe;
        AltMemB : PaRegIndirectB,PaDirectB,PaIndexeB;
        AltMemL : PaRegIndirectL,PaDirectL,PaIndexeL;
    % registres de données ;
        DRGW : PaImplRegDW;
        DRGB : PaImplRegDB;
        DRGL : PaImplRegDL;
    % registres d'adresses ;
        ARGW : PaImplRegAW;
        ARGL : PaImplRegAL;
    % registres de données et d'adresses ;
        DARGW : PaRegDW,PaRegAW;
        DARGB : PaRegDB,PaRegAB ;
        DARGL : PaRegDL,PaRegAL;
    % valeurs immédiates ;
        CM3 : PaImplImm3;
        CM3L: PaImplImm3L ;
        CM8 : PaImplImm8;
        CM8B : PaImplImm8B;
        CM8L: PaImplImm8L ;
        CM16 : PaImplImm16;
        CM32 : PaImplImm32 ;
        ZERO : PaZero;
        ZEROB : PaZeroB;
        ZEROL : PaZeroL;
    % pas d'opérande ;
        Nil : PaNil;
```

```
INSTRUCTIONS;
   % Options par défaut ;
      Cout := (2,4);
      Taille := Taille2;
      Result := ResultImplG;
   OPERATEURS : PlusInt ;
      Plus1 : DRGW,EAW,Cout=(2,2),Codop='ADD1;
      Plus2 : DRGW,AltMemW,Result=ResultImplD,Codop='ADD2;
      Plus3 : ARGW,EAW,Codop='ADDA;
      Plus4 : DARGW,CM16,Cout=(4,4),Taille=Taille4,Codop='ADDI;
      Plus5 : AltMemW,CM16,Cout=(4,6),Taille=Taille4,Codop='ADDI;
      Plus6 : DARGW,CM3,Cout=(2,2),Codop='ADDQ;
      Plus7 : AltMemW,CM3,Codop='ADDQ;
   OPERATEURS: PlusAdr ;
      PlusA1 : DRGL,EAL,Codop='ADD1L;
      PlusA2 : DRGL,AltMemL,Cout=(2,6),Result=ResultImplD,Codop='ADD2L;
      PlusA3 : ARGL,EAL,Codop='ADDAL;
      PlusA4 : DARGL,CM32,Cout=(6,8),Taille=Taille6,Codop='ADDIL;
      PlusA5 : AltMemL,CM32,Cout=(6,10),Taille=Taille6,Codop='ADDIL;
      PlusA6 : DARGL,CM3L,Codop='ADDQL;
      PlusA7 : AltMemL,CM3L,Cout=(2,6),Codop='ADDQL;
   OPERATEURS : MinusInt ;
      Moins1 : IDEM Plus1,Codop='SUB1;
      Moins2 : IDEM Plus2,Codop='SUB2;
      Moins3 : IDEM Plus3,Codop='SUBA;
      Moins4 : IDEM Plus4,Codop='SUBI;
      Moins5 : IDEM Plus5,Codop='SUBI;
      Moins6 : IDEM Plus6,Codop='SUBQ;
      Moins7 : IDEM Plus7,Codop='SUBQ;
   OPERATEURS : MultInt ;
      Mult1 : DRGW,EAW,Result=ResultSurReg('g,Mot),Cout=(2,35),
                 Codop='MULS;
   OPERATEURS : DivInt ;
      TYPE: Mot,DemiMot;
      Div1 : DRGL,EAW,Result=ResultSousReg('g,2,DemiMot),Cout=(1,79),
                 Codop='DIVS ;
   OPERATEURS : CompInt ; % moins unaire entier ;
      Neg1 : DARGW,Nil,Cout=(2,2),Codop='NEG;
      Neg2 : AltMemW,Nil,Codop='NEG;
   OPERATEURS: EqualInt,NotEqualInt,LessInt,GreaterInt,LessEqualInt,
                 GreaterEqualInt ;
      TYPE: DemiMot,DemiMot;
      Comp1 : DRGW,EAW,Cout=(2,2),Result=nil,Codop='CMP;
      Comp2 : DARGW,CM16,Result=nil,Cout=(4,4),Taille=Taille4,
                 Codop='CMPI;
      Comp3 : AltMemW,CM16,Result=nil,Cout=(4,6),Taille=Taille4,
                 Codop='CMPI;
      Comp4 : ARGW,EAW,Cout=(2,3),Result=nil,Codop='CMPA;
```

```
OPERATEURS: EqualBool,EqualChar,NotEqualBool,NotEqualChar,
              LessBool,LessChar,GreaterBool,GreaterChar,
              LessEqualBool,LessEqualChar,
              GreaterEqualBool,GreaterEqualChar ;
    TYPE: Octet,Octet;
    CompB1 : DRGB,EAB,Cout=(2,2),Result=nil,Codop='CMPB;
    CompB2 : DARGB,CM8B,Result=nil,Cout=(4,4),Taille=Taille4,
              Codop='CMPBI;
    CompB3 : AltMemB,CM8B,Result=nil,Cout=(4,6),Taille=Taille4,
              Codop='CMPBI;
OPERATEURS : And ;
    TYPE: Octet,Octet;
    And1 : DRGB,EAB2,Cout=(2,2),Codop='AND1;
    And2 : DRGB,AltMemB,Result=ResultImplD,Codop='AND2;
    And3 : AltMemB,CM8B,Cout=(4,6),Taille=Taille4,Codop='ANDI;
    And4 : DARGB, CM8B, Cout=(4,4), Taille=Taille4, Codop='ANDI ;
OPERATEURS : Or ;
    TYPE: Octet,Octet;
    Or1 : IDEM And1,Codop='OR1;
    Or2 : IDEM And2,Codop='OR2;
    Or3 : IDEM And3,Codop='ORI;
    Or4 : IDEM And4, Codop='ORI;
OPERATEURS : Not ;
    TYPE: Octet;
    Not1 : DARGB,Nil,Cout=(2,2),Codop='NOT;
    Not2 : AltMemB,Nil,Codop='NOT;

AFFECTATIONS ;
  % Options par défaut ;
    Cout := (2,2);
    Taille := Taille2 ;
    Result := ResultImplG ;
  OPERATEURS: AffInt ;
    TYPE: DemiMot,DemiMot;
    Aff1 : DataAltW,EAW,Codop='MOVE;
    Aff2 : DRGW,CM8,Codop='MOVEQ;
    Aff3 : DARGW,ZERO,Codop='CLR;
    Aff4 : AltMemW,ZERO,Cout=(2,4),Codop='CLR;
    Aff5 : ARGW, EAW, Codop='MOVEA ;
  OPERATEURS: AffReal,AffPointer ;
    % Définition locale de l'option Cout ;
      Cout := (2,2) ;
    TYPE: Mot,Mot;
    AffL1 : ARGL,EAL,Codop='MOVEA;
    AffL2 : DARGL,ZEROL,Cout=(2,3),Codop='CLRL;
    AffL3 : AltMemL,ZEROL,Cout=(2,6),Codop='CLRL;
    AffL4 : DataAltL,EAL,Codop='MOVEL ;
    AffL5 : DRGL,CM8L,Codop='MOVEQL ;
```

```
    OPERATEURS: AffBool, AffChar ;
            Cout := (2,2) ;
        TYPE: Octet,Octet;
        AffB1  : DataAltB,EAB,Codop='MOVEB;
        AffB2  : DRGB,CM8B,Codop='MOVEQB;
        AffB3  : DARGB,ZEROB,Codop='CLRB;
        AffB4  : AltMemB,ZEROB,Cout=(2,4),Codop='CLRB;
    OPERATEURS: ConvWL ; % conversion demi-mot -> mot ;
        TYPE: DemiMot,Mot;
        ConvWL1  : DRGW,Nil,Result=ResultSurReg('g,Mot),Cout=(2,2),
                   Codop='EXT;
    OPERATEURS: ConvLW ; % conversion mot -> demi-mot ;
        TYPE: Mot,DemiMot;
        ConvLW1  : DRGL,Nil,Result=ResultSousReg('g,2,DemiMot),Cout=(0,0),
                   Codop=nil;

FINDESC;
```

ANNEXE C : DESCRIPTION DU PROCESSEUR LSI-11.

