Titre: Title:	Méthode de construction de traducteurs de pseudocode schématique
Auteur: Author:	Alain Grenier
Date:	1989
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Grenier, A. (1989). Méthode de construction de traducteurs de pseudocode schématique [Mémoire de maîtrise, Polytechnique Montréal]. PolyPublie. https://publications.polymtl.ca/58233/

UNIVERSITÉ DE MONTRÉAL

MÉTHODE DE CONSTRUCTION DE TRADUCTEURS DE PSEUDOCODE SCHÉMATIQUE.

par

Alain GRENIER

DÉPARTEMENT DE GÉNIE ÉLECTRIQUE

ÉCOLE POLYTECHNIQUE

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION

DU GRADE DE MAÎTRE ÈS SCIENCES APPLIQUÉES (M.Sc.A.)

février 1989

ISBN 0-315-52724-2

Canada

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE

Ce mémoire intitulé:

MÉTHODE DE CONSTRUCTION DE TRADUCTEURS DE PSEUDOCODE SCHÉMATIQUE.

présenté par: Alain Grenier

en vue de l'obtention du grade de: Maître es Sciences Appliquées (M.Sc.A.)

a été dûment accepté par le jury d'examen constitué de:

M. H. H. Hoang	Ph.D., président
M. P. N. Robillard	Ph.D, directeur
M. C. Christen	Ph.D., membre

SOMMAIRE

Le pseudocode schématique établit des normes de programmation. Il aide à la réalisation des programmes en utilisant des formes structurées et une approche systématique à la documentation. Les formes structurées sont schématiques, indépendantes du langage de programmation et peuvent être traduites en tout langage orienté procédure.

Le but de notre recherche consiste à établir une méthode de réalisation de traducteurs de pseudocode schématique applicable à tout langage orienté procédure. Trois approches sont considérées:

- traduction des règles de production de la grammaire du pseudocode schématique, aussi appelée traduction directe;
- traduction des structures complètes du pseudocode schématique, aussi appelée traduction complète;
- traduction orientée sur les structures de base du langage cible, aussi appelée traduction ciblée.

Dans le cadre de ce projet de recherche, nous avons conçu et réalisé des traducteurs de pseudocode schématique pour les langages de programmation C, Ada et assembleur 6502. Pour chaque langage, trois traducteurs (direct, complet et ciblé) ont été réalisés. La méthode de construction de traducteurs est partiellement automatisée et repose sur les techniques utilisées dans la réalisation des compilateurs.

ABSTRACT

Schematic pseudocode is more than a way of representing software; it's a standard. Because it represents the structure of the program with a schematic point of view, it is independent of the target language. It also includes a top-down documentation mechanism. The schematic representation can be translated into all procedural languages.

The purpose of our study is to establish a method which allows for the construction of schematic pseudocode translators into all procedural languages. Three approaches are considered:

- direct translation: straight forward translation of the production rules of the schematic pseudocode grammar;
- complete translation: a translation based on complete schematic structures;
- target translation: a translation based on the structure-statements of the target language.

As a result of our research, we have designed and implemented three schematic pseudocode translators (direct, complete and target) for the following languages: C, Ada and 6502 assembly language. The method for constructing translators is based on techniques and tools used in compiler-construction.

REMERCIEMENTS

Je tiens à remercier Thérèse, ma famille et mes amis pour leur support moral, leurs encouragements et leur patience.

Mon directeur de mémoire, M. Pierre N. Robillard, m'a été d'un grand secours tout au long de mes travaux, je le remercie.

Je remercie aussi mes confrères du laboratoire de recherche en génie logiciel et tout particulièrement Jean-Bernard, André, Daniel et Mario pour leurs commentaires et leur apport à l'évolution de la traduction du pseudocode schématique; sans eux il m'aurait été très difficile d'accomplir un tel travail. Schemacode International et le laboratoire de recherche en génie logiciel ont mis à ma disposition des ressources qui m'ont permis de réaliser mes recherches, je les en remercie. Finalement, je tiens à remercier Marie-Hélène pour la correction de ce mémoire.

*

* *

TABLE DES MATIÈRES

LISTE DES TABLEAUX

CHAPITRE 7 : EXPÉRIMENTATIONS

LISTE DES FIGURES

CHAPITRE 6 : RÉALISATION DES TRADUCTEURS

CONTENU DE L'ANNEXE

INTRODUCTION

La suprême récompense du travail n'est pas ce qu'il vous permet de gagner, mais ce qu'il vous permet de devenir.

John RUSKIN

Les outils logiciels ont connu une évolution toute récente. De nos jours, la gamme d'outils d'aide à la programmation est très étendue: des outils d'aide à la conception aux outils d'analyses statiques et dynamiques de programme. Au début de l'ère informatique, peu d'outils logiciels étaient nécessaires au développement des programmes. Avec l'apparition des langages de haut niveau, on vit croître les compilateurs; ancêtres des outils logiciels. Un des premiers compilateurs serait celui de Laning et Zierler pour le calcul mathématique en 1954 (Sammet [1983]).

En ce temps, les programmes informatiques étaient de petite taille et souvent écrits par un seul individu. Au fur et à mesure que les besoins informatiques prenaient de l'importance, le matériel devenant plus puissant, on se mit à faire des logiciels de grande taille faisant intervenir des équipes complètes de programmeurs. L'absence de démarche systématique faisait en sorte qu'une grande part artistique entrait dans le processus de développement entraînant des problèmes de personnel, de coûts de développement et de maintenance. Des logiciels qui ne réalisent pas les spécifications; des logiciels qui ne fonctionnent pas; des logiciels difficiles à entretenir, livrés dans des délais beaucoup plus longs que prévus sont des problèmes qui émergent de partout. Afin de contrer ce fléau, appelé la crise du logiciel, une nouvelle science naquit: le génie logiciel.

Le génie logiciel tente depuis 1968, moment où il a été introduit à la conférence de l'OTAN, d'apporter une approche organisée au développement du logiciel par l'établissement de normes. Cette nouvelle branche du génie concentre ses efforts à trouver des moyens pour diminuer la complexité et augmenter la fiabilité des programmes afin de rendre la maintenance moins coûteuse; principale préoccupation contemporaine. Les principaux apports du génie logiciel ont été, jusqu'à présent, la notion de programmation structurée et surtout la reconnaissance du cycle de vie du logiciel (Ramamoorthy et Siyan [1983]).

Le cycle de vie conventionnel du logiciel est constitué de six grandes étapes: la définition, la spécification, la conception, la réalisation, la vérification et la maintenance. Une bonne concentration de méthodes et d'outils logiciels est présente à chaque étape du cycle de vie. L'étape de réalisation, et plus précisément la représentation d'algorithme est un domaine qui évolue constamment. Un des premiers modes de représentation utilisé dans ce domaine fut le «Flow Chart», automatisé par plusieurs outils logiciels et encore utilisé aujourd'hui. Ce mode de représentation comporte une série de lacunes constatées à l'usage. De nouveaux modes de représentation d'algorithme tels le pseudocode ou mieux encore le pseudocode schématique ont alors pris naissance. Ces nouveaux modes de représentation intègrent de récents concepts, fruits du génie logiciel, tels l'approche descendante, l'utilisation de formes structurées ainsi que des mécanismes de documentation qui facilitent la maintenance.

Le mode de représentation d'algorithme qui nous concerne est celui du pseudocode schématique, avènement majeur dans le domaine. Le pseudocode schématique est une technique de représentation des structures d'un programme. C'est aussi un ensemble de règles à suivre lors de la réalisation d'un programme assurant une meilleure documentation par le mécanisme reconnu d'abstraction successive. Ce mode de représentation et support de réalisation contribue aussi à diminuer la complexité des programmes en soustrayant les détails non nécessaires à la prise de décision. Le pseudocode schématique peut être vu comme une couche superposée à un langage de programmation qui formalise l'usage des structures de programmation. Ces structures schématiques peuvent être ensuite traduites en langage informatique (Robillard [1985a]).

Les utilisateurs d'outils logiciels qui automatisent le pseudocode schématique se soucient de la forme de ce code produit. En tenant compte de cette préoccupation, la traduction du pseudocode schématique revêt une importance capitale pour l'acceptation de ce mode de représentation comme standard de réalisation de programmes.

Notre projet de recherche consiste à définir une méthode de construction de traducteurs de pseudocode schématique en langage orienté procédure. Ceci, dans le but de formaliser la phase de traduction des outils logiciels qui automatisent le pseudocode schématique. Cela nous permet d'obtenir un meilleur contrôle sur la forme, donc potentiellement, un meilleur contrôle de la performance.

Le mémoire est organisé de la façon suivante:

Au chapitre I on présente le pseudocode schématique tel que défini dans la littérature la plus récente.

Le chapitre II présente la définition de la traduction du pseudocode schématique, la description des différents genres de traduction et les critères de comparaison applicables aux traducteurs de pseudocode schématique.

Le chapitre III résume la théorie des langages informatiques et présente certaines méthodes pour les analyser.

Au chapitre IV on décrit la méthode proposée pour la construction d'un traducteur de pseudocode schématique.

Le chapitre V décrit les outils LEX, YACC et PREP disponibles sur UNIX et introduit l'usage des types abstraits de données de haut niveau.

Au chapitre VI on présente l'utilisation de la méthode de construction de traducteurs à travers la construction de neuf traducteurs de pseudocode schématique.

Le chapitre VII présente les performances des traducteurs construits selon la méthode proposée.

I. LE PSEUDOCODE SCHÉMATIQUE

Nous présentons, dans ce premier chapitre, le pseudocode schématique par son histoire. La définition du pseudocode schématique et la description de ses structures de base apportent l'information nécessaire à une bonne compréhension de nos travaux.

A. Historique

Le pseudocode schématique est né à la fin de 1979 du besoin de voir les structures d'un programme indépendamment de la syntaxe du langage utilisé (Robillard et Thalmann [1980]). Pour ce faire, trois structures schématiques sont mises à la disposition des utilisateurs pour représenter un algorithme: la structure séquentielle représentée par un trait vertical où l'on peut aligner des énoncés en langage cible; la structure conditionnelle où les branches sont montées les unes sur les autres à l'horizontale et la structure répétitive représentée par deux traits verticaux en retrait.

Fig. 1.1 Pseudocode schématique originel - Structure séquentielle

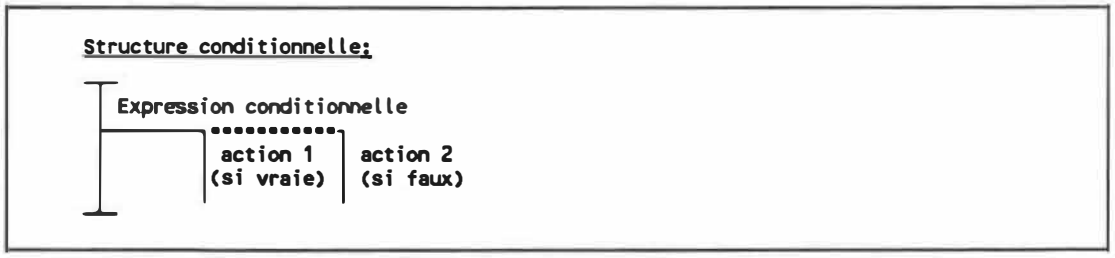


Fig. 1.2 Pseudocode schématique originel - Structure conditionnelle

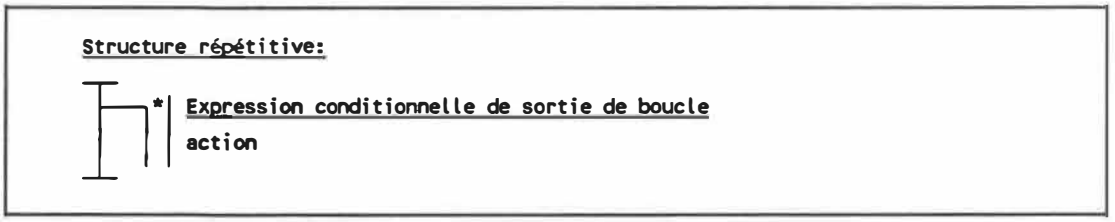


Fig. 1.3 Pseudocode schématique originel - Structure répétitive

L'interprétation des schémas des figures 1.1, 1.2 et 1.3 se fait comme suit:

- Pour la structure séquentielle, les actions 1, 2 et 3 sont exécutées les unes à la suite des autres.
- Pour la structure conditionnelle, si l'expression conditionnelle, exprimée selon la syntaxe d'un langage informatique, est vraie alors l'action 1 est exécutée sinon l'action 2 est exécutée.
- Pour la structure répétitive, tant que l'expression conditionnelle de sortie est fausse l'action est exécutée, on peut dire aussi que la boucle se termine lorsque la condition de sortie est vraie.