```
DESCRIPTION: LSI11;

EQUIVALENCES;
    % Intervalles d'entiers ;
        Int0 : (0,0);
        Int8S : (-128,127);
        Int16S: (-32768,32767);
        Int16 : (0,65535);
    % Descripteurs des différents types de résultat ;
        ResultImplG : 'Implicite.'g;
        ResultImplD : 'Implicite.'d;

TYPES;
    MACHINE;
        Octet: 1,1;
        Mot  : 2,2;
        MotL: 4,2;
    PASCAL;
        integer : Mot;
        boolean : Octet;
        char : Octet;
        real : MotL;
        pointer : Mot;

REGISTRES;
    REFREG;
        RL[0,2] : MotL;
        SP : Mot;
    SOUSREG;
        R0 : RL0 (0,Mot);
        R1 : RL0 (16,Mot);
        R2 : RL1 (0,Mot);
        R3 : RL1 (16,Mot);
        R4 : RL2 (0,Mot);
        R5 : RL2 (16,Mot);
        RB[0,5] : R[0,5] (8,Octet);

MODELES;
    DEPLACEMENTS;
        Depl0 : Mot.Int0;
        Depl16: Mot.Int16;
```

```
CONSTANTES;
    ENTIERS;
        ZeroW  :  Mot,0;
        ZeroB  :  Octet,0;
        UnW  :  Mot,1;
    INTERVALLES;
        Interv8S  :  Octet,Int8S;
        Interv16S:  Mot,Int16S;
CLASSREG;
    RegL  :  MotL,RL0,RL1,RL2;
    RegW  :  Mot,R0,R1,R2,R3,R4,R5;
    RegEW:  Mot,R0,R2,R4;
    RegB  :  Octet,RB0,RB1,RB2,RB3,RB4,RB5;
CLASSMEM;
    IndexW  :  Mot,Depl16,RegW,nil;
    IndexB  :  Octet,Depl16,RegW,nil;
    RegIndirectW  :  Mot,Depl0,RegW,nil;
    RegIndirectB  :  Octet,Depl0,RegW,nil;
    IndirectW  :  Mot,Depl0,IndexW,nil;
    IndirectB  :  Octet,Depl0,IndexW,nil;
POSSADR;
    % Définition des options par defaut ;
        Mode  :=  0;
        Cout  :=  (0,0);
        Taille  :=  0;
    % Modes d'adressage (Pa) pour les registres ;
        PaRegL  :  RegL;
        PaRegW  :  RegW;
        PaRegEW:  RegEW;
        PaRegB  :  RegB;
    % Rédéfinition de l'option Taille ;
        Taille  :=  2;
    % Pa pour les valeurs immédiates ;
        PaImm16:  Interv16S,Mode=2,Cout=(2,140);
        PaImm8  :  Interv8S,Mode=2,Cout=(2,105);
    % Pa pour les positions mémoire ;
        PaIndexSW  :  IndexW,Mode=6,Cout=(2,420);
        PaIndexDW  :  IndexW,Mode=6,Cout=(2,490);
        PaIndexSB  :  IndexB,Mode=6,Cout=(2,385);
        PaIndexDB  :  IndexB,Mode=6,Cout=(2,455);
        PaRegIndirectSW  :  RegIndirectW,Mode=1,Cout=(0,140);
        PaRegIndirectDW  :  RegIndirectW,Mode=1,Cout=(0,195);
        PaRegIndirectSB  :  RegIndirectB,Mode=1,Cout=(0,110);
        PaRegIndirectDB  :  RegIndirectB,Mode=1,Cout=(0,160);
        PaIndirectSW  :  IndirectW,Mode=7,Cout=(2,630);
        PaIndirectDW  :  IndirectW,Mode=7,Cout=(2,630);
        PaIndirectSB  :  IndirectB,Mode=7,Cout=(2,595);
        PaIndirectDB  :  IndirectB,Mode=7,Cout=(2,595);
    % Rédéfinition de la taille ;
        Taille  :=  0;
```

```
      % modes d'adressage implicites :
          % constantes immédiates :
         PaUnW : UnW;
         PaZeroW : ZeroW;
         PaZeroB : ZeroB;
      % mode d'adressage nul :
         PaNil : nil;
   CLASSPA:
      % modes "source" :
         SourceW : PaRegW,PaImm16,PaIndexSW,PaRegIndirectSW,PaIndirectSW;
         SourceB : PaRegB,PaImm8,PaIndexSB,PaRegIndirectSB,PaIndirectSB;
      % modes "destination" :
         DestinMemW : PaIndexDW,PaRegIndirectDW,PaIndirectDW;
         DestinMemB : PaIndexDB,PaRegIndirectDB,PaIndirectDB;
      % modes registres :
         DestinRegW : PaRegW;
         DestinRegEW: PaRegEW;
         DestinRegL : PaRegL;
         DestinRegB : PaRegB;
      % modes constantes implicites :
         UNW          : PaUnW;
         ZEROW        : PaZeroW;
         ZEROB        : PaZeroB;
      % mode nul :
         Nil          : PaNil;

INSTRUCTIONS:
   % Définition options par défaut :
      Taille := 2;
      Result := ResultImplD;
   OPERATEURS: PlusInt,PlusAdr;
      Plus1 : SourceW,DestinMemW,Cout=(2,420),Codop='ADD;
      Plus2 : SourceW,DestinRegW,Cout=(2,350),Codop='ADD;
      Plus3 : UNW,DestinMemW,Cout=(2,490),Codop='INC;
      Plus4 : UNW,DestinRegW,Cout=(2,420),codop='INC;
   OPERATEURS: MinusInt;
      Moins1 : DestinMemW,SourceW,Cout=(2,420),Result=ResultImplG,
                  Codop='SUB;
      Moins2 : DestinRegW,SourceW,Cout=(2,350),Result=ResultImplG,
                  Codop='SUB;
      Moins3 : DestinMemW,UNW,Cout=(2,490),Result=ResultImplG,Codop='DEC;
      Moins4 : DestinRegW,UNW,Cout=(2,420),Result=ResultImplG,Codop='DEC;
   OPERATEURS: MultInt;
      Mult1 : SourceW,DestinRegEW,Cout=(2,6400),Codop='MUL,
                 Result=ResultSurReg('d,MotL);
   OPERATEURS: DivInt;
      Div1 : DestinRegL,SourceW,Result=ResultSousReg('g,1,Mot),
                Cout=(2,7800),Codop='DIV;
   OPERATEURS: ComplInt;
      Neg1 : DestinMemW,Nil,Cout=(2,455),Codop='NEG;
```

```
        Neg2  :  DestInRegW,Nil,Cout=(2,420),Codop='NEG;
    OPERATEURS:  EqualInt,EqualAdr,NotEqualInt,NotEqualAdr,LessInt,
                 LessAdr,GreaterInt,GreaterAdr,LessEqualInt,
                 LessEqualAdr,GreaterEqualInt,GreaterEqualAdr;
        Comp1  :  SourceW,DestInMemW,Cout=(2,315),Result=nil,Codop='CMP;
        Comp2  :  SourceW,DestInRegW,Cout=(2,350),Result=nil,Codop='CMP;
    OPERATEURS:  EqualBool,EqualChar,NotEqualBool,NotEqualChar,
                 LessBool,LessChar,GreaterBool,GreaterChar,
                 LessEqualBool,LessEqualChar,GreaterEqualBool,
                 GreaterEqualChar;
        CompB1  :  SourceB,DestInMemB,Cout=(2,280),Result=nil,Codop='CMPB;
        CompB2  :  SourceB,DestInRegB,Cout=(2,315),Result=nil,Codop='CMPB;
    OPERATEURS:  And;
        And1  :  SourceB,DestInMemB,Cout=(2,385),Codop='AND;
        And2  :  SourceB,DestInRegB,Cout=(2,385),Codop='AND;
    OPERATEURS:  Or;
        Or1  :  SourceB,DestInMemB,Cout=(2,385),Codop='BISB;
        Or2  :  SourceB,DestInRegB,Cout=(2,385),Codop='BISB;
    OPERATEURS:  Not;
        Not1  :  SourceB,DestInMemB,Cout=(2,420),Codop='COMB;
        Not2  :  SourceB,DestInRegB,Cout=(2,385),Codop='COMB;

AFFECTATIONS;
    % Définition options par défaut ;
        Taille := 2 ;
        Result := ResultImplG ;
    OPERATEURS:  AffInt,AffAdr;
        Aff1  :  DestInMemW,SourceW,Cout=(2,245),Codop='MOV;
        Aff2  :  DestInRegW,SourceW,Cout=(2,350),Codop='MOV;
        Aff3  :  DestInMemW,ZEROW,Cout=(2,420),Codop='CLR;
        Aff4  :  DestInRegW,ZEROW,Cout=(2,385),Codop='CLR;
    OPERATEURS:  AffBool,AffChar;
        AffB1  :  DestInMemB,SourceB,Cout=(2,385),Codop='MOVB;
        AffB2  :  DestInRegB,SourceB,Cout=(2,385),Codop='MOVB;
        AffB3  :  DestInMemB,ZEROB,Cout=(2,420),Codop='CLRB;
        AffB4  :  DestInRegW,ZEROB,Cout=(2,385),Codop='CLRB;
    OPERATEURS:  ConvWL;  % conversion mot -> double-mot ;
        TYPE:  Mot,MotL;
        ConvWL1  :  DestInRegW,Nil,Result=ResultSurReg('g,MotL),Cout=(2,595),
                    Codop='SXT;
    OPERATEURS:  ConvLW;  % conversion double-mot -> mot ;
        TYPE:  Mot,MotL;
        ConvLW1  :  DestInRegW,Nil,Result=ResultSousReg('g,1,Mot),Cout=(0,0),
                    Codop=nil;

FINDESC;
```

ANNEXE D : REPRESENTATION INTERNE GEMME.

Un programme est représenté sous la forme d'un arbre abstrait syntaxique à l'intérieur du système GEMME. Les noeuds sont divisés en 11 classes regroupant des opérateurs du même type. Nous présentons ci-après ces différentes classes, à l'aide d'une représentation graphique.

1. Feuilles.