Au début de 1981 deux professeurs du département de Génie Électrique de l'École Polytechnique de Montréal mettaient au point un prototype qui automatise le concept du pseudocode schématique (Robillard et Plamondon [1981]). À l'état embryonnaire, le projet s'appelait «Computer Assisted Software Design (CASD) System» et par la suite, "SCHEMACODE". Ce prototype était équipé d'un traducteur en langage FORTRAN selon la norme ANSI FORTRAN 66.

Le pseudocode schématique automatisé n'était pas tout à fait le même que celui défini dans la première grammaire présentée dans Robillard et Thalmann [1979]. On y avait ajouté un mécanisme de raffinements successifs pour permettre l'imbrication des structures. Ce mécanisme automatisé sous la forme de commentaires opérationnels s'est avéré l'instrument tout indiqué pour conférer au pseudocode schématique la propriété d'intégrer l'approche descendante. La grammaire du pseudocode schématique a été modifiée pour être en accord avec le prototype. La représentation d'un algorithme telle que définie par la première grammaire devait être intégrée (représentée dans un seul plan). Grâce au commentaire opérationnel, un algorithme pouvait être représenté par raffinements; processus plus facile à automatiser.

La représentation des structures conditionnelles telle qu'automatisée par ce premier prototype était très contraignante. La limite du nombre de branches conditionnelles consécutives était de 6 et le texte des expressions conditionnelles ne devait pas être trop long. Ces limites étaient nécessaires à cause de la représentation horizontale des branches des conditionnelles.

En 1983, une nouvelle version de SCHEMACODE faisait son apparition et s'identifiait comme un outil logiciel d'aide à la conception et à la maintenance des programmes. La nouvelle version de l'outil avait des traducteurs pour les langages Pascal, FORTRAN 66, FORTRAN 77 et l'assembleur du PDP11 (Robillard et Tan [1983]).

L'outil se présentait avec une nouvelle grammaire du pseudocode schématique qui offrait non plus les structures conditionnelles à l'horizontale mais plutôt à la verticale. Ce changement permettait d'avoir théoriquement un nombre infini de branches conditionnelles, d'écrire des expressions conditionnelles sans contrainte et d'avoir un grand nombre d'énoncés par branche.

Les réalisations entourant l'outil continuèrent, d'autres traducteurs s'ajoutèrent, et en 1984 on intégra un module permettant la représentation intégrée des algorithmes (Robillard [1984]) pour la première version commerciale. En 1985, pour succéder aux versions de l'outil pour ordinateur IBM 360/70 sous TSO puis pour IBM 4341 sous MUSIC et puis pour VAX 750 sous VMS, une version pour micro ordinateur de type IBM PC fit son apparition. Avec celle-ci une modification mineure à la grammaire: la condition de sortie généralisée qui remplaça la condition de sortie avec raffinement et la condition de sortie sans énoncé (Robillard, P.N. [1985b]).

Aujourd'hui l'outil logiciel SCHEMACODE est disponible pour les ordinateurs VAX sous VMS et les ordinateurs de la famille IBM PC sous PC-DOS ou MS-DOS. Les langages supportés sont le FORTRAN, le Pascal, le C, le COBOL, le BASIC et le dBASE III (Schemacode [1987]).

B. Définition

Le pseudocode schématique tire ses racines du pseudocode conventionnel; ce dernier, contrairement au pseudocode schématique, n'a pas de règle de construction syntaxique et sémantique. Ce formalisme donne l'avantage au pseudocode schématique d'être beaucoup moins sensible à la démarche personnelle que chacun apporte dans sa façon de faire des programmes (problème connu sous le nom «Variation In Practice»).

Le pseudocode schématique est un concept de développement de programme informatique. Ce concept met en relief, par des schémas, les structures d'un programme. Trois structures de base servent à représenter les programmes. La structure séquentielle, la structure conditionnelle et la structure répétitive représentent l'algorithme d'un programme de façon indépendante du langage de programmation.

Le concept intègre aussi un mécanisme de raffinements successifs. La tâche à réaliser est divisée en plusieurs sous-tâches, les sous-tâches sont redivisées et ainsi de suite. On obtient un programme complet, construit selon un schème de décomposition fonctionnelle arborescente. Chaque niveau d'abstraction est résumé par un commentaire opérationnel. La documentation se construit au fur et à mesure, sans effort supplémentaire.

Un commentaire opérationnel résume un raffinement. Un raffinement est une division de l'algorithme indépendamment de son implantation. Le commentaire opérationnel n'est pas fonction du langage cible, il en est d'autant plus pertinent.

Ainsi, même si plusieurs programmeurs ont à modifier les détails d'implantation d'un raffinement, la documentation ne se détériore pas, le texte du commentaire opérationnel est toujours aussi approprié (Robillard, Trouvé, Grenier et Beaucage [1988]).

Un algorithme en pseudocode schématique est défini à l'intérieur d'un raffinement de départ, habituellement nommé raffinement #0. Un raffinement débute par un schéma en forme de "T", il est suivi d'une structure séquentielle qui se poursuit jusqu'à la fin et se termine par un schéma de la forme d'un "T" renversé. Sur cette structure séquentielle on peut disposer des structures conditionnelles et répétitives. Un seul niveau de structures conditionnelles ou répétitives peut être superposé à la séquentielle. Cette spécification oblige le programmeur à créer des raffinements si l'algorithme commande l'imbrication de structures conditionnelles ou répétitives.

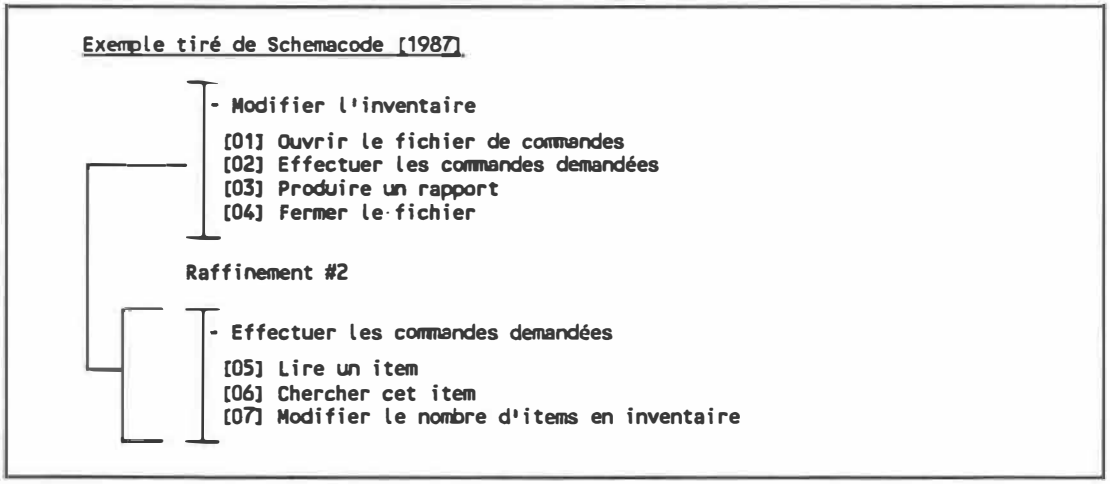


Fig. 1.4 Le commentaire opérationnel et son raffinement

C. Les structures de base

1. La structure séquentielle

La structure séquentielle est représentée par une ligne verticale peut contenir trois types d'énoncés:

- le commentaire narratif, commentaire local à un groupe d'instructions;
- l'énoncé en langage cible effectuant une instruction d'affectation, d'entrée/sortie, d'appel à un sous-programme etc.;
- le commentaire opérationnel qui décrit un raffinement en langage naturel.

Les trois types d'énoncés ont des représentations distinctes. Le texte du commentaire narratif, écrit en langage naturel, est précédé d'un tiret. Le texte du commentaire opérationnel est précédé d'un numéro unique identifiant son raffinement. L'énoncé en langage cible suit la syntaxe du langage de programmation et occupe une ligne complète.

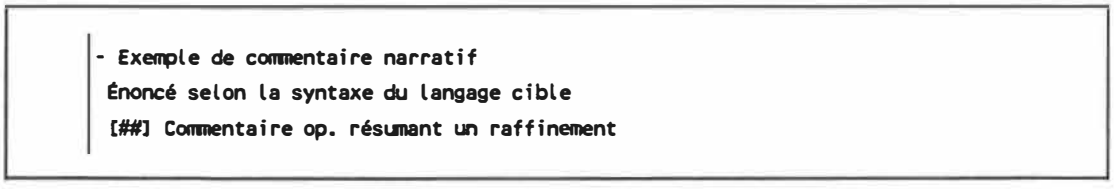


Fig. 1.5 La structure séquentielle

2. La structure conditionnelle

La stucture conditionnelle est formée, au minimum, d'une branche "SI". Un nombre variable de branches "SINON SI" suivent le "SI" et une branche "SINON" facultative termine la structure.

La structure conditionnelle est représentée par une ligne horizontale qui brise la séquentielle et détourne le flux de contrôle du programme. Les branches "SINON SI" et "SINON" sont représentées avec des schémas qui sont des variantes du schéma de la branche "SI" (figure 1.8).

Toutes les structures de sélection d'un langage cible sont présentées par une seule forme schématique. L'avantage de n'avoir qu'une seule représentation aide les intervenants qui ne connaissent pas le langage cible à voir l'algorithme utilisé. Une représentation simple et schématique facilite la communication entre concepteurs. Certains considèrent comme un désavantage de ne pas pouvoir distinguer les différentes structures de sélection offertes dans un langage de programmation, c'est le prix à payer pour avoir une représentation uniforme des structures et indépendante des langages de programmation.

Les expressions conditionnelles s'expriment en langage cible. À l'intéreur des branches conditionnelles on retrouve les trois mêmes types énoncés définis à la structure séquentielle.

Voici un exemple de structure conditionnelle avec branche "SI":

Fig. 1.6 Structure conditionnelle avec branche SI

Voici un exemple de structure conditionnelle avec branches "SI" et "SINON":

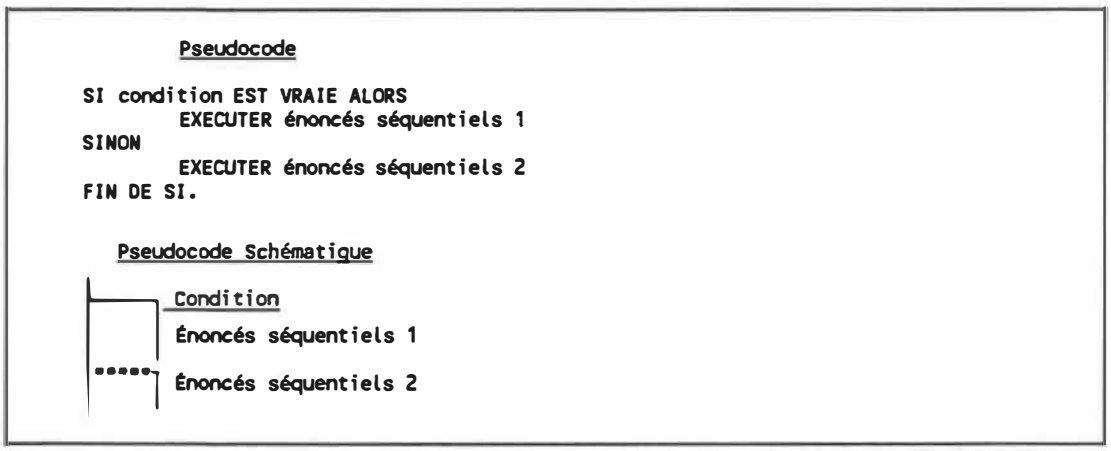


Fig. 1.7 Structure conditionnelle avec branches SI et SINON

Voici un exemple de structure conditionnelle avec branches "SI","SINON SI" et "SINON":

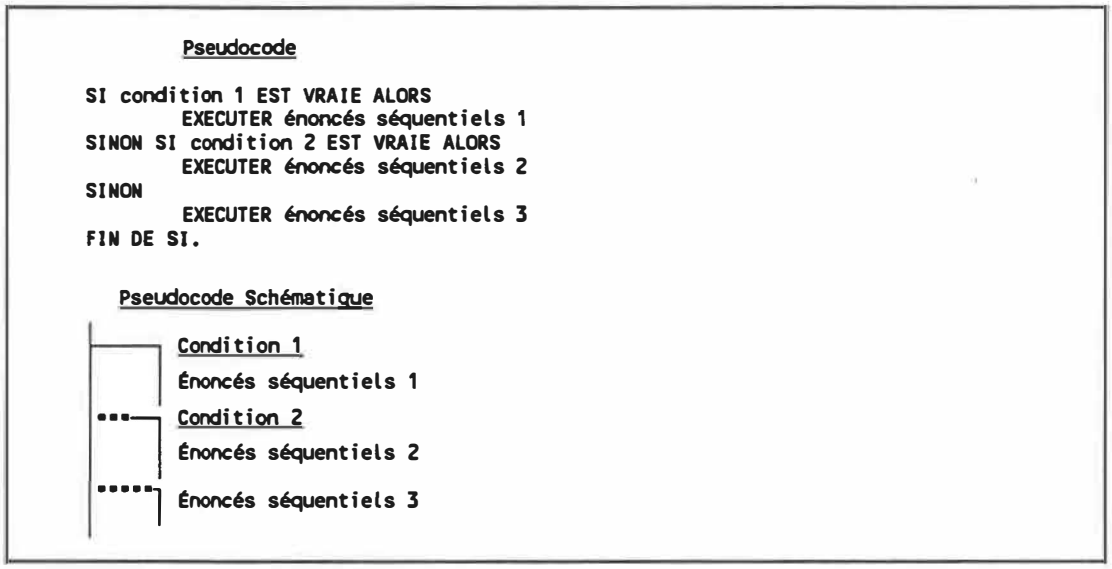


Fig. 1.8 Structure conditionnelle avec branches SI, SINON SI et SINON

3. La structure répétitive

La structure répétitive sert à représenter des énoncés séquentiels devant être répétés plusieurs fois. À nouveau, un seul schéma représente toutes les instructions répétitives d'un langage, avec ses avantages et inconvénients. La structure répétitive doit contenir au moins une condition de sortie (selon la syntaxe du pseudocode schématique). Par contre, plus d'une condition de sortie peut être présente. Ce qui est remarquable, c'est la possibilité d'écrire des énoncés séquentiels dans la condition de sortie, très pratique lorsqu'on a plus d'une condition de sortie.