```
                       ---------------
Classe                 ! 'Feuille    !
                       ---------------
Descripteur opérande   !             !
                       ---------------
```

Les feuilles correspondent aux opérandes élémentaires du programme (identificateurs).

2. Opérateurs d'affectation.

```
                    ---------------
Classe              ! 'OpAffect   !
                    ---------------
Opérateur           !             !
                    ---------------
SousExpr (*)        !             !
                    ---------------
Fils                !      !      !
                    -------!-------
                           v
```

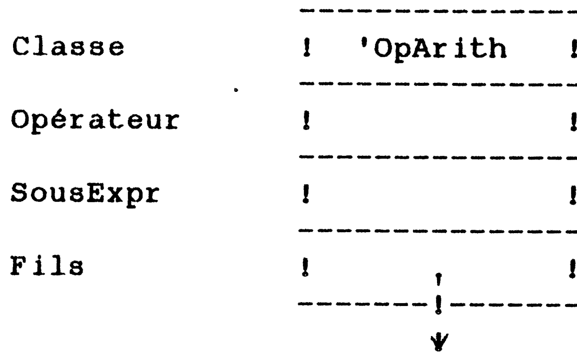
(‹arbre partie gauche› ‹arbre partie droite›)

(*) Indique si le sous-arbre est une sous-expression commune

Les opérateurs d'affectation sont les suivants :

:=i	entiers	:=c	caractères
:=r	réels	:=p	pointeurs
:=b	booléens	:=l	structures

3. Opérateurs arithmétiques.

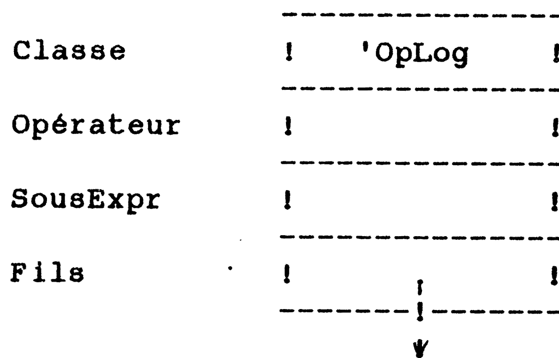


(<opérande gauche> <opérande droit>)

Cette classe est composée par les opérateurs suivants :

+i	addition entière	+r	addition réelle
-i	soustraction entière	-r	soustraction réelle
*i	multiplication entière	*r	multiplication réelle
-ui	moins unaire entier	-ur	moins unaire réel
div	division entière	/	division réelle
mod	modulo entier		

4. Opérateurs logiques.



(<opérande gauche> <opérande droit>)

Trois opérateurs logiques existent : "or", "and" et "not".

5. Opérateurs de comparaison.

```
                 ---------------
Classe           !   'OpComp   !
                 ---------------
Opérateur        !             !
                 ---------------
SousExpr         !             !
                 ---------------
Fils             !      !      !
                 -------!-------
                        V
```

(<opérande gauche> <opérande droit>)

Les opérateurs de comparaison sont les suivants :

=i	égal entier	<>i	différent entier
=r	égal réel	<>r	différent réel
=b	égal booléen	<>b	différent booléen
=c	égal caractères	<>c	différent caractères
=p	égal pointeurs	<>p	différent pointeurs
=s	égal ensembles	<>s	différent ensembles
=st	égal chaines	<>st	différent chaines
<i	inférieur entier	>i	supérieur entier
<r	inférieur réel	>r	supérieur réel
<c	inférieur caractères	>c	supérieur caractères
<st	inférieur chaines	>st	supérieur chaines
<=i	inférieur-égal entier	>=i	supérieur-égal entier
<=r	inférieur-égal réel	>=r	supérieur-égal réel
<=c	inférieur-égal caractères	>=c	supérieur-égal caractères
<=s	inclusion ensembles	>=s	inclusion ensembles
<=st	inférieur-égal chaines	>=st	supérieur-égal chaines

6. Opérateurs ensemblistes.

```
                 ---------------
Classe           !   'OpEns    !
                 ---------------
Opérateur        !             !
                 ---------------
SousExpr         !             !
                 ---------------
Fils             !      !      !
                 -------!-------
                        V
```

(<opérande gauche> <opérande droit>)

Quatre opérateurs composent cette classe :

+s	union	*s	intersection
-s	soustraction	in	appartenance

7. Fonctions prédéfinies.

```
                   ---------------
Classe             !  'Fonction  !
                   ---------------
Id. fonction       !             !
                   ---------------
SousExpr           !             !
                   ---------------
Fils               !      ,------!--> (<opérande>)
                   ---------------
```

Ces fonctions sont les suivantes :

absi	valeur absolue entière	sqri	carré entier
absr	valeur absolue réelle	sqrr	carré réel
sin	sinus	exp	exponentiel
cos	cosinus	ln	logarithme
arctan	arc tangent	sqrt	racine carrée
succ	successeur	eof	fin de fichier
pred	prédécesseur	eoln	fin de ligne

8. Opérateurs de conversion.

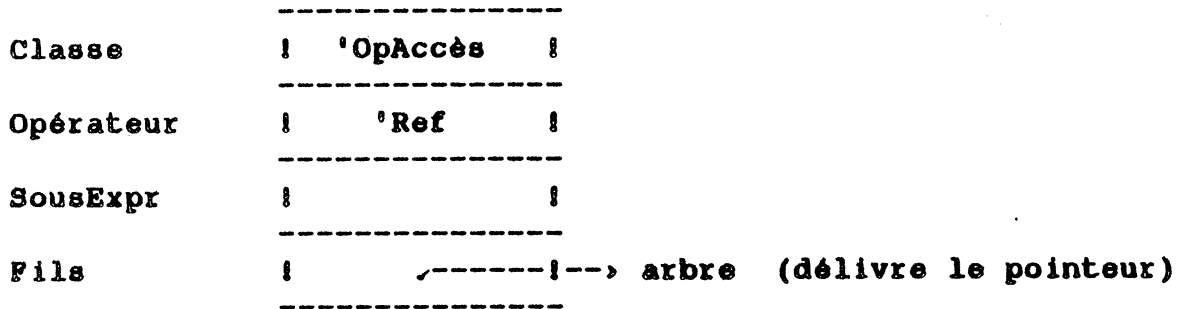
```
                   ---------------
Classe             !   'OpConv   !
                   ---------------
Id. fonction       !             !
                   ---------------
SousExpr           !             !
                   ---------------
Fils               !      ,------!--> (<opérande>)
                   ---------------
```

Cette classe est constituée par six opérateurs :

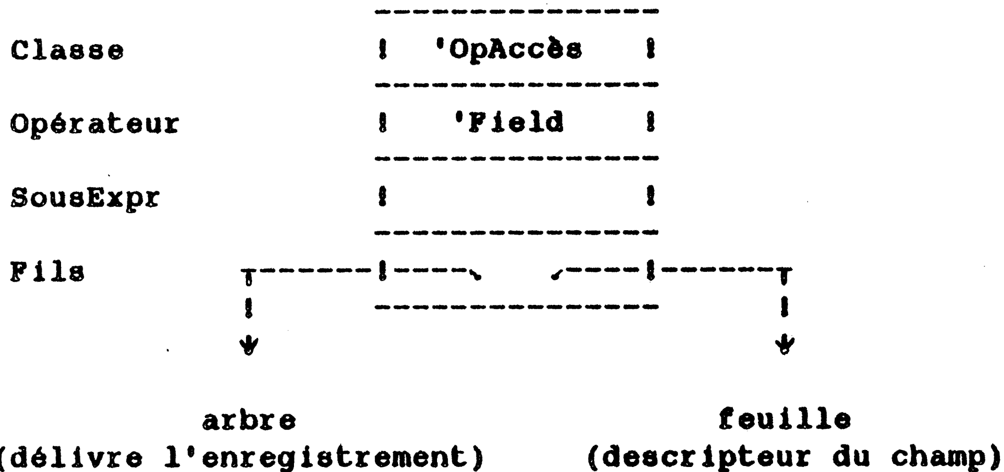
trunc	réel	-->	entier
round	réel	-->	entier (arrondi)
ord	scalaire	-->	entier
chr	entier	-->	caractère
real	entier	-->	réel
pointer	entier	-->	adresse

9. Opérateurs d'accès.

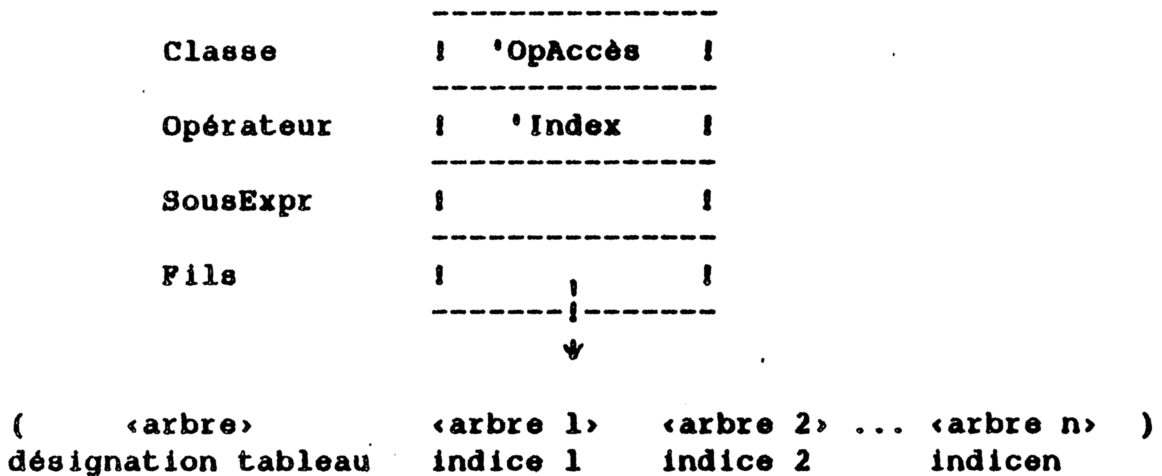
a) Indirection à travers un pointeur (↑).



b) Champ d'un enregistrement (.).



c) Elément d'un tableau([ ]).



10. Instructions.