Voici un exemple de structure répétitive avec condition de sortie placée au début:

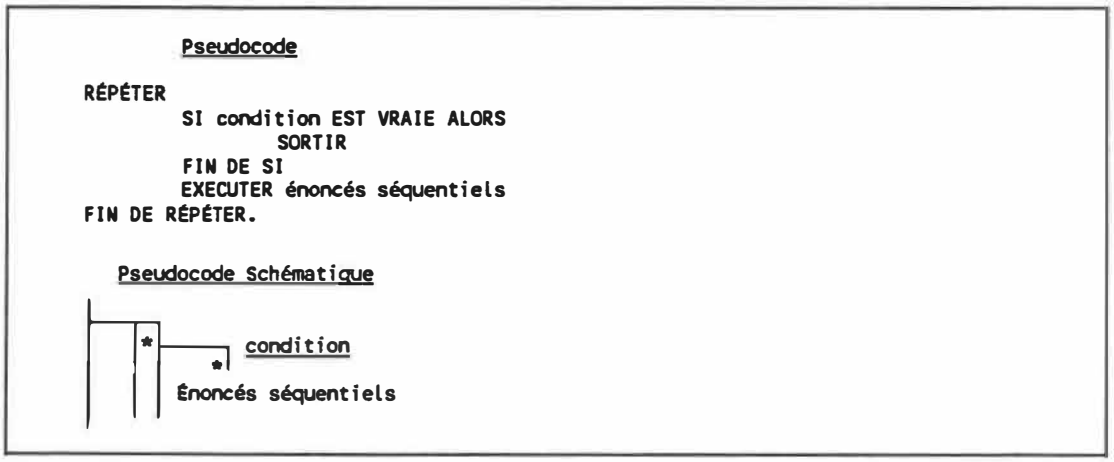


Fig. 1.9 Structure répétitive avec condition de sortie placée au début

Voici un exemple de structure répétitive générale:

Pseudocode

```
RÉPÉTER
        EXECUTER énoncés séquentiels 1
        SI condition 1 EST VRAIE ALORS
                SORTIR
        FIN DE SI
        EXECUTER énoncés séquentiels 2
        SI condition 2 EST VRAIE ALORS
                EXECUTER énoncés séquentiels 3
                SORTIR
        FIN DE SI
FIN DE RÉPÉTER.
```

Pseudocode Schématique

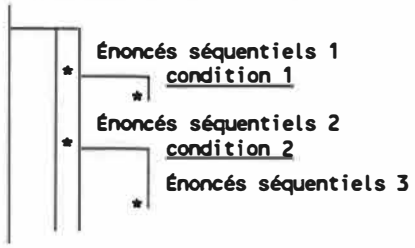


Fig. 1.10 Structure répétitive générale

D. Description formelle

Défini par une grammaire, le pseudocode schématique est un langage en soi. Voici la grammaire du pseudocode schématique principalement tirée de Robillard, P.N. [1986]:

Définition des symboles

::=	Est défini comme
< >	Symbole non terminal
{ }	Item répété
*	Occurrence incluant l'élément vide
+	Occurrence n'incluant pas l'élément vide
INSTRUCTION	Instruction qui suit la syntaxe du langage cible
CONDITION	Expression conditionnelle selon la syntaxe du langage cible
CARACTERE	Caractère ou symbole
CHIFFRE	Chiffre compris entre 0 et 9

Grammaire du Pseudocode Schématique

<schéma>	::=	{<raffinement>}+
<raffinement>	::=	"┌─┐" <titre> {<structure>}+ "└─┘"
<structure>	::=	<structure séquentielle> | <structure conditionnelle> | <structure répétitive>
<structure séquentielle>	::=	"|" <énoncé séquentiel>
<énoncé séquentiel>	::=	<commentaire narratif> | <commentaire opérationnel> | <énoncé en langage cible>
<titre>	::=	"-" {CARACTERE}*
<commentaire narratif>	::=	"-" {CARACTERE}*
<commentaire opérationnel>	::=	"[" {CHIFFRE}+ "]" {CARACTERE}*
<énoncé en langage cible>	::=	INSTRUCTION

```
<structure conditionnelle>   ::=   <si> {<sinon si>}* ( <sinon> | <vide> )

<si>                         ::=   "├────┐" CONDITION
                                   {<élément conditionnel>}+

<élément conditionnel>       ::=   "|    |" <énoncé séquentiel>

<sinon si>                   ::=   "|┈┈┐" CONDITION
                                   {<élément conditionnel>}+

<sinon>                      ::=   "|┈┈┈┐"
                                   {<élément conditionnel>}+

<vide>                       ::=   ε

<structure répétitive>       ::=   "├──┬─┐"
                                   <élément de rép. de base>
                                   {<élément de répétitive>}*

<élément de rép. de base>    ::=   {<corps>}+ <sortie>
                             |     {<sortie>}+ <corps>

<élément de répétitive>      ::=   <corps> | <sortie>

<corps>                      ::=   "|   | |" <énoncé séquentiel>

<sortie>                     ::=   "|   |*├────┐" CONDITION
                                   {<élément de sortie>}*
                                   "|   | |    *"

<élément de sortie>          ::=   "|   | |    |"
                                   <énoncé séquentiel>
```

Fig. 1.11 Grammaire du pseudocode schématique selon Robillard [1986]

II. LA TRADUCTION DU PSEUDOCODE SCHÉMATIQUE

Ce deuxième chapitre présente une définition de la traduction du pseudocode schématique, une description des genres de traduction et élabore des critères d'évaluation de la qualité du code produit.

A. Définition de la traduction

La traduction du pseudocode schématique peut se définir ainsi: action de transposer les structures schématiques en langage de programmation, tout en conservant leur sens original et en respectant la syntaxe du langage cible. Pour des considérations pratiques, nous ajoutons les spécifications suivantes:

- il faut que le code source produit soit dans une forme lisible, se rapprochant du code écrit à la main;
- il faut que le code source produit soit indenté en fonction du niveau d'imbrication des structures.

Ces spécifications supplémentaires ne sont pas nécessaires au bon fonctionnement du traducteur, cependant les éventuels utilisateurs aiment que le code produit automatiquement ressemble au code qu'ils écrivent. Il s'agit d'une composante psychologique qui favorise l'adoption d'un traducteur. Si l'on se compare aux premiers compilateurs, on constate le même scénario. Les compilateurs rendaient disponible une sortie documentée en langage assembleur (avec mnémoniques) du programme.

La tâche d'un traducteur est semblable à celle d'un compilateur. Pour cette raison, la méthode de réalisation de traducteurs est inspirée des techniques de conception de compilateurs.

Les principales phases de compilation d'un fichier source sont représentées par le schéma suivant:

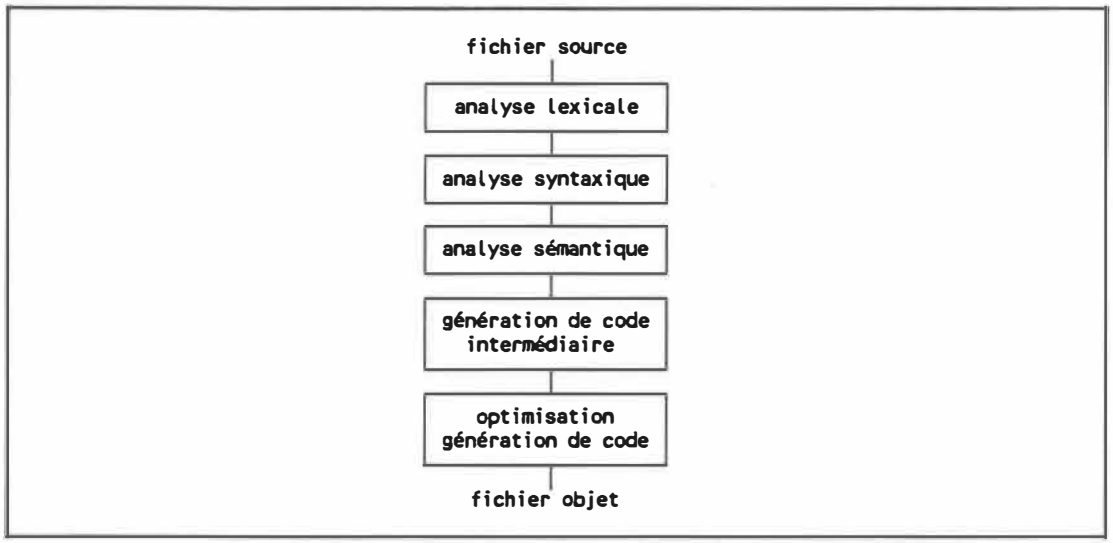


Fig. 2.1 Phases d'un compilateur

Pour un traducteur, les phases sont présentées comme ceci:

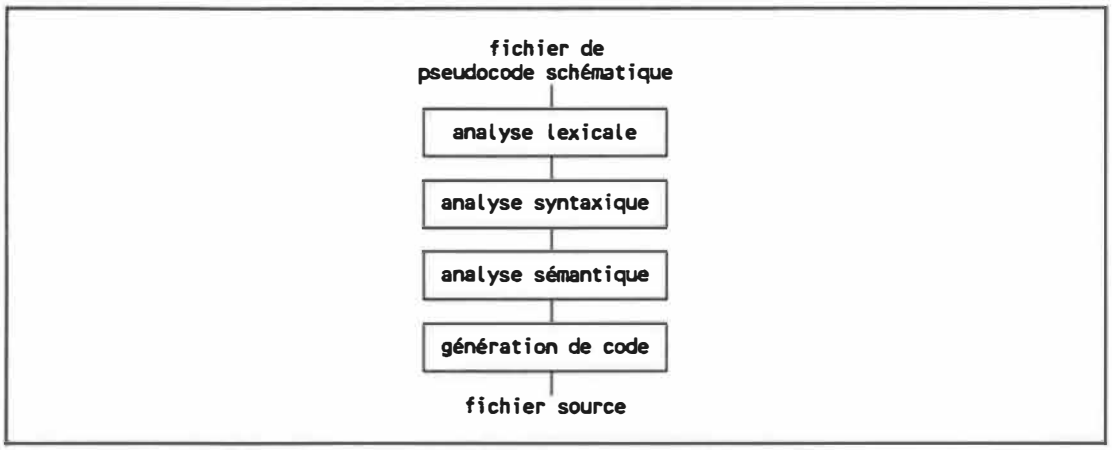


Fig. 2.2 Phases d'un traducteur

On constate que les différences entre la traduction du pseudocode schématique et la compilation sont mineures; il n'y a pas d'optimisation proprement dite dans la traduction du pseudocode schématique. Jusqu'à un certain point, on peut considérer le code intermédiaire d'un compilateur comme équivalent au code source d'un traducteur.

Voici une description sommaire des phases de traduction du pseudocode schématique:

L'analyse lexicale a le rôle de lire le code source, de former des jetons («tokens»); ensemble de caractères ordonnés selon des critères précis, et de les transmettre à l'analyseur syntaxique. Les jetons sont souvent spécifiés par des expressions régulières.