```
                  ---------------
Classe            ! 'Statement   !
                  ---------------
Instruction       !   'Begin     !
                  ---------------
Attribut          !    nil       !
                  ---------------
Fils              !      !       !
                  -------!-------
                         v

(  <instr 1>  <instr 2>   .......    <instr n>   )
```

```
                  ---------------
Classe            ! 'Statement   !
                  ---------------
Instruction       !     'If      !
                  ---------------
Attribut          !    nil       !
                  ---------------
Fils              !      !       !
                  -------!-------
                         v

( <condition>    <partie then>    <partie else> )
```

```
                  ---------------
Classe            ! 'Statement   !
                  ---------------
Instruction       !    'goto     !
                  ---------------
Attribut          ! l'étiquette  !
                  ---------------
Fils              !    nil       !
                  ---------------
```

```
                      ---------------
Classe               ! 'Statement  !
                      ---------------
Instructions         !  'Repeat    !
                      ---------------
Attribut             !    nil      !
                      ---------------
Fils                 !      !      !
                      -------!-------
                             v

   ( ‹liste d'instructions›   ‹expression› )
```

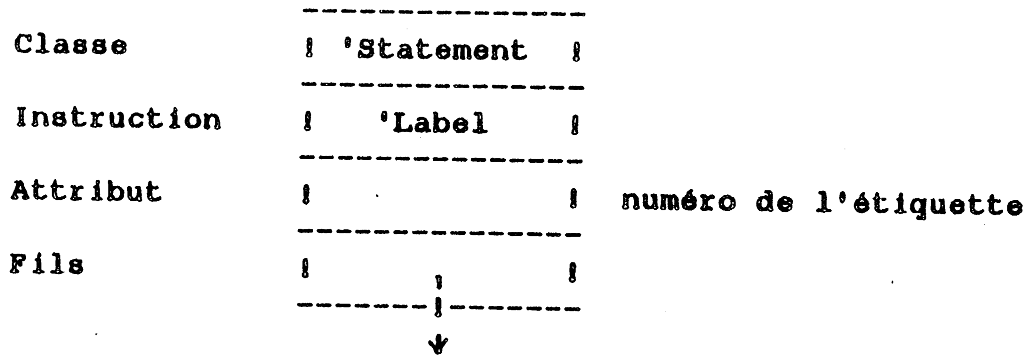
```
                      ---------------
Classe               ! 'Statement  !
                      ---------------
Instructions         !    'Case    !
                      ---------------
Attribut             !    nil      !
                      ---------------
Fils                 !      !      !
                      -------!-------
                             v

      ( ‹expression›   ‹liste de cas› )

‹liste de cas› ::=  ( ‹liste de constantes›  ‹instruction› )
‹liste de constantes› ::= ‹descripteur constante›*
```

```
                      ---------------
Classe               ! 'Statement  !
                      ---------------
Instruction          !   'While    !
                      ---------------
Attribut             !    nil      !
                      ---------------
Fils                 !      !      !
                      -------!-------
                             v

      ( ‹expression›   ‹instruction› )
```

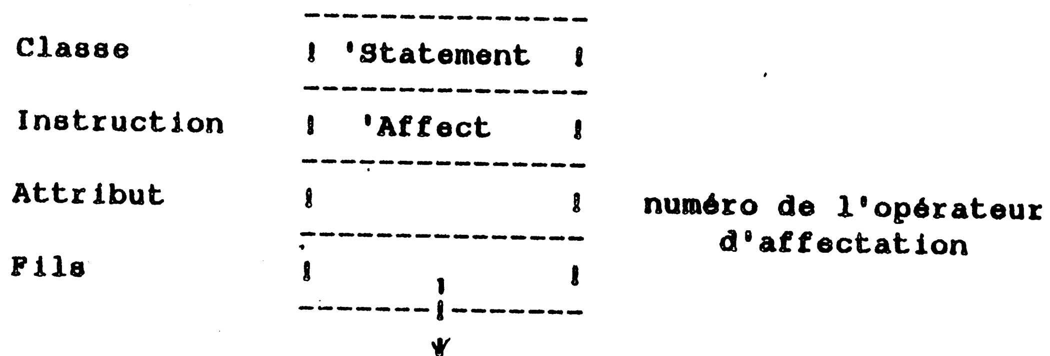
```
                    ---------------
Classe              ! 'Statement  !
                    ---------------
Instruction         !    'For     !
                    ---------------       +1 si "to"
Attribut            !     /-------!-->
                    ---------------       -1 si "downto"
Fils                !      !      !
                    -------!-------
                           v

( <variable> <valeur initiale> <valeur finale> <instruction> )
```

```
                    ---------------
Classe              ! 'Statement  !
                    ---------------
Instruction         !    'With    !
                    ---------------
Attribut            !     nil     !
                    ---------------
Fils                !      !      !
                    -------!-------
                           v

   ( <liste de variables> <instruction> )
```

```
                    ---------------
Classe              ! 'Statement  !
                    ---------------
Instruction         !   'Appel    !
                    ---------------
Attribut            !             !   nom de la procédure
                    ---------------
Fils                !      !      !
                    -------!-------
                           v

(   <param 1> <param 2>    ....    <param n>   )
```



(‹arbre de l'instruction étiquettée›)



(‹destination› ‹source›)

ANNEXE E : EXEMPLES DE DERIVATION DE CODE.

Nous présentons ci-après quelques exemples de sélection d'instructions réalisée par les générateurs GEMME. Nous utilisons pour ce faire les traces produites par ces générateurs, simplifiées pour la clarté de la présentation.

1. Soustraction 68000.