L'analyse syntaxique construit l'arbre syntaxique et s'assure que la chaîne de jetons constituée en phrase respecte les règles de grammaire du langage. La façon la plus courante de représenter la grammaire d'un langage est la BNF (Backus-Naur Form, voir Knuth [1964]) .

Les règles sémantiques sont plus dificiles à décrire de façon formelle, il n'existe pas de notation pour celles-ci. Pour les traducteurs, ces règles sont décrites sous forme textuelle.

La phase de la génération du code pour un traducteur est semblable à celle de la génération du code intermédiaire d'un compilateur. Elle consiste à traverser l'arbre syntaxique en appliquant les règles sémantiques afin de produire le code.

La réalisation d'un traducteur de pseudocode schématique peut être effectuée sans méthode précise, par contre un traducteur construit suivant une méthode systématique sera plus facile à maintenir et à modifier pour servir les demandes des utilisateurs.

B. Langages cibles

Le pseudocode schématique ne peut être traduit que dans certains langages informatiques. Le langage cible doit posséder un ensemble minimum d'instructions. Il est préférable qu'il contienne des énoncés pour supporter les structures conditionnelles et répétitives telles que définies dans le pseudocode schématique. Il est cependant possible de traduire le pseudocode schématique dans un langage qui ne possède pas de tels énoncés à condition que le langage ait au moins un énoncé de branchement et un énoncé de comparaison. Entre autres, la famille des langages assembleurs correspond à cette dernière catégorie.

Lorsque nous sommes en présence d'un langage cible possédant des instructions de conditionnelle et de répétitive ainsi que des énoncés de branchement et de comparaison, le traducteur doit opter pour l'ensemble d'instructions de conditionnelle et de répétitive afin de traduire le pseudocode schématique. Il s'agit ici d'opter pour le choix qui correspond le plus aux pratiques des programmeurs, afin que le code source produit leurs soit familier. Même si le code source produit automatiquement est le résultat d'une phase intermédiaire entre le pseudocode schématique et le programme exécutable, il doit être conforme à la bonne pratique.

C. Description des différents genres de traduction

En considération des besoins distincts des différents groupes d'utilisateurs du pseudocode schématique, le choix des énoncés de structure compris dans un langage de programmation peut, pour la traduction, se faire à différents niveaux de détail. En conséquence, nous présentons trois genres de traduction:

- la traduction directe, c'est-à-dire la traduction des règles de production de la grammaire du pseudocode schématique;
- la traduction complète, c'est-à-dire la traduction des structures complètes du pseudocode schématique;
- la traduction ciblée, c'est-à-dire la traduction orientée sur les structures de base du langage cible.

1. Traduction directe

Il s'agit de faire correspondre à chaque règle de production de la grammaire du pseudocode schématique son équivalent sémantique selon la syntaxe du langage cible. En d'autres mots, il faut trouver dans le langage cible les instructions de structure qui correspondent à chaque symbole terminal de la grammaire du pseudocode schématique. Mais attention, il faut aussi respecter la syntaxe du langage cible pour la transposition des énoncés.

Pour illustrer ce que nous venons de décrire, on peut prendre le symbole terminal correspondant au début d'une conditionnelle () qui doit être suivi par une expression écrite selon la syntaxe du langage cible (CONDITION). Voici un tableau de traduction qui présente cette règle et son équivalent en langage C.

Grammaire	Langage C
CONDITION	if (CONDITION) {

Fig. 2.3 Tableau de traduction

Dans cet exemple on constate que la traduction demande d'ajouter du texte avant et après la CONDITION. Il faut aussi tenir compte de l'indentation.

Voici une liste des symboles terminaux du pseudocode schématique auxquels il faut trouver une correspondance dans chaque langage cible automatisé.

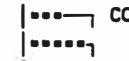
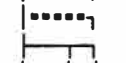
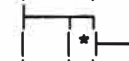
Descriptions	Symboles terminaux
début de raffinement	
fin de raffinement	
commentaire narratif	- CARACTERE *
commentaire opérationnel	[CHIFFRE +] CARACTERE *
énoncé en langage cible	INSTRUCTION
branche si	 CONDITION
branche sinon si	 CONDITION
branche sinon	
début de répétitive	
début de condition de sortie	 CONDITION
fin de condition de sortie	

Fig. 2.4 Symboles terminaux du pseudocode schématique

Nous convenons qu'il est difficile de reconnaître la fin de la structure conditionnelle ainsi que la fin de la structure répétitive car elles n'ont pas de symbole terminal associé. La fin d'une structure conditionnelle est détectée lorsque le schéma en début de ligne est "| |" et à la ligne suivante on retrouve le schéma "|". La fin d'une structure répétitive est détectée lorsque le schéma en début de ligne est "| | |" et à la ligne suivante on retrouve le schéma "|".

Nous allons définir une nouvelle grammaire mieux adaptée à la tâche de traduction au chapitre IV.

2. Traduction complète

Il s'agit de faire correspondre aux structures complètes du pseudocode schématique leur équivalent sémantique selon la syntaxe du langage cible. Mais quelles sont ces structures complètes? À la première analyse, elles sont trois, c'est-à-dire la structure séquentielle, la structure conditionnelle et la structure répétitive. En regardant de plus près, on distingue certains arrangements au sein d'une structure qui peuvent être dissociés et porter le titre de structure complète. Pour la structure conditionnelle on identifie différents types de construction comme:

- la structure conditionnelle constituée d'une seule branche SI;
- la structure conditionnelle constituée d'une branche SI et d'une branche SINON;

- la structure conditionnelle constituée d'une branche SI, d'une ou plusieurs branches SINON SI et facultativement d'une branche SINON.

L'avantage de dissocier ces différents types de construction d'une structure conditionnelle est de pouvoir utiliser plus efficacement les structures spécifiques d'un langage cible. À titre d'exemple, prenons le langage dBASE III qui offre la possibilité d'utiliser les instructions IF et IF ELSE ou CASE pour construire une conditionnelle; dans ce cas-ci, il est préférable d'utiliser la construction IF et IF ELSE pour les structures conditionnelles sans branche SINON SI et le CASE dans le cas contraire (dBase III [1984]).

Pour la structure répétitive le nombre de possibilités de construction est plus grand. Nous pouvons rencontrer des structures répétitives sans condition de sortie, avec une condition de sortie placée au début, au milieu ou à la fin et des structures répétitives à plus d'une condition de sortie dont une est placée au début, à la fin ou toutes au centre. Ces conditions de sortie peuvent contenir des énoncés ou ne pas en contenir.

La traduction complète doit faire correspondre à chaque type de structure son équivalent dans le langage cible.

3. Traduction ciblée

Ce type de traduction exige une connaissance très approfondie du langage cible. Il permet d'exploiter toutes les possibilités, au niveau des énoncés de structures du langage cible. La traduction ciblée fonctionne de la même façon que la traduction complète, mais en plus elle doit être capable de reconnaître certains énoncés ou expressions conditionnelles contenus dans le texte associé au schéma d'un algorithme en pseudocode schématique.

Un traducteur ciblé permet d'utiliser les structures d'un langage cible qui ne seraient pas disponibles avec un traducteur complet. Par exemple, un traducteur ciblé dont le langage cible serait le BASIC permettrait deux interprétations d'une structure répétitive avec une condition de sortie placée au début. Il peut la traduire à l'aide de l'instruction FOR ou faire l'usage d'un GOTO à la fin du corps de la répétitive. Le traducteur doit prendre en considération le texte contenu dans l'expression de la condition de sortie pour prendre sa décision.

Comme on le verra plus loin ce type de traduction peut s'avérer indispensable pour certains langages informatiques.

D. Critères de comparaison entre les genres de traduction

Afin de pouvoir comparer les différents genres de traduction entre eux , nous allons établir des critères. Ces critères sont regroupés en deux classes:

- critères qualitatifs;
- critères quantitatifs.

1. Les critères qualitatifs

On trouve dans cette classe de critères, un ensemble de critères qui devraient donner les mêmes résultats d'évaluation indépendamment du genre de traduction utilisée. Cette sous-classe est appelée "critères qualitatifs visés". Les résultats de l'évaluation de ces critères doivent correspondre aux buts visés décrits dans la spécification. Aussi, l'usage d'une méthode systématique de construction de traducteurs suppose que certains autres critères fassent partie de cette sous-classe.

a) Les critères qualitatifs visés

Voici la liste des critères qualificatifs visés par les spécifications ainsi qu'une brève description:

Indentation:	la mise en retrait de certaines parties de code afin de mettre en évidence leur appartenance à une structure donnée.

Structuration: le respect du principe de base: d'un point d'entrée et d'un point de sortie.

Disposition des commentaires: l'emplacement judicieux des commentaires opérationnels à l'intérieur du code source.

b) Les critères qualitatifs comparatifs pour l'utilisateur

Un critère qualitatif comparatif pour l'utilisateur a été retenu:

Disponibilité de toutes les instructions de structure: la possibilité offerte au programmeur de pouvoir utiliser toutes les instructions de structuration d'un langage (orienter la traduction).

c) Les critères qualitatifs comparatifs sur le code produit

D'autres critères qualitatifs peuvent être considérés sur le code source produit par les traducteurs. On peut énoncer:

Usage d'énoncés appropriés: l'utilisation des énoncés du langage cible les plus appropriés aux besoins de l'algorithme (par ex.: l'usage des énoncés reliés aux répétitives pour réaliser une conditionnelle sera fortement condamné par ce critère).

Présence de code superflu:	le code produit qui est non nécessaire peut être condamné selon ce critère.

2. Les critères quantitatifs

Il y a des critères que l'on peut mesurer de façon objective. Dans cette classe de critères on peut distinguer deux sous-classes: un ensemble de critères qui mesure l'efficacité du traducteur (exécution du programme de traduction) et un autre ensemble qui mesure celle du code produit.

a) Les critères quantitatifs associés à la traduction

Les critères quantitatifs associés à la traduction sont:

Le temps de traduction:	le temps que prend le traducteur à traduire un programme écrit en pseudocode schématique.
Espace mémoire nécessaire à la traduction:	le nombre de kilo-octets de mémoire nécessaire pour traduire un programme écrit en pseudocode schématique.

b) Les critères quantitatifs associés au code produit

Les critères quantitatifs associés au code produit sont:

Le nombre de lignes du programme traduit:	le nombre de lignes produites par le traducteur pour un programme de pseudocode schématique donné.
Le nombres de caractères contenus dans le programme traduit:	le nombre de caractères produits (produits et transcrits) par le traducteur pour un programme de pseudocode schématique donné.
Le temps de compilation:	le temps que prend le compilateur ou l'assembleur à compiler le code traduit.
Le temps d'exécution:	l'efficacité du code produit; le temps que prend le microprocesseur à exécuter le programme traduit par le traducteur.

Le but de nos travaux n'est pas de mesurer la qualité ou l'efficacité du programme dans son ensemble, mais de permettre de comparer les différents genres de traduction entre eux. Les critères de comparaison présentés poursuivent ce but. Même si les genres de traduction sont fondamentalement différents, leur évaluation est indispensable pour tirer des conclusions justes.

III. THÉORIE DES LANGAGES

Les définitions qui suivent sont tirées en partie d'Aho, Sethi et Ullman [1986].

A. La grammaire

La grammaire d'un langage présente les règles qui régissent sa syntaxe. Une grammaire G est définie par le quadruple <T,N,P,S> où:

T: un ensemble de symboles appelés **terminaux**.

N: un ensemble de symboles appelés **non-terminaux**.

P: un ensemble de **règles de production** dont la partie de gauche est un symbole non-terminal suivi d'une flèche en direction de la partie de droite où l'on retrouve une suite de symboles terminaux et non-terminaux.

S: un non-terminal qui indique le **symbole de départ**.

Le langage L(G) est l'ensemble de toutes les phrases constituées uniquement de symboles terminaux qui suivent la syntaxe spécifiée par la grammaire G.

Voici un exemple de règles de production utilisées pour la spécification de la syntaxe d'un calculateur (addition et soustraction):

```
expr          ->     expr + chiffre
expr          ->     expr - chiffre
expr          ->     chiffre
chiffre       ->     0
chiffre       ->     1
chiffre       ->     2
chiffre       ->     3
chiffre       ->     4
chiffre       ->     5
chiffre       ->     6
chiffre       ->     7
chiffre       ->     8
chiffre       ->     9
```

Fig. 3.1 Règles de production d'un calculateur

Dans cet exemple, les symboles terminaux sont: **+, -, 0, 1, 2, 3, 4, 5, 6, 7, 8 , 9** et les symboles non-terminaux sont: **expr** et **chiffre.**

Une expression (expr) peut être formée d'un chiffre, d'une expression plus un chiffre ou d'une expression moins un chiffre pour être conforme à la syntaxe du calculateur. Toutes expressions qui ne se conforment pas à ces règles comportent des erreurs de syntaxe.

Pour faciliter l'écriture nous convenons que les règles de production ayant la même partie de gauche peuvent être regroupées en une seule, chacune des parties de droite se succèdent en étant séparées par le symbole "|" qui signifie "ou".

```
expr     ->     expr + chiffre
          |     expr - chiffre
          |     chiffre

chiffre  ->     0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Fig. 3.2 Règles de production d'un calculateur - forme condensée

B. L'arbre syntaxique

Pour représenter l'expression suivante: 2+3-4, nous utilisons un arbre syntaxique.

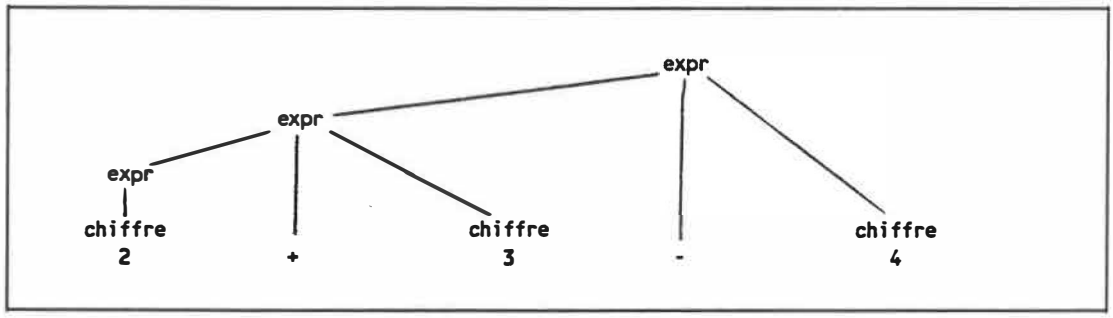


Fig. 3.3 Arbre syntaxique de "2+3-4" (association des opérateurs à gauche)

Plus formellement, un arbre syntaxique possède les caractéristiques suivantes:

- le sommet de l'arbre correspond au symbole de départ;
- les feuilles de l'arbre (derniers éléments) sont des symboles terminaux;
- les noeuds intérieurs sont des symboles non-terminaux;
- l'arbre se lit de gauche à droite, les fils d'un noeud lues de gauche à droite correspondent à la partie de droite des règles de production de la grammaire du langage.

C. La définition dirigée par la syntaxe

Une définition dirigée par la syntaxe consiste en une grammaire augmentée par l'addition d'attributs associés à chaque symbole.

Ces attributs peuvent prendre toutes sortes de valeurs, comme des chaînes de texte, des nombres, des adresses mémoires, etc. Il existe deux genres d'attributs:

- les attributs hérités: la valeur des attributs d'un noeud de l'arbre syntaxique est calculée à partir des informations en provenance des noeuds frères et père;
- les attributs synthétisés: la valeur des attributs est calculée à l'aide des informations retournées par leurs fils.

Une définition dirigée par la syntaxe est formée de règles de production servant à présenter la syntaxe et de règles sémantiques associées aux règles de production spécifiant les opérations à effectuer afin d'évaluer les attributs. Les attributs sont notés par le nom du symbole suivi d'un point et du nom de l'attribut.

Voici un exemple de définition dirigée par la syntaxe. Cette définition transforme une équation numérique de la notation infixe à la notation postfixe.

Règles de production	Règles sémantiques
expr -> expr1 + chiffre	expr.t = expr1.t || chiffre.t || "+"
expr -> expr1 - chiffre	expr.t = expr1.t || chiffre.t || "-"
expr -> chiffre	expr.t = chiffre.t
chiffre -> 0	chiffre.t = "0"
chiffre -> 1	chiffre.t = "1"
...	...
chiffre -> 9	chiffre.t = "9"

Fig. 3.4 Exemple de définition dirigée par la syntaxe

L'attribut "t" permet de construire une chaîne de texte représentant l'équivalent en notation postfix d'une équation numérique en notation infix; c'est un attribut synthétisé.

L'opérateur "| |" joint ses deux opérandes en une seule chaîne de texte.

Un arbre syntaxique peut aussi être présenté avec ses attributs, on le dit alors "arbre syntaxique avec attributs". Voici un exemple d'arbre syntaxique suivant les règles énoncées à la figure 3.4 pour la transformation de l'expression 2+3-4.

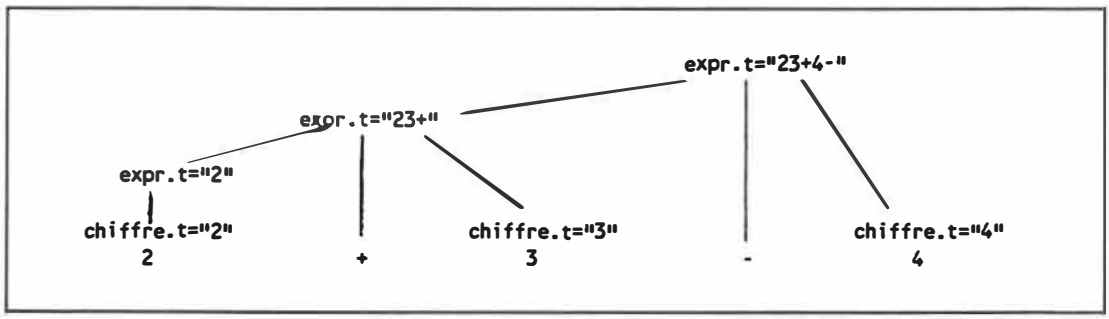


Fig. 3.5 Arbre syntaxique avec attributs de "2+3-4"

Voici un autre exemple de définition dirigée par la syntaxe, où cette fois nous utilisons un attribut hérité. Il s'agit de construire la valeur d'un nombre à partir de ses chiffres lus de gauche à droite comme on peut les lire dans un fichier.

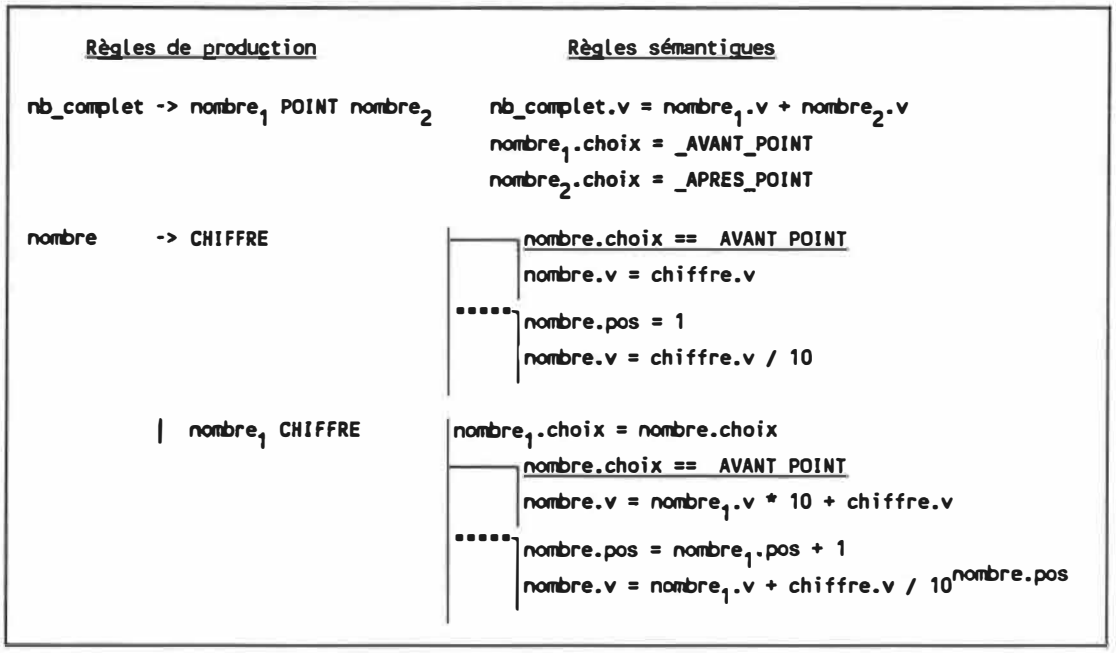
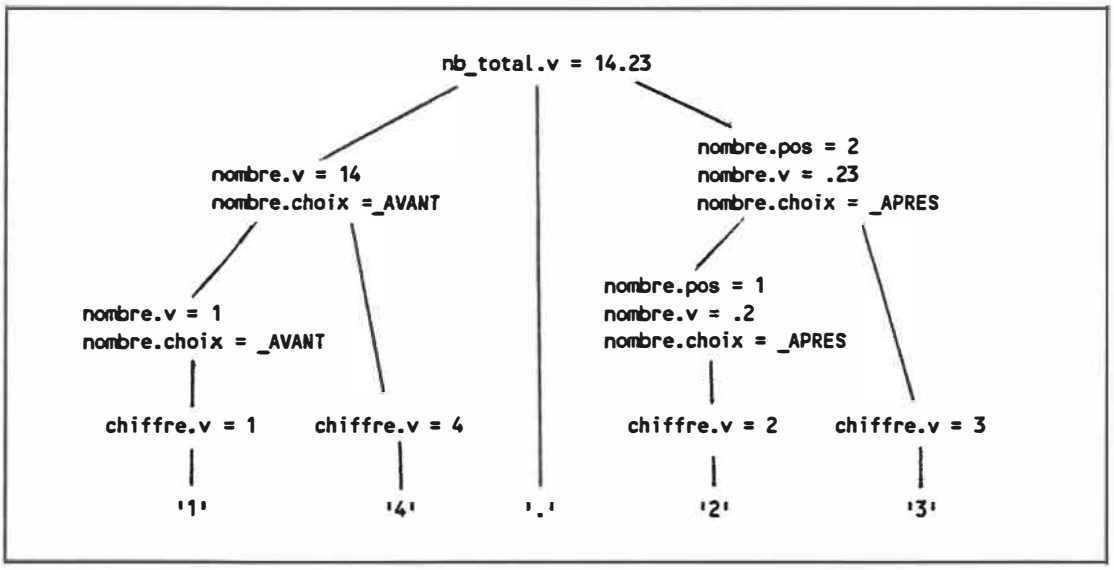


Fig. 3.6 Exemple de définition dirigée par la syntaxe avec attributs hérités

Les symboles terminaux sont écrits en lettres majuscules et les non-terminaux en lettres minuscules. Le symbole terminal "CHIFFRE" a un attribut "v" qui correspond à la valeur numérique du chiffre rencontré. L'attribut hérité "choix" est associé au symbole "nombre", il peut prendre deux valeurs soient "_AVANT_POINT" ou "_APRES_POINT" pour indiquer s'il s'agit du traitement de la partie entière ou de la partie fractionnaire. "_AVANT_POINT" et "_APRES_POINT" ne sont pas des symboles de la grammaire, mais seulement des constantes de valeur différente.

Voici un arbre syntaxique avec attributs qui suit la définition énoncée plus haut (figure 3.6) et qui représente l'évaluation de la séquence de nombre "14.23"



D. Schéma de transformation

Nous introduisons la notation de schéma de transformation car elle intègre les règles sémantiques à l'intérieur des règles de production. Cette notation est utilisée dans le chapitre sur la réalisation des traducteurs.

Exemple d'utilisation d'un schéma de transformation:

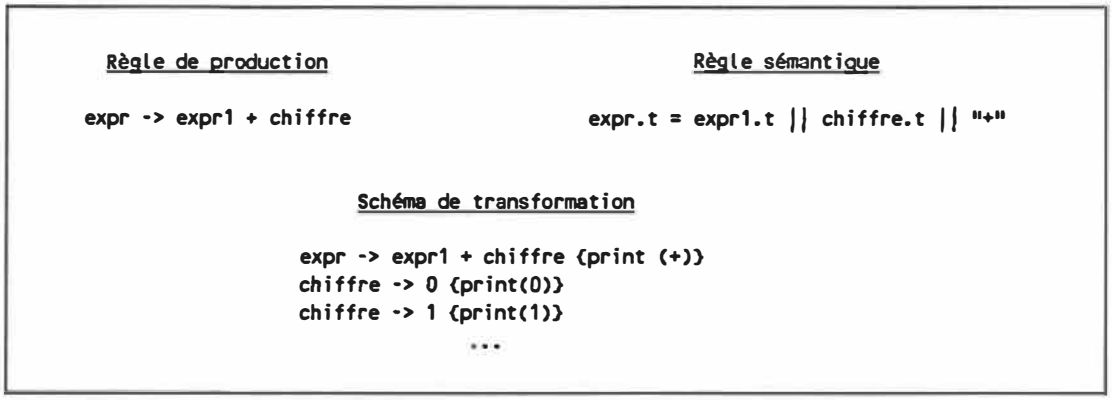


Fig. 3.8 Exemple de schéma de transformation

E. Construction d'un arbre syntaxique

Il arrive parfois que les attributs ne peuvent être évalués lors de l'analyse syntaxique. Alors, il faut créer, en premier lieu, un arbre syntaxique correspondant aux règles de production en premier lieu, puis parcourir celui-ci pour produire le code.

Voici un exemple de définition dirigée par la syntaxe qui ne peut être évaluée lors de l'analyse syntaxique.

Règle de production	Règle sémantique
vecteur -> id '[' num ']'	id.dimension = num.valeur
id -> NAME	id.add = alloc_mem(id.dimension)

Fig. 3.9 Exemple de définition dirigée par la syntaxe dont les attributs ne peuvent être évalués lors de l'analyse syntaxique

La valeur de "num" doit être connue lorsqu'on rencontre "id" pour, par exemple, réserver de l'espace mémoire. Il est nécessaire dans ce cas de construire un arbre syntaxique.

Pour construire un arbre syntaxique nous devons nous doter d'une ensemble de fonctions qui nous permet de créer des noeuds et de les relier entre eux:

- Mknode(type, valeur) crée un noeud; son premier argument sert à identifier le noeud et son deuxième, facultatif, sert à conserver une valeur. La fonction retourne un pointeur sur le noeud créé;
- Gphadd(nodeptr$_1$, nodeptr$_2$) joint deux noeuds par un lien orienté du premier argument (pointeur sur le noeud de départ) vers le deuxième (pointeur sur le noeud de destination).

Voici un exemple de définition dirigée par la syntaxe pour la construction d'un arbre syntaxique afin d'évaluer une expression.

Règles de production	Règles sémantiques
expr -> expr1 + nombre	expr.nodeptr = mknode (_OPADD) gphadd (expr.nodeptr,expr1.nodeptr) gphadd (expr.nodeptr,nombre.nodeptr)
expr -> expr1 - nombre	expr.nodeptr = mknode (_OPSOUS) gphadd (expr.nodeptr,expr1.nodeptr) gphadd (expr.nodeptr,nombre.nodeptr)
expr -> nombre	expr.nodeptr = nombre.nodeptr
nombre -> CHIFFRE	nombre.nodeptr = mknode (_NOMB,CHIFFRES.val)

Fig. 3.10 Définition dirigée par la syntaxe servant à la construction d'un arbre syntaxique

Le symbole terminal "CHIFFRES" possède un attribut "val" qui contient la valeur du nombre. La constante "_NOMB" identifie un noeud qui contient une valeur numérique. Les constantes "_OPADD" et "_OPSOUS" identifient respectivement des noeuds qui correspondent à l'opérateur d'addition et à l'opérateur de soustraction. Les symboles non-terminaux "expr" et "nombre" possèdent des attributs. Ces attributs

sont des pointeurs sur des noeuds de l'arbre en construction. On remarque que ces attributs sont synthétisés.

Voici un arbre syntaxique construit selon la définition dirigée par la syntaxe présentée à la figure 3.10 pour l'évaluation de 2+3-4.

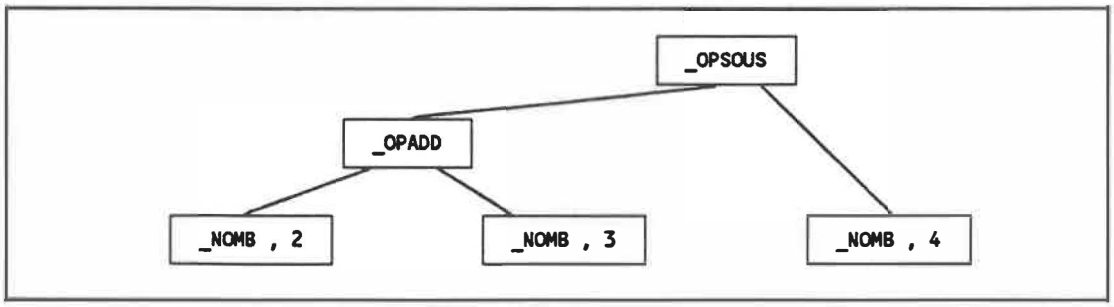


Fig. 3.11 Arbre syntaxique construit selon la définition de la figure 3.10 avec l'expression "2+3-4"

F. Parcours d'un arbre syntaxique

Nous reprenons les règles de production de l'exemple précédent (fig. 3.10) mais nous changeons les règles sémantiques afin d'évaluer l'expression contenue dans l'arbre syntaxique. Voici un extrait de définition dirigée par la syntaxe qui régit l'évaluation des attributs afin de produire le code.

Règles de production	Règles sémantiques
expr -> $expr_1$ + nombre	expr.v <- nombre.v + $expr_1$.v
expr -> $expr_1$ - nombre	expr.v <- $expr_1$.v - nombre.v
expr -> nombre	expr.v <- nombre.v

Fig. 3.12 Extrait d'une définition dirigée par la syntaxe

Nous avons la possibilité d'évaluer les attributs des symboles dans l'ordre que l'on désire. Pour l'addition, les règles sémantiques commandent d'évaluer d'abord l'attribut associé au nombre, puis le reste de l'expression; pour la soustraction, c'est le contraire. Cette liberté nous vient du fait que l'arbre syntaxique est conservé lors de l'analyse syntaxique.

Nous allons considérer l'arbre syntaxique suivant dont les noeuds sont numérotés dans un ordre quelconque.

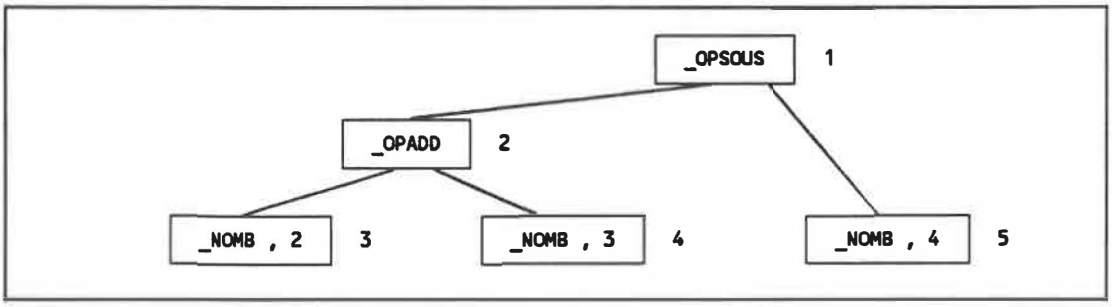


Fig. 3.13 Arbre syntaxique avec noeuds numérotés

Voici un programme qui nous permet d'évaluer le symbole "expr" en parcourant l'arbre syntaxique selon les règles sémantiques énoncées plus haut.