```
Opérandes :
    dw0 = registre dw0
    i   = position mémoire a0+depl_i

Appel noyau :
    Génération( "-i", dw0, i )

Trace :
EvalOpération( "-i", dw0, i, non_affectation )
    EvalConversion( dw0, TypeOpG( "-i" ) )
    EvalConversion( dw0, TypeOpD( "-i" ) )
    EvalConformité( "-i", dw0, i )
        EvalPossibilité( Po(1,"-i"), dw0, i )
            Conf1 := Conformité( dw0, DRGW )
            ==> PaRegDW
            Conf2 := Conformité( i, EAW )
            ==> PaAdrDirect
            EvalRésultat( Conf1, Conf2, gauche )
            ==> dw0
        ==> conforme, coût=(3,6), instruction=SUB
        EvalPossibilité( Po(2,"-i"), dw0, i )
            Conf1 := Conformité( dw0, DRGW )
            ==> PaRegDW
            Conf2 := Conformité( i, AltMemW )
            ==> PaAdrDirect
            EvalRésultat( Conf1, Conf2, droit )
            ==> échec
        ==> échec, description_analyse, instruction=SUB
        EvalPossibilité( Po(3,"-i"), dw0, i )
            Conf1 := Conformité( dw0, ARGW )
            ==> échec
            Conf2 := Conformité( i, EAW )
            ==> échec
            EvalRésultat( Conf1, Conf2, gauche )
            ==> échec
        ==> échec, description_analyse, instruction=SUBA
```

```
EvalPossibilité( Po(4,"-I", dw0, I )
    Conf1 := Conformité( dw0, DARGW )
    ==> PaRegDW
    Conf2 := Conformité( I, CM16 )
    ==> échec
    EvalRésultat( Conf1, Conf2, gauche )
    ==> dw0
==> échec, description_analyse, instruction=SUBI
EvalPossibilité( Po(5,"-I"), dw0, I )
    Conf1 := Conformité( dw0, AltMemW )
    ==> échec
    Conf2 := Conformité( I, CM16 )
    ==> échec
    EvalRésultat( Conf1, Conf2, gauche )
    ==> échec
==> échec, description_analyse, instruction=SUBI
EvalPossibilité( Po(6,"-I"), dw0, I )
    Conf1 := Conformité( dw0, DARGW )
    ==> PaRegDW
    Conf2 := Conformité( I, CM3 )
    ==> échec
    EvalRésultat( Conf1, Conf2, gauche )
    ==> dw0
==> échec, description_analyse, instruction=SUBQ
EvalPossibilité( Po(7,"-I"), dw0, I )
    Conf1 := Conformité( dw0, AltMemW )
    ==> échec
    Conf2 := Conformité( I, CM3 )
    ==> échec
    EvalRésultat( Conf1, Conf2, gauche )
    ==> échec
==> échec, description_analyse, instruction=SUBQ
==> conforme, coût=(3,6), instruction=SUB,
    description_choix
==> coût=(3,6), description_choix
Générer( description_choix )
    Emettre( instructions( description_choix ) )
    SUB d0,depl_i(a0)
```

2. <u>Moins unaire PDP-11.</u>

Opérande :
    l = position mémoire r5+depl_l

Appel noyau : Génération("-ul", l, nil)

Trace :

```
EvalOpération( "-ul", l, nil, non_affectation )
    EvalConversion( l, TypeOpG( "-ul" ) )
    EvalConformité( "-ul", l, nil )
        EvalPossibilité( Po(1,"-ul"), l, nil )
            Conf1 := Conformité( l, DestinationMemW )
            ==> PaIndexSW
            EvalRésultat( Conf1, nil, gauche )
            ==> échec
        ==> échec, description_analyse, Instruction=NEG
        EvalPossibilité( Po(2,"-ul"), l, nil )
            Conf1 := Conformite( l, DestinationRegW )
            ==> échec
            EvalRésultat( Conf1, nil, gauche )
            ==> échec
        ==> échec, description_analyse, Instruction=NEG
    ==> échec, description_échec
    EvalTransformation( "-ul", l, nil, description_échec(1) )
        EvalOpGauche( l, description_échec_opgauche )
            EvalTransform( l, PaIndexSW, contiendra_résultat )
            ==> échec
            EvalTransform( l, PaIndirectSW, contiendra_résultat )
            ==> échec
            EvalTransform( l, PaRegIndirectSW, contiendra_résultat )
            ==> échec
        ==> échec
    ==> échec
    EvalTransformation( "-ul", l, nil, description_échec(2) )
        EvalOpGauche( l, description_échec_opgauche )
            EvalTransform( l, PaRegW, contiendra_résultat )
                EvalAllouerReg( PaRegW )
                ==> r0
                EvalOpération( ":=l", r0, l, affectation )
                ==> conforme, coût=(4,770), Instruction=MOV,
                    description_affectation
            ==> conforme, coût=(4,770), description_choix
        ==> conforme, coût=(4,770), description_choix
    ==> conforme, coût=(4,770), description_choix
==> conforme, coût=(6,1190), Instruction=NEG,
    description_choix
```

```
Générer( description_choix )
    GénérerOpG( description_choix_opgauche )
        Transformer( i, r0 )
            AllouerReg( r0 )
            Générer( description_affectation )
                Emettre( instructions( description_affectation ) )
                MOV depl_i(r5),r0
    Emettre( instructions( description_choix ) )
    NEG r0
```

3. **Multiplication 68000.**

Opérandes :

```
    i = position mémoire a0+depl_i
    j = position mémoire a0+depl_j
```

Appel noyau : Génération("*I", i, j)

Trace :

```
EvalOpération( "*I", i, j, non_affectation )
    EvalConversion( i, TypeOpG( "*I" ) )
    EvalConversion( j, TypeOpD( "*I" ) )
    EvalConformité( "*I", i, j )
        EvalPossibilité( Po(1,"*I"), i, j )
            Conf1 := Conformité( i, DRGW )
            ==> échec
            Conf2 := Conformité( j, EAW )
            ==> PaAdrDirect
            EvalRésultat( Conf1, Conf2, gauche )
            ==> échec
        ==> échec, description_analyse, instruction=MULS
        EvalPossibilité( Po(1,"*I"), j, i ) (* commutativité *)
            Conf1 := Conformité( j, DRGW )
            ==> échec
            Conf2 := Conformité( i, EAW )
            ==> PaAdrDirect
            EvalRésultat( Conf1, Conf2, gauche )
            ==> échec
        ==> échec, description_analyse, instruction=MULS
    ==> échec, description_échec
    EvalTransformation( "*I", i, j, description_échec(1) )
        EvalOpGauche( i, description_échec_opgauche )
            EvalTransform( i, PaRegDW )
                EvalAllouerSurReg( PaRegDW )
                ==> dw0 (dl0)
                EvalOpération( ":=I", dw0, i, affectation )
                ==> conforme, coût=(3,6), instruction=MOVE,
                      description_affectation
            ==> conforme, coût=(3,6), description_choix
        ==> conforme, coût=(3,6), description_choix
        EvalOpDroit( j, description_échec_opdroit )
        ==> conforme, coût=(0,0)
    ==> conforme, coût=(3,6), description_choix
```

```
EvalTransformation( "*i", j, i, description_échec(2) )
    EvalOpGauche( j, description_échec_opgauche )
        EvalTransform( j, PaRegDW )
            EvalAllouerSurReg( PaRegDW )
            ==> dw0 (dl0)
            EvalOpération( ":=i", dw0, j, affectation )
            ==> conforme, coût=(3,6), instruction=MOVE,
                 description_affectation
        ==> conforme, coût=(3,6), description_choix
    ==> conforme, coût=(3,6), description_choix
    EvalOpDroit( i, description_échec_opdroit )
    ==> conforme, coût=(0,0)
  ==> conforme, coût=(3,6), description_choix
==> conforme, coût=(5,41), instruction=MULS,
      description_choix (* à coût égal, on prend la première *)
Générer( description_choix )
    GénérerOpG( description_choix_opgauche )
        Transformer( i, dw0 )
            AllouerReg( dl0 )
            Générer( description_affectation )
                Emettre( instructions( description_affectation ) )
                MOVE  depl_i(a0),dw0
    Emettre( instructions( description_choix ) )
    MULS  dw0,depl_j(a0)
```

ANNEXE F : COMPARAISON DU CODE GENERE.

1. PROGRAMME SOURCE.

```
program conversion( input, output ) ;
var car : char ;
     valcar, valeur : integer ;
begin
   (* ignorer les blancs du début *)
   read( car ) ;
   while car = ' ' do read( car ) ;

   (* boucle de traitement *)
   while car <> '.' do
   begin
      (* traitement d'un entier *)
      valeur := ord( car ) - ord( '0' ) ;
      read( car ) ;
      while ( car <> ' ' ) and ( car <> '.' ) do
      begin
         valeur := valeur * 10 ;
         valcar := ord( car ) - ord( '0' ) ;
         valeur := valeur + valcar ;
         read( car )
      end ;
      write( valeur ) ;

      (* ignorer les blancs entre 2 entiers *)
      while car = ' ' do read( car )
   end
end.
```

2. CODE GENERE POUR LE MOTOROLA 68000.

2.1. Pascal GEMME/68000.