```
Fonction  expr(n)
n est un numéro de noeud
      opération(n) ==  OPADD
      résultat = nombre( fils(n,2) )
      résultat = résultat + expr( fils(n,1) )
      opération(n) ==  OPSOUS
      résultat = expr( fils(n,1) )
      résultat = résultat - nombre( fils(n,2) )
      - Évaluer le nombre courant
      résultat = nombre(n)
retourner résultat
```

Fig. 3.14 Programme récursif d'évaluation d'attibuts

La fonction "fils" introduite ici nous donne le numéro du noeud du $i^{ème}$ noeud fils d'un noeud. Lorsque le $2^{ème}$ argument est "1" cela signifie de retourner le premier fils du noeud "n", si c'est "2" alors on retourne le deuxième et ainsi de suite.

La fonction "expr" évalue une sous-expression par récursion, son argument lui indique un numéro de noeud qui doit être évalué. La fonction "nombre" détermine la valeur numérique d'un noeud.

En appelant la fonction "expr" avec le numéro du noeud "1" nous obtenons l'ordre d'évaluation suivante: 4, 3, 2, 5 et 1

L'algorithme d'évaluation d'expression est déduit des règles sémantiques et fixe l'ordre d'évaluation des noeuds d'un arbre syntaxique.

Ce que nous avons résumé ici est très important pour la suite de nos travaux, ces notions sont fondamentales. La théorie associée à d'autres notions fondamentales, telles l'analyse lexicale et l'analyse syntaxique n'est pas présentée ici parce que nous utilisons des outils de construction d'analyseurs. Ces outils sont présentés au chapitre: "Outils de construction".

IV. PRÉSENTATION DE LA MÉTHODE DE CONSTRUCTION

La méthode de construction de traducteurs présentée dans ce chapitre est le résultat de l'expérience acquise lors de la réalisation de plusieurs traducteurs de pseudocode schématique en langage orienté procédure. Nous avons participé à la réalisation de traducteurs pour les langages C, FORTRAN, dBASE III, Pascal, BASIC, COBOL et quelques langages assembleurs. Forts de cette expérience, nous présentons une méthode qui permet la réalisation d'un traducteur de pseudocode schématique de façon systématique.

La méthode de construction de traducteurs de pseudocode schématique s'applique aux trois genres de traductions présentés précédemment. La méthode se présente en trois variantes, chacune d'elles associées à un genre de traducteur. Les traducteurs construits selon des variantes différentes de la méthode de construction présentent des caractéristiques différentes. Nous ne cherchons pas à identifier la meilleure variante mais nous pouvons quand même extraire certaines caractéristiques de fonctionnement des traducteurs ainsi que du comportement de leurs codes sources produits. Des mesures quantitatives comme le temps de traduction, le nombre de lignes du code source produit et le temps de compilation du code source sont prises en considération afin de pouvoir choisir la meilleure variante de la méthode de construction de traducteurs en fonction des besoins de ses utilisateurs. Un autre élément plus difficile à quantifier peut aider à choisir la meilleure variante en fonction des besoins, il s'agit de l'acceptation de la forme du code source par les utilisateurs du traducteur; cette caractéristique peut être évaluée uniquement de façon subjective.

A. Grammaire des traducteurs

La grammaire du pseudocode schématique présentée par Robillard [1986] ainsi que celle présentée au début de ce mémoire ne peuvent être utilisées comme base de réalisation d'un traducteur. Nous allons donc définir, dans cette section, une nouvelle grammaire qui servira de base pour la construction de nos traducteurs. Afin de distinguer le pseudocode défini précédemment et notre pseudocode orienté pour la traduction, nous leurs donnons des abréviations différentes: "SPC" pour le pseudocode schématique présenté auparavant et "PCS" pour le pseudocode schématique orienté sur la traduction. La raison pour laquelle la traduction ne peut se faire sur le SPC est que sa grammaire ne permet pas une vision intégrée de l'algorithme. Dans le SPC, un raffinement est une entité en soi tandis que dans le PCS, le raffinement est le détail d'un commentaire opérationnel. Nous profitons de la nécessité de reformuler la grammaire pour changer les schémas par des symboles terminaux, plus faciles à éditer et analyser (lexical).

Notons d'abord qu'un nouvel élément s'est ajouté par rapport à la grammaire du SPC, c'est la notion de continuation. Une continuation sert à continuer sur une autre ligne une expression conditionnelle, un énoncé ou une autre continuation. Cette notion s'est ajoutée pour satisfaire l'édition électronique du pseudocode schématique. Un autre élément à remarquer: l'ajout de symboles terminaux identifiant la fin des structures conditionnelles et répétitives.

Grâce à l'amabilité des responsables du support de l'outil logiciel SCHEMACODE, nous avons pu réaliser un programme qui transforme un algorithme écrit à l'aide de cet outil en un algorithme équivalent selon la syntaxe du PCS.

Nous présentons la grammaire du PCS selon les conventions d'écriture suivantes: les symboles terminaux sont écrits en majuscules et les symboles non-terminaux sont écrits en minuscule. Le symbole de départ est: "algorithme".