```
et1: pea    _input
     jsr    _readc
     moveb d7,car(a0)
     addql #4,a7
     cmpib car(a0),#' '
     beq    et1
et2: cmpib car(a0),#'.'
     beq    et3
     moveb car(a0),d0
     subib #'0',d0
     extw   d0
     extl   d0
     movel d0,valeur(a0)
et4: pea    _input
     jsr    _readc
     moveb d7,car(a0)
     addql #4,a7
     cmpib car(a0),#' '
     beq    et5
     cmpib car(a0),#'.'
     beq    et5
     pea    10
     movel valeur(a0),-(a7)
     jsr    _multi
     movel d7,valeur(a0)
     addql #8,a7
     moveb car(a0),d0
     subib #'0',d0
     extw   d0
     extl   d0
     movel d0,valcar(a0)
     addl   d0,valeur(a0)
     bra    et4
et5: pea    _output
     movel valeur(a0),-(a7)
     pea    _formati
     jsr    _writei
     addl    12,a7
et6: cmpib car(a0),#' '
     bne    et2
     pea    _input
     jsr    _readc
     moveb d7,car(a0)
     addql #4,a7
     bra    et6
et3: <fin  prog.>
```

Taille programme = 160 octets

2.2. Pascal Siemens (SM90).

```
       movel -input,a0                    jsr    _pimul
       moveb (a0),car(a5)                 addql #8,a7
       pea   _input                       movel d0,valeur(a5)
       jsr   _get                         clrl  d2
       addql #4,a7                        moveb car(a5),d2
et1:   cmpib #' ',car(a5)                 subl  #'0',d2
       bnel  et2                          movel d2,valcar(a5)
       movel _input,a1                    addl  d0,d2
       moveb (a1),car(a5)                 movel d2,valeur(a5)
       pea   _input                       movel _input,a3
       jsr   _get                         moveb (a3),car(a5)
       addql #4,a7                        pea   _input
       bra   et1                          jsr   _get
et2:   cmpib #'.',car(a5)                 addql #4,a7
       beql  et3                          bra   et4
       clrl  d1                    et5:   pea   12
       moveb car(a5),d1                   movel valeur(a5),-(a7)
       subl  #'0',d1                      pea   _output
       movel d1,valeur(a5)                jsr   _pwri
       movel _input,a2                    addl  #12,a7
       moveb (a2),car(a5)          et6:   cmpib #' ',car(a5)
       pea   _input                       bnel  et7
       jsr   _get                         movel _input,a4
       addql #4,a7                        moveb (a4),car(a5)
et4:   cmpib #' ',car(a5)                 pea   _input
       beql  et5                          jsr   _get
       cmpib #'.',car(a5)                 addql #4,a7
       beql  et5                          bra   et6
       pea   10                    et7:   bral  et2
       movel valeur(a5),-(a7)      et3:   <fin prog>
```

Taille programme = 262 octets

2.3. Pascal Berkeley (Micromega).

```
        movel input,-8(a6)
        movel -8(a6),-(a7)
        jsr   READC
        addql #4,a7
        moveb d0,car(a6)
14:     moveb car(a6),d0
        moveq #' ',d1
        extw  d1
        extl  d1
        cmpb  d1,d0
        bnel  15
        movel input,-8(a6)
        movel -8(a6),-(a7)
        jsr   READC
        addql #4,a7
        moveb d0,car(a6)
        bra   14
15:16:moveb car(a6),d0
        moveq #'.',d1
        extw  d1
        extl  d1
        cmpb  d1,d0
        beql  et7
        moveb car(a6),d0
        subl  #'0',d0
        movel d0,valeur(a6)
        movel input,-8(a6)
        movel -8(a6),-(a7)
        jsr   READC
        addql #4,a7
        moveb d0,car(a6)
18:     moveb car(a6),d0
        moveq #' ',d1
        extw  d1
        extl  d1
        cmpb  d1,d0
        beq   1100
        moveq #1,d0
        bra   1101
1100:   clrl  d0
1101:   moveb car(a6),d1
        moveq #'.',d2
        extw  d2
        extl  d2
        cmpb  d2,d1
        beq   1102
```

```
        moveq #1,d1
        bra   1103
1102:   clrl  d1
1103:   andl  d1,d0
        beql  19
        movel #10,-(a7)
        movel valeur(a6),-(a7)
        jsr   lmul
        addql #8,a7
        movel d0,valeur(a6)
        moveb car(a6),d0
        subl  #'0',d0
        movel d0,valcar(a6)
        movel valeur(a6),d0
        addl  valcar(a6),d0
        movel d0,valeur(a6)
        movel input,-8(a6)
        movel -8(a6),-(a7)
        jsr   READC
        addql #4,a7
        moveb d0,car(a6)
        bra   18
110:    ascii "%10D/0"
19:     movel output,-8(a6)
        movel valeur(a6),-(a7)
        movel 110,-(a7)
        movel -4(a6),-(a7)
        jsr   ACTFILE
        addql #4,a7
        movel d0,-(a7)
        jsr   fprintf
        addl  #12,a7
111:    moveb car(a6),d0
        moveq #32,d1
        extw  d1
        extl  d1
        cmpb  d1,d0
        bnel  112
        movel input,-8(a6)
        movel -8(a6),-(a7)
        jsr   READC
        addql #4,a7
        moveb d0,car(a6)
        bra   111
112:    bral  16
17:     <fin prog>
```

Taille programme = 294 octets

3. CODE GENERE POUR LE PDP-11.

3.1. Pascal GEMME/PDP-11

```
et1: mov   _input(r5),-(sp)
     jsr   pc,_readc
     mov   (sp)+,(r5)
     cmpb  (r5),#' '
     beq   et1
et2: cmpb  (r5),#' '
     beq   et3
     movb  (r5),r0
     sub   #' ',r0
     mov   r0,valeur(r5)
et4: mov   _input(r5),-(sp)
     jsr   pc,_readc
     mov   (sp)+,(r5)
     cmpb  (r5),#' '
     beq   et5
     cmpb  (r5),#'.'
     beq   et5
     mov   #10,r0
     mul   valeur(r5),r10
     mov   r1,valeur(r5)
     movb  (r5),r0
     sub   #'0',r0
     mov   r0,valcar(r5)
     add   valcar(r5),valeur(r5)
     br    et4
et5: mov   _output(r5),-(sp)
     mov   valeur(r5),-(sp)
     mov   _formati(r5),-(sp)
     jsr   pc,_writei
     add   #6,sp
et6: cmpb  (r5),#' '
     bne   et2
     mov   _input(r5),-(sp)
     jsr   pc,_readc
     mov   (sp)+,(r5)
     br    et6
et3: <fin prog>
```

Taille programme = 122 octets

3.2. OMSI Pascal-1 (PDP-11)

```
        jsr   pc,_readc
        movb  (sp)+,(r5)
11:     cmpb  (r5),#' '
        beq   12
        jmp   13
12:     jsr   pc,_readc
        movb  (sp)+,(r5)
        jmp   11
13:10:15:
        cmpb  (r5),#'.'
        bne   16
        jmp   17
16:     movb  (r5),r0
        sub   #'0',r0
        mov   r0,valeur(r5)
        jsr   pc,_readc
        movb  (sp)+,(r5)
19:     cmpb  (r5),#' '
        bne   110
        clr   r0
        br    111
110:111:
        cmpb  (r5),#'.'
        bne   112
        clr   r1
        br    113
112:    mov   #1,r1
113:    comb  r1
        bicb  r1,r0
        bne   114
        jmp   115
114:    mov   valeur(r5),-(sp)
        mov   #10,-(sp)
        jsr   pc,_mul
        mov   (sp)+,valeur(r5)
        movb  (r5),r0
        sub   #'0',r0
        mov   r0,valcar(r5)
        add   valcar(r5),valeur(r5)
        jsr   pc,_readc
        movb  (sp)+,(r5)
        jmp   19
115:18:
        mov   valeur(r5),-(sp)
        mov   #7,-(sp)
        jsr   pc,_writei
117:    cmpb  (r5),#' '
        beq   118
        jmp   119
118:    jsr   pc,_readc
        movb  (sp)+,(r5)
        jmp   117
119:116:
        jmp   15
17:14:
        <fin prog>
```

Taille programme = 168 octets

BIBLIOGRAPHIE

BIBLIOGRAPHIE

<Aho 76>
Aho A.V., Johnson S.C.
"Optimal code generation for expression trees"
Journal of the ACM, vol.23, 3 (Juillet 1976), pp.488-501

<Aho 79>
Aho A.V., Ullman J.D.
"Principles of compiler design"
Addison-Wesley Publishing Company
New Jersey, Avril 1979

<Akin 82>
Akin T.A., Leblanc R.J.
"The design and implementation of a code generation tool"
Software-Practice and Experience, vol.12, 11 (Novembre 1982), pp.1027-1041

<Allen 80>
Allen F.E. et al
"The Experimental Compiling System"
IBM Journal of research and development, vol.24, 6 (Novembre 1980), pp.695-715

<Ammann 73>
Ammann U.
"The method of structured programming applied to the development of a compiler"
International Computing Symposium
Davos, Suisse, Septembre 1973, pp.93-99

<Ammann 77>
Ammann U.
"On code generation in a Pascal compiler"
Software, Practice and Experience, vol.7, 3 (Juin 1977), pp.391-423

<Atkinson 82>
Atkinson L.V.
"Optimizing two-state case statements in Pascal"
Software-Practice and Experience, vol.12, 6 (Juin 1982), pp.571-581

<Barbacci 78>
Barbacci M.R., Barnes G., Cattell R., Siewiorek D.
"ISPS reference manual"
Department of Computer Science
Carnegie-Mellon University, Mars 1978

<Bird 82>
Bird P.L.
"An implementation of a code generator specification language for table driven code generators"
Sigplan 82 Symposium on Compiler Construction
Boston, Juin 1982, pp.44-55

<Bonkowski 79>
Bonkowski G.B., Gentleman W.M., Malcolm M.A.
"Porting the Zed compiler"
SIGPLAN Symposium on Compiler Construction
Denver, Août 1979, pp.92-97

<Boullier 80>
Boullier P.
"Génération automatique d'analyseurs syntaxiques avec rattrapage d'erreurs"
Journées Francophones "Production assistée de logiciel"
Genève, Janvier 1980

<Briat 81>
Briat J. et al
"Adèle : Un atelier de développement de logiciel"
AFCET Informatique
Paris, Novembre 1981, pp.189-199

<Bruno 75>
Bruno J.L., Sethi R.
"The generation of optimal code for stack machines"
Journal of the ACM, vol.22, 3 (Juillet 1975), pp.382-396

<Carter 77>
Carter J.L.
"A case study of a new code generation technique for compilers"
Communications of the ACM, vol.20, 12 (Décembre 1977), pp.914-920

<Cassagne 82>
Cassagne B., Hochain J.C., Santana M.
"Génération de code multicible"
Journées d'études CONCERTO
Perros Guirec, Décembre 1982

<Cattell 77>
Cattell R.G.
"A survey and critique of some models of code generation"
Computer Science Department
Carnegie-Mellon University, Novembre 1977

<Cattell 78>
Cattell R.G.
"Formalization and automatic derivation of code generators"
T.R. 78-115, PhD thesis, Computer Science Department
Carnegie-Mellon University, Avril 1978

<Cattell 79a>
Cattell R.G., Newcomer J.M., Leverett B.W.
"Code generation in a machine-independent compiler"
Sigplan Symposium on Compiler Construction
Denver, Août 1979, pp.65-75

<Cattell 79b>
Cattell R.G.
"Code generation and machine descriptions"
T.R. CSL-79-8, Xerox Palo Alto Research Center
Palo Alto, Octobre 1979

<Cattell 80>
Cattell R.G.
"Automatic derivation of code generators from machine descriptions"
ACM Transactions on Programming Languages and Systems, vol.2, 2 (Août 1980), pp.173-190

<Cheval 82>
Cheval J.L., Estublier J., Ghoul S., Krakowiak S.
"Modularité et composition des programmes dans l'atelier de logiciel Adèle"
Colloque Génie Logiciel (AFCET)
Paris, Juin 1982, pp.183-197

<Coleman 74>
Coleman S.S., Poole P.C., Waite W.M.
"The mobile programming system JANUS"
Software, Practice and Experience, vol.4, 1 (Janvier 1974), pp.5-23

<Crawford 82>
Crawford J.
"Engineering a production code generator"
Sigplan 82 Symposium on Compiler Construction
Boston, Juin 1982, pp.205-215

<Cunin 76>
Cunin P.Y., Simonet M., Voiron J.
"Méthodologie d'écriture de compilateurs"
Thèse 3ème Cycle, Faculté d'Informatique
Université de Grenoble I, Avril 1976

<Davidson 80>
Davidson J.W., Fraser C.W.
"The design and application of a retargetable peephole optimizer"
ACM TOPLS, vol.2, 2 (Avril 1980), pp.191-202

<Davidson 81>
Davidson J.W.
"Simplifying code generation through peephole optimization"
T.R. 81-19, Department of Computer Science
University of Arizona, Décembre 1981

<Davidson 82>
Davidson J.W., Fraser C.W.
"Eliminating redundant object code"
9th Annual Symposium on Principles of Programming Languages
Albuquerque, Janvier 1982, pp.128-132

<Dec 75>
"PDP-11 Processor Handbook"
Digital Equipment Corp.
Maynard, Mass., 1975

<Dec 77>
"VAX 11/780 Architecture Handbook"
Digital Equipment Corporation
Maynard, 1977

<Dietmeyer 74>
Dietmeyer D.L.
"Introducing DDL"
IEEE Computer, vol.7, 12 (Décembre 1974), pp.34-38

<Donegan 73>
Donegan M.K.
"An approach to the automatic generation of code generators"
PhD thesis, Computer Science and Engineering Department
Rice University, 1973

<Donegan 79>
Donegan M.K., Noonan R.E., Feyock S.
"A code generator generator language"
SIGPLAN Symposium on Compiler Construction
Denver, Août 1979, pp.58-64

<Elson 70>
Elson M., Rake S.T.
"Code generation technique for large-language compilers"
IBM Systems Journal, vol.9, 3 (Décembre 1970), pp.166-188

<Estublier 83>
Estublier J., Krakowiak S., Mossière J., Rouzaud Y.
"Design principles of the Adele programming environment"
7th International Computing Symposium
Nuremberg, Mars 1983, pp.?

<Finger 82>
Finger U. et al
"Description générale de la SM90"
Note technique NT/PAA/OGE/SML/703
CNET Lannion, Mai 1982

<Fortier 75>
Fortier R.
"Etude et définition du langage intermédiaire et d'une machine formelle multiprocesseurs orientée vers l'exécution du langage Pascal"
Thèse 3ème Cycle, Faculté d'Informatique
Université de Grenoble I, Octobre 1975

<Fraser 77>
Fraser C.W.
"Automatic generation of code generators"
PhD thesis, Computer Sciences Department
Yale University, 1977

<Fraser 79>
Fraser C.W.
"A compact machine independant peephole optimizer"
6th Annual Symposium on Principles of Programming Languages
San Antonio, Janvier 1979, pp.1-6

<Ganapathi 81>
Ganapathi M., Fischer C.N.
"A review of automatic code generation techniques"
T.R. 407, Computer Sciences Department
University of Wisconsin-Madison, Janvier 1981

<Ganapathi 82a>
Ganapathi M., Fischer C.N.
"Description driven code generation with attribute grammars"
9th Annual Symposium on Principles of Programming Languages
Alburquerque, Janvier 1982, pp.108-119

<Ganapathi 82b>
Ganapathi M., Fischer C.N., Hennessy J.L.
"Retargetable compiler code generation"
Computing Surveys, vol.14, 4 (Décembre 1982), pp.573-592

<Ganzinger 77>
Ganzinger H., Ripken K., Wilhelm R.
"Automatic generation of optimizing multipass compilers"
Information Processing 77
Toronto, Août 1977, pp.535-540

<Glanville 78>
Glanville R.S., Graham S.L.
"A new method for compiler code generation"
5th Annual ACM Symposium on Principles of Programming Languages"
Tucson, Janvier 1978, pp.231-240

<Graham 80>
Graham S.L.
"Table-driven code generation"
IEEE Computer, vol.13, 8 (Août 1980), pp.25-34

<Graham 82>
Graham S.L., Henry R.R., Schulman R.A.
"An experiment in table driven code generation"
Sigplan 82 Symposium on Compiler Construction
Boston, Juin 1982, pp.32-43

<Gries 65>
Gries D., Paul M., Wiehle H.R.
"Some techniques used in the ALCOR ILLINOIS 7090"
Communications of the ACM, vol.8, 8 (Août 1965), pp.496-500

<Gries 71>
Gries D.
"Compiler construction for digital computers"
John Wiley & sons, Inc
New York, 1971

<Griffiths 69>
Griffiths M., Peltier M.
"A macro-generable language for the 360 computers"
Computer bulletin, vol.13, 11 (Novembre 1969)

<Haddon 78>
Haddon B.K., Waite W.M.
"Experience with the universal intermediate language JANUS"
Software, Practice and Experience, vol.8, 5 (Octobre 1978), pp.601-616

<Hennessy 82>
Hennessy J.L., Mendelsohn N.
"Compilation of the Pascal case statement"
Software-Practice and Experience, vol.12, 9 (Septembre 1982), pp.879-882

<Henry 81>
Henry R.R.
"The code generator's generator work station : experiments with the Graham-Glanville machine independent algorithms for code generation"
UCB/ERL M81/47, Master thesis, Electronics Research Laboratory
University of California at Berkeley, Juin 1981

<Herrmann 82>
Herrmann M., Raymond J.
"Le poste de travail Adèle"
In "Adèle : Un atelier de développement de logiciel"
Laboratoire IMAG, R.R. 299
Grenoble, 1982

<Hochain 80>
Hochain J.C.
"Compilateur CPL1 version 4.2"
Rapport interne CAP-SOGETI
Montrouge, Juin 1980

<Holvoet 82>
Holvoet Y., Mullet B.
"Un noyau de générateur universel de code paramétré par une description de machine"
Rapport DEA
ENSIMAG, Grenoble, Juin 1982

<Intel 78>
"MCS-86 User's manual"
Intel Corporation
Santa Clara, 1978

<Johnson 75>
Johnson S.C.
"Yacc : Yet another compiler-compiler"
T.R. 32, Computing Science
Bell Laboratories, Murray Hill, 1975

<Johnson 78>
Johnson S.C.
"A portable compiler : theory and practice"
5th Annual Symposium on Principles of Programming Languages
Tucson, Janvier 1978, pp.97-104

<Knuth 71>
Knuth D.E.
"The art of computer programming"
Addison-Wesley
Reading, Massachusetts, 1971

<Kozlak 81>
Kozlak R.H.
"Machine-independent code generation"
T.R. CSRG-125, Computer Systems Research Group
University of Toronto, Janvier 1981

<Krumme 82>
Krumme D.W., Ackley D.H.
"A practical method for code generation based on exhaustive search"
Sigplan 82 Symposium on Compiler Construction
Boston, Juin 1982, pp.185-196

<Lamb 81>
Lamb D.A.
"Construction of a peephole optimizer"
Software-Practice & Experience, vol.11, 6 (Juin 1981), pp.639-647

<Landwehr 82>
Landwehr R., Jansohn H.S., Goos G.
"Experience with an automatic code generator generator"
Sigplan 82 Symposium on Compiler Construction
Boston, Juin 1982, pp.56-66

<Lebarbier 79>
Lebarbier D.
"Etude et réalisation d'un producteur de code paramétrable par les descriptions des langages source et cible"
Thèse 3ème Cycle, Faculté de Mathématiques et Informatique
Université de Rennes, Octobre 1979

<Leverett 80>
Leverett B.W. et al
"An overview of the Production-Quality Compiler-Compiler project"
IEEE Computer, vol.13, 8 (Août 1980), pp.38-49

<Leverett 81>
Leverett B.W.
"Register allocation in optimizing compilers"
T.R. 81-103, Department of Computer Science
Carnegie-Mellon University, Février 1981

<McKeeman 65>
McKeeman W.M.
"Peephole optimization"
Communications of the ACM, vol.8, 7 (Juillet 1965), pp.443-444

<Miller 71>
Miller P.L.
"Automatic creation of a code generator from a machine description"
T.R. 85, Projet MAC
Massachusetts Insitute of Technology, 1971

<Mossière 82>
Mossière J., Raymond J., Rouzaud Y.
"Représentation interne et manipulation de programmes dans l'atelier de logiciel Adèle"
Colloque Génie Logiciel (AFCET)
Paris, Juin 1982, pp.137-150

<Motorola 80>
"M680000 Processor Handbook"
Motorola Company
California, 1980

<Naffah 79>
Naffah N.
"Exemple d'un poste de travail burotique à interface universelle"
T.R. MEV.2.503, Projet pilote KAYAK
INRIA, Juin 1979

<Nakata 67>
Nakata I.
"On compiling algorithms for arithmetic expressions"
Communications of the ACM, vol.10, 8 (Août 1967), pp.492-494

<Newcomer 75>
Newcomer J.M.
"Machine independent generation of optimal local code"
PhD thesis, Department of Computer Science
Carnegie-Mellon University, 1975

<Nori 75>
Nori K.V., Ammann U., Jensen K., Nageli H.
"The Pascal(P) compiler implementation notes"
Institut fur Informatik, Eidgenossische Technische Hochschule
Zurich, 1975

<Perkins 79>
Perkins D.R., Sites R.L.
"Machine independent Pascal code optimization"
Sigplan Symposium on Compiler Construction
Denver, Août 1979, pp.201-207

<Piloty 80>
Piloty R., Barbacci M., Borrione D., Dietmeyer D., Hill F., Skelly P.
"CONLAN - A formal construction method for hardware description languages : basic principles, language derivation and language application"
AFIPS
Anaheim, California, Mai 1980, pp.209-236

<Richards 77>
Richards M.
"The portability of the BCPL compiler"
Software, Practice and Experience, vol.1 (1977), pp.135-146

<Ripken 78>
Ripken K.
"A formal method for describing machine code generation with local optimizations"
Le point sur la compilation
Cours IRIA, pp. 247-306, Janvier 1978

<Rudmik 79>
Rudmik A., Lee E.S.,
"Compiler design for efficient code generation and program optimization"
SIGPLAN Symposium on Compiler construction
Denver, Août 1979, pp.127-139

<Sale 81>
Sale A.
"The implementation of case statements in Pascal"
Software-Practice and Experience, vol.11, 9 (Septembre 1981), pp.929-942

<Santana 83>
Santana M.
"Algorithmes de compilation des opérations ensemblistes"
Rapport interne, Projet Adèle
Laboratoire IMAG, Octobre 1983

<Sems 78>
"Solar 16"
Société Européenne de Mini-informatique et de Systèmes
Grenoble, 1978

<Sethi 70>
Sethi R., Ullman J.D.
"The generation of optimal code for arithmetic expressions"
Journal of the ACM, vol.17, 4 (Octobre 1970), pp.715-728

<Shimasaki 80>
Shimasaki M., Fukaya S., Ikeda K., Kiyono T.
"An analysis of Pascal programs in compiler writing"
Software-Practice and Experience, vol.10, 2 (Février 1980), pp.149-157

<Sibley 61>
Sibley R.A.
"The SLANG system"
Communications of the ACM, vol.4, 1 (Janvier 1961), pp.75-84

<Snyder 75>
Snyder A.
"A portable compiler for the language C"
T.R. 149, Projet MAC
Massachusetts Institute of Technology, 1975

<Steel 61>
Steel T.B.
"A first version of UNCOL"
Western Joint Computer Conference (AFIPS)
Los Angeles, Mai 1961, pp.371-377

<Strong 58>
Strong J. et al
"The problem of programming communication with changing machines : a proposed solution"
Communications of the ACM, vol.1, 8 (Août 1958), pp.12-18

<Szymanski 78>
Szymanski T.G.
"Assembling code for machines with span-dependent instructions"
Communications of the ACM, vol.21, 4 (Avril 1978), pp.300-308

<Tanenbaum 82>
Tanenbaum A.S., Stavaren H.van, Stevenson J.W.
"Using peephole optimization on intermediate code"
ACM TOPLS, vol.4, 1 (Janvier 1982), pp.21-36

<Tanenbaum 83>
Tanenbaum A.S., van Staveren H., Keizer E.G., Stevenson J.W.
"A practical tool kit for making portable compilers"
Communications of the ACM, vol.26, 9 (Septembre 1983), pp.654-662

<Thomson 82>
"Micromega-32. Guide utilisateur"
Thomson-CSF, Département Informatique de Bureau.
Paris, 1982

<Weingart 73>
Weingart S.W.
"An efficient and systematic method of compiler code generation"
PhD thesis, Computer Sciences Department
Yale University, 1973

<Wilcox 71>
Wilcox T.R.
"Generating machine code for high level programming languages"
T.R. 71-103, Department of Computer Science
Cornell University, Septembre 1971

<Wulf 75>
Wulf W.A., Johnson R.K., Weinstock Ch.B., Hobbs S.O., Geschke Ch.M.
"The design of an optimizing compiler"
American Elsevier Publishing Co.
New York, 1975

<Wulf 79>
Wulf W. et al
"An overview of the Production Quality Compiler Compiler project"
T.R. CS-79-105, Department of Computer Science
Carnegie-Mellon University, Février 1979

<Wulf 80>
Wulf W.A.
"PQCC : A machine-relative compiler technology"
T.R. 80-144, Department of Computer Science
Carnegie-Mellon University, Septembre 1980

<Young 74>
Young R.
"The coder : a program module for code generation in high level language compilers"
M.S. thesis, Computer Science Department
University of Illinois, 1974

DERNIERE PAGE D'UNE THESE

3È CYCLE, DOCTEUR INGÉNIEUR OU UNIVERSITÉ

Vu les dispositions de l'arrêté du 16 avril 1974,

Vu les rapports de M. ... Krakowiak ...

M. ....

M. ... SANTANA ORMENO ... est autorisé

à présenter une thèse en vue de l'obtention du grade de DOCTEUR ... 3e cycle

... Informatique ...

Grenoble, le 25 NOV. 1983

Le Président de l'Université Scientifique et Médicale

M. TANCHE

M. Tanche