```
algorithme                  ->   raffinement

raffinement                 ->   DRAF TITRE
                                 list_structure FRAF

list_structure              ->   ε
                            |    list_structure structure

structure                   ->   séquentielle
                            |    conditionnelle
                            |    répétitive

séquentielle                ->   élément-structural

élément_structural          ->   COMNA
                            |    commentaire_op
                            |    AFFEC continuée
                            |    TELQ

list_ele_structural         ->   ε
                            |    list_ele_structural élément_structural

commentaire_op              ->   raffinement

continuée                   ->   ε
                            |    continuée CONTI

conditionnelle              ->   DSI continuée list_ele_structural
                                 list_alt_cond alternative FCOND

list_alt_cond               ->   ε
                            |    list_alt_cond alternative_conditionnelle

alternative_conditionnelle  ->   DSISI continuée list_ele_structural

alternative                 ->   DSINO list_ele_structural
```

```
répétitive                  ->    DREP list_ele_répétitive FREP

list_ele_répétitive         ->    ele_répétitive
                            |     list_ele_répétitive ele_répétitive

ele_répétitive              ->    condition de sortie
                            |     élément_structural

condition_de_sortie         ->    DCS continuée list_ele_structural FCS
```

Fig. 4.1 Grammaire du PCS

Chaque symbole terminal représente une ligne dans un fichier construit selon la syntaxe du PCS. Voici les expressions régulières qui nous permettent de reconnaître ces symboles.

Symboles terminaux	Expressions régulières	Attributs
DRAF	"D" " " [0-9][0-9] "\n"	numéro de raffinement
FRAF	"F" "\n"	
TITRE	"T" " " [\040-\377]{0,78} "\n"	titre du raffinement
COMNA	"C" " " [\040-\377]{0,78} "\n"	commentaire narratif
AFFEC	"A" " " [\040-\377]{0,78} "\n"	texte de l'énoncé
TELQ	"<" " " [\040-\377]{0,78} "\n"	texte du telquel
CONTI	"+" " " [\040-\377]{0,78} "\n"	texte de la continuation
DSI	"I" " " [\040-\377]{0,78} "\n"	expression
DSISI	"S" " " [\040-\377]{0,78} "\n"	expression
DSINO	"L" "\n"	
FCOND	"E" "\n"	
DREP	"R" "\n"	
FREP	"Z" "\n"	
DCS	"X" " " [\040-\377]{0,78} "\n"	expression
FCS	"Q" "\n"	

Note: " " représente une espace.

Fig. 4.2 Expressions régulières servant à reconnaître les symboles terminaux du PCS.

Les symboles terminaux FRAF, DSINO, FCOND, DREP, FREP et FCS n'ont pas d'attribut, pour les autres, l'attribut est le texte de la ligne à l'exception des deux premiers caractères de la ligne et du "\n" de la fin.

Pour être complet dans notre description, nous devons mentionner une autre spécification. Lors de l'introduction de la notion de continuation, les concepteurs ont dû ajouter un mécanisme pour permettre de joindre, à la ligne courante, le texte d'une continuation situé à la ligne suivante. Pour ce faire, on doit terminer une ligne à continuer avec le caractère "^"; alors, la ligne suivante sera jointe à la ligne courante dans le fichier source produit, sinon elles se présenteront sur deux lignes séparées. Cette caractéristique n'est pas présente dans la grammaire du PCS car elle s'applique au niveau des attributs des symboles terminaux.

Enfin, une ligne de fichier PCS contient, au plus, 80 caractères; ce qui laisse 78 caractères pour la valeur de l'attribut. Cette limite nous est héritée du SPC. Les traducteurs de PCS peuvent tenir compte de cette limite lors de leur construction.

B. Langages choisis

La méthode de réalisation de traducteurs s'applique à tout traducteur dont le langage cible est un langage de programmation orienté procédure contenant des énoncés de structures répétitives et conditionnelles ou tout simplement des instructions de test et de saut. Dans le cadre de cette recherche nous voulons prouver la validité de la méthode en réalisant des traducteurs pour un ensemble restreint de langages connus. Ils ont été choisis parce qu'ils couvrent une grande partie des applications informatiques d'aujourd'hui. D'autres critères ont aussi été appliqués pour le choix de ces langages, telle la disponibilité de compilateurs. Trois langages informatiques ont été retenus à cette fin, soient:

- le langage assembleur dédié à une famille de microprocesseurs dont le 6502 fabriqué par plusieurs manufacturiers. Subséquemment appelé le langage assembleur 6502, ce langage a été choisi pour représenter les applications en langage assembleur toujours très en demande;
- le langage C principalement utilisé dans la réalisation de systèmes d'exploitation, de compilateurs, d'éditeurs et d'applications en temps réel; il est l'un des langages les plus utilisés actuellement, il est aussi supporté par l'outil logiciel SCHEMACODE;
- le langage Ada, résultat d'un effort du Département de la Défense des États-Unis; ce langage est très prometteur, il inclut les notions de génie logiciel sans les inconvénients des langages moins récents.

C. Description de la méthode

La méthode de construction de traducteurs se présente en trois variantes. Chaque variante est associée à un genre de traduction. En d'autres termes, il existe une variante de la méthode pour construire des traducteurs directs, une variante pour les traducteurs complets et une autre pour les traducteurs ciblés.

La variante directe et la variante complète sont divisées en deux parties, une partie indépendante du langage cible, appelée "de reconnaissance" et une autre, fonction du langage cible, appelée "de transcription". La reconnaissance se compose de l'analyseur lexical et de l'analyseur syntaxique. La transcription peut être intégrée à l'analyseur syntaxique ou détachée de celui-ci. Dans ce dernier cas, l'analyseur syntaxique charge le programme PCS dans une liste en mémoire et la traduction se fait à partir du parcours de cette liste. Quant à la variante ciblée, nous pouvons aussi identifier une étape de reconnaissance et une étape de transcription, par contre ces deux étapes sont fonction du langage cible.

Pour construire un traducteur, il faut d'abord réaliser un analyseur lexical puis un analyseur syntaxique, pour compléter la reconnaissance. Ensuite, et c'est surtout là que l'on constate le plus de différence entre les variantes, il faut établir et réaliser la transcription.

1. Variante de la méthode pour traducteurs directs

Nous utiliserons l'abréviation "VMD" pour identifier "la variante de la méthode de construction de traducteurs se basant sur les règles de production de la grammaire du pseudocode schématique" dans notre texte afin d'augmenter la lisibilité.

La VMD est la plus simple des variantes pour la construction de traducteurs. La transcription est intégrée à l'analyseur syntaxique par l'entremise d'actions associées à chaque règle de production. Voici les étapes qu'il faut suivre pour construire un traducteur selon la VMD:

1) Identifier les symboles terminaux du PCS et leurs attributs;
2) Réaliser un analyseur lexical qui reconnaîtra ces terminaux;
3) À partir des règles de grammaire du PCS, réaliser un analyseur syntaxique;
4) Identifier dans le langage cible, l'équivalent sémantique de chaque symbole terminal au sein des règles de production;
5) Établir les actions à effectuer pour traduire les règles de production dans leurs équivalents en langage cible en se servant des attributs synthétisés des symboles terminaux et non-terminaux;
6) Intégrer l'analyseur lexical et syntaxique au traducteur.

2. Variante de la méthode pour traducteurs complets

Nous utiliserons l'abréviation "VMC" pour identifier "la variante de la méthode de construction de traducteurs utilisant les structures du pseudocode schématique".

La VMC nous permet de construire des traducteurs qui produiront du code ressemblant, à peu de chose près, au code écrit à la main. Les traducteurs construits selon la VMC sont un peu plus fonction du langage cible que ceux construits selon la VMD. Cependant, la VMC ne s'intéresse toujours pas au contenu des attributs des symboles terminaux. Pour cette raison un traducteur construit selon la VMC peut tirer profit des résultats de la construction d'un traducteur direct pour un même langage. De même, une partie du travail lors de la construction d'un traducteur complet pour un langage donné peut être réutilisée lors de la construction d'un autre traducteur pour un langage semblable. Voici les étapes à suivre pour construire un traducteur complet:

1) Identifier les symboles terminaux du PCS et leurs attributs;
2) Réaliser un analyseur lexical qui reconnaîtra ces terminaux;
3) À partir des règles de grammaire du PCS, réaliser un analyseur syntaxique qui charge un fichier PCS dans une liste en mémoire;
4) Identifier les différentes structures complètes du PCS;
5) Déterminer les actions à effectuer pour reconnaître les structures complètes du PCS à l'intérieur de l'analyseur syntaxique et les attacher aux symboles terminaux chargés en mémoire;
6) Intégrer les actions à l'analyseur syntaxique;

7) Identifier dans le langage cible, l'équivalent sémantique de chaque structure complète;

8) Établir les actions à effectuer pour parcourir la représentation en mémoire du programme PCS et traduire les structures complètes en langage cible.

9) Intégrer l'analyseur lexical et syntaxique au traducteur.

3. Variante de la méthode pour traducteurs ciblés

Nous utiliserons l'abréviation "VMI" pour identifier "la variante de la méthode de construction de traducteurs orientée sur les structures de base du langage cible".

La VMI n'est pas seulement une amélioration de la VMC, comme c'est le cas de la VMC par rapport à la VMD. Il s'agit d'un changement de philosophie; ici on ne tente plus de trouver dans le langage cible un équivalent sémantique à un algorithme en PCS. On essaie plutôt de trouver une façon de spécifier, dans le pseudocode schématique, toutes les différentes constructions du langage cible. Un traducteur ciblé est construit entièrement en fonction d'un langage cible.

Par exemple pour dissocier une boucle "DO" d'une boucle "WHILE" en Fortran VAX/VMS, il faut analyser la syntaxe de l'expression conditionnelle d'une condition de sortie de boucle lorsqu'elle est placée au début. Pour ce faire, dans cet exemple, on peut se baser sur le fait qu'il ne peut y avoir de "=" dans une expression conditionnelle d'une boucle "WHILE" (VAX-11 FORTRAN [1982]). En présence d'un "=", on peut conclure qu'il s'agit d'une boucle "DO". Un autre moyen peut être utilisé lorsqu'il n'est pas possible de déduire la volonté du concepteur à travers le contexte, il s'agit d'ajouter un ou plusieurs mots qui permettent au traducteur de différencier les possibilités. Pour l'exemple du "FORTRAN" spécifié plus haut, on pourrait demander au programmeur d'écrire un "DO" ou un "WHILE" au début de l'expression de la condition de sortie pour spécifier sa volonté. Dans un cas comme dans l'autre, le traducteur est obligé d'analyser le texte de l'algorithme pour déterminer le type de transformation qu'il doit effectuer. Voici les étapes à suivre pour la réalisation d'un traducteur selon la VMI; on remarque que les six premières étapes sont les mêmes que pour la variante complète:

1) Identifier les symboles terminaux du PCS et leurs attributs;
2) Réaliser un analyseur lexical qui reconnaîtra ces terminaux;
3) À partir des règles de grammaire du PCS, réaliser un analyseur syntaxique qui charge un fichier PCS dans une liste en mémoire;
4) Identifier les différentes structures complètes du PCS;
5) Déterminer les actions à effectuer pour reconnaître les structures complètes du PCS à l'intérieur de l'analyseur syntaxique et de les attacher aux symboles terminaux chargés en mémoire;
6) Intégrer les actions à l'analyseur syntaxique;

7) Identifier dans le langage cible les différentes syntaxes qui peuvent être associées à une même structure complète;

8) Déterminer des critères qui permettront d'associer à une structure complète un seul équivalent sémantique dans le langage cible;

9) Établir les actions à effectuer pour parcourir la représentation en mémoire du programme PCS et appliquer les critères de différentiation pour traduire les structures complètes différenciées en langage cible.

10) Intégrer les composantes du traducteur.

Dans toutes les variantes, aucune des étapes ne nous garantit que le code produit soit de forme acceptable pour l'utilisateur, au niveau de l'indentation, du traitement des continuations, etc... Tout ce que nous pouvons dire, c'est que le concepteur d'un traducteur doit tenir compte de ces facteurs dans le choix des formes sémantiques équivalentes suivant la syntaxe du langage cible.

V. OUTILS DE CONSTRUCTION

A. Générateurs d'analyseurs

Les outils LEX, YACC et PREP sont standard sur UNIX. Il existe, en plus, des versions de ces outils qui ont été portées sur d'autres systèmes (Johnson et Lesk [1978] et Brown [1984]).

1. LEX

Les analyseurs lexicaux, nécessaires à la reconnaissance des symboles terminaux de nos traducteurs, sont construits à l'aide d'un outil de construction d'analyseurs lexicaux appelé "LEX". LEX est en fait un compilateur qui accepte le "langage LEX" et produit un programme en langage C qui, une fois compilé, transforme le texte à analyser en un ensemble de jetons (Lesk et Schmidt [1978]).

a) Syntaxe de LEX

Un programme typique écrit en langage LEX se présente de la façon suivante:

```
%{
        Déclaration des communications.
%}
        Déclaration des définitions.
%%
        Règles de transformation.
%%
        Fonctions utilitaires.
```

Fig. 5.1 Quatre grandes sections du langage LEX

On retrouve, entre autres, dans la section de déclaration des communications, la déclaration de valeurs numériques constantes servant à identifier certains symboles terminaux ainsi que la déclaration de leurs attributs. Dans la section de déclaration des définitions, on retrouve la définition de certaines expressions régulières utilisées à plusieurs reprises dans la section de transformation.

La section de transformation se présente comme ceci:

```
expression régulière1                    { actions1 }
expression régulière2                    { actions2 }
...
```

Fig. 5.2 Syntaxe des règles de transformation

Voici les opérateurs d'expressions régulières permis normalement par LEX, ils sont présentés par ordre décroissant de préséance. Dans cette table, "c" correspond à un caractère seul, "r" à une expression régulière et "s" correspond à une chaîne de texte.

Expression	Reconnaît	Exemple
1) c	Caractère non-opérateur	a
2) "s"	Chaîne de caractères telle quelle	"*a."
3) \c	Caractère tel quel	*
4) [s]	Tous caractères compris dans s	[abc]
5) [c_1-c_2]	Tous caractères de c_1 à c_2	[a-z]
6) [^s]	Tous caractères sauf ceux compris dans s	[^0-9]
7) .	Tous caractères sauf «new line»	a.*b
8) ^r	Expression en début de ligne	^abc
9) r$	Expression en fin de ligne	abc$
10) r?	Expression optionnelle	a?
11) r*	0 ou plus occurrence de r	^[a-z]*
12) r+	1 ou plus occurrence de r	[0-9]+
13) r{m,n}	De m à n occurence de r	[0-9]{1,5}
14) r_1 r_2	r_1 puis r_2	"0x" [0-9A-F]*
15) r_1 | r_2	r_1 ou r_2	"ab" | "ba"
16) (r_1)	Expression régulière r_1	("ab" | "ba")*
17) r_1 / r_2	r_1 uniquement suivi de r_2	^[\t]* / "\n"

Fig. 5.3 Expressions régulières de LEX

Certains des opérateurs d'expressions régulières mentionnés précédemment peuvent être construits, lorsqu'ils ne sont pas disponibles, à l'aide d'un ensemble restreint d'opérateurs. Par exemple, l'expression régulière (a?b) peut être remplacée par l'expression (ab | b). C'est pourquoi, ce n'est pas toutes les implantations de LEX qui présentent un aussi vaste ensemble d'opérateurs d'expressions régulières. Notre version de LEX ne supporte que les opérateurs # 1,2,3,4,5,11,14,15,16 et 17. Cet ensemble plus restreint est tout de même suffisant pour notre usage.

L'outil LEX que nous possédons engendre un analyseur lexical en langage C. Il est possible de construire un générateur d'analyseurs lexicaux pour d'autres langages compte tenu du fait que le principe de fonctionnement de l'analyseur lexical n'est pas fonction du langage cible.

b) Fonctionnement de LEX

Nous ne détaillons pas la méthode à suivre pour construire un générateur d'analyseurs lexicaux comme LEX. Cette information est disponible entre autres dans Aho, Sethi et Ullman [1986]. Il est tout de même intéressant de mentionner les six grandes étapes de construction d'un analyseur lexical à partir d'un programme en langage LEX:

- Analyse lexicale du langage Lex;
- Analyse syntaxique;
- Création d'un automate à états finis non déterministe (NFA) à partir des expressions régulières;
- Transformation du NFA en DFA (automate déterministe);
- Optimisation du DFA;
- Génération de code.

Il est à noter qu'il existe des méthodes qui permettent de construire un DFA sans passer par un NFA.

2. YACC

YACC est un générateur d'analyseurs syntaxiques. Il intègre un mécanisme de manipulation d'attributs synthétisés ainsi qu'un embryon de mécanisme pour les attributs hérités. La syntaxe de YACC est semblable à celle d'un schéma de transformation où il est possible d'accéder aux attributs des symboles terminaux et non-terminaux.

YACC engendre un analyseur basé sur un automate à états finis et une pile. L'analyseur syntaxique, ainsi produit, est construit pour analyser selon la technique LALR («lookahead» LR). C'est-à-dire qu'il lit l'entrée de gauche à droite ("L" «left-to-right scanning») et qu'il analyse la syntaxe des feuilles vers la racine ("R" «bottom-up parsing») avec un symbole d'avance («lookahead») (Johnson [1978]).

L'algorithme d'analyse est basé sur deux principes:

- le passage d'un état à un autre («shift»);
- la réduction d'une règle à un symbole («reduce»).

YACC permet la spécification de relations de précédence des opérateurs à l'aide des commandes "%left", "%right", "%noassoc" et "%prec". Il permet aussi la reconnaissance rapide d'une erreur de syntaxe et supporte la possibilité de poursuivre après une telle erreur («error-recovery mecanism»).

De par son implantation, YACC favorise l'usage de la récursion à gauche dans l'écriture des règles de grammaire.

a) Syntaxe de YACC

La syntaxe d'un programme YACC est simple, elle se compose de quatre sections:

```
%{
        Déclaration des communications.
%}
        Déclaration des symboles.
%%
        Règles de transformation.
%%
        Fonctions utilitaires.
```

Fig. 5.4 Quatre grandes sections du langage YACC

Dans la section de déclaration des communications, on retrouve les déclarations en langage C des variables et des constantes.

La section de déclaration des symboles réunit la déclaration des symboles terminaux et non-terminaux ainsi que la déclaration de leurs attributs exprimés sous la forme d'une déclaration de type. Les symboles n'ont pas obligatoirement d'attributs associés. Dans cette même section, on retrouve aussi la spécification de la précédence des opérateurs et la déclaration obligatoire du symbole de départ.

La section des règles de transformation suit une syntaxe qui ressemble au schéma de transformation présenté précédemment, mais on a remplacé la flèche ("->") par deux points (":"). Une section réservée aux fonctions utilitaires est disponible à la fin d'un programme YACC.

b) Manipulation des attributs

YACC propose un mécanisme très simple pour manipuler les attributs synthétisés. Il associe à chaque symbole du schéma de transformation, une variable nommée en fonction de la position du symbole dans la règle. Voici, pour illustrer l'usage de cet outil, un exemple qui réalise un interpréteur d'expressions mathématiques simples:

```
%start  liste

%token  <int> NOMBRE

%left   '+' '-'
%left   '*' '/'
%left   MOINS_UNU

%%

liste   :
        |  liste expr '\n'              { printf("Expression = %d\n",$2); }
        ;

expr    :  '(' expr ')'                { $$ = $2; }
        |  expr '+' expr               { $$ = $1 + $3; }
        |  expr '-' expr               { $$ = $1 - $3; }
        |  expr '*' expr               { $$ = $1 * $3; }
        |  expr '/' expr               { $$ = $1 / $3; }
        |  '-' expr %prec MOINS_UNU    { $$ = - $2; }
        |  NOMBRE                      { $$ = $1; }
        ;
%%
```

Fig. 5.5 Programme YACC pour l'interprétation d'expressions

La première partie du programme indique que le premier symbole qu'il faut réduire est le symbole "liste". Ensuite, on retrouve la déclaration des symboles terminaux "NOMBRE", '+', '-', '*', '/' et "MOINS_UNU". "NOMBRE" possède un

attribut de type "int", on remarque que l'attribut ne porte pas de nom spécifique. Les autres symboles n'ont pas d'attribut, par contre, ils sont des opérateurs associatifs par la gauche («left-associative operator»). Le '+' et le '-' sont au même niveau de préséance mais ils sont moins importants que le '*' et le '/' qui sont à leur tour moins importants que "MOINS_UNU" en ce qui concerne l'ordre d'évaluation des opérateurs.

Pour la section de transformation, on peut dire que lorsqu'on rencontre un '\n' on affiche le résultat de l'expression. Dans le présent schéma, la phrase terminée par '\n' qui ne contient pas d'expression constitue une erreur de syntaxe.

Pour ce programme syntaxique en YACC, il doit y avoir quelque part une contrepartie lexicale. Cette contrepartie reconnaît un "NOMBRE" et lui associe une valeur. L'analyseur lexical doit aussi reconnaître les opérateurs '+', '-', '*', '/' et la constante '\n'. "MOINS_UNU" (pour moins unuaire) n'est pas reconnu par l'analyseur lexical, il sert uniquement à donner une plus forte préséance à l'opérateur '-' lorsqu'il est rencontré seul.

En résumé, YACC est un outil indispensable pour la construction rapide d'un analyseur syntaxique associé à un langage que l'on peut décrire à l'aide d'une grammaire sans contexte («context free grammar»).

3. PREP

PREP est un préprocesseur pour YACC. À partir d'un programme en langage PREP, PREP engendre un programme YACC. Le langage PREP est semblable au langage YACC, il est augmenté d'un mécanisme plus avancé de traitement des attributs hérités et synthétisés. Plutôt que de référencer un attribut par la position du symbole associé, PREP utilise des noms pour les attributs du schéma (Katwijk [1983]).

Voici le même exemple de calculatrice qui a été présenté avec YACC. Mais comme cet exemple n'utilisait pas d'attribut hérité, nous avons changé la spécification pour permettre une expression vide, ce qui implique l'initialisation de la valeur de départ de l'expression. Le but de l'exemple est de montrer l'usage des attributs hérités et non pas de proposer une solution à l'interprétation des expressions.

```
%start  liste

%token  <int> NOMBRE

%left  '+' '-'
%left  '*' '/'
%left  MOINS_UNU

%attribute init(^int)
%attribute expr(^int)
%attribute sous_expr(|int,^int)

%%

liste          :
               | liste init(^val_init)  sous_expr(|val_init,^result) '\n'
                                                { printf("Expression = %d\n",$result); }
               ;

init(^val_init) :                               { $val_init  = 0; }
               ;

sous_expr(|val_init,^result) :                  { $result = $val_init; }
               | expr(^res_expr)                { $result = $res_expr; }
               ;

expr(^res_expr) : '(' expr(^res1) ')'           { $res_expr = $res1; }
               | expr(^res1) '+' expr(^res2)    { $res_expr = $res1 + $res2; }
               | expr(^res1) '-' expr(^res2)    { $res_expr = $res1 - $res2; }
               | expr(^res1) '*' expr(^res2)    { $res_expr = $res1 * $res2; }
               | expr(^res1) '/' expr(^res2)    { $res_expr = $res1 / $res2; }
               | '-' expr(^res1) %prec MOINS_UNU { $res_expr = - $res1; }
               | NOMBRE                         { $res_expr = $1; }
               ;

%%
```

Fig. 5.6 Programme PREP d'évaluation d'expressions

PREP est avantageux lorsqu'il y a des attributs hérités à manipuler, il augmente aussi la visualisation du cheminement des attributs. Il faut par contre se discipliner lors du choix des noms des attributs dans les schémas de transformation car cela peut facilement devenir confus.

B. Types abstraits de données de haut niveau

Un type abstrait de données est une classe d'objets qui fixe un ensemble de constructeurs et d'opérateurs qui lui est propre. Un ensemble de types abstraits de données de base est disponible dans les langages informatiques. Plusieurs langages permettent l'usage de chaînes de texte de longueur variable («string»), de structures de données («record») et bien d'autres (Booch [1987b]).

La distinction entre un type abstrait de données de base et un type abstrait de données de haut niveau est difficile à faire. On peut convenir qu'un type abstrait de données qui n'est pas défini pour un langage de programmation est un type abstrait de haut niveau pour ce langage.

La réalisation des traducteurs de pseudocode schématique est faite en langage C. Dans ce langage il n'existe pas de pile («stack»), de queue, de liste ou de graphe (Harbison et Steele [1987]).

Un ensemble de constructeurs sur des types abstraits de données de haut niveau a été développé dans le cadre de cette recherche. Tous ne sont pas obligatoirement utilisés dans la réalisation des traducteurs, mais tous ont été nécessaires à un moment ou un autre dans l'exploration d'une méthode de construction de traducteurs de pseudocode schématique. La conception de la représentation interne des types abstraits de données a été faite de telle sorte que son implantation soit indépendante de son usage. C'est-à-dire que les constructeurs

peuvent servir à une multitude d'applications différentes qui nécessitent l'utilisation de tels types de données.

Les types abstraits de données de haut niveau sont construits à l'aide de types abstraits de données de base et de types non-abstraits de données. Le domaine de certains types non-abstraits de données en langage C est fonction du matériel (ordinateur). Par exemple pour un DEC VAX, le type "int" peut prendre les valeurs -2,147,483,648 à +2,147,483,647 (Programming in VAX C [1985]) et sur IBM PC, un "int" peut prendre les valeurs -32,768 à +32,767 (Microsoft C Compiler [1986]). Pour rendre l'ensemble des constructeurs de types abstraits de données le plus portable possible, nous définissons de nouveaux types non-abstraits qui seront ajustés selon le matériel utilisé afin de rendre leur domaine indépendant du matériel.

Nouveau type	Domaine	Type du langage C
UNS8	0 à 255	unsigned char
UNS16	0 à 65535	unsigned short
UNS32	0 à 4,294,967,295	unsigned long
INT16	-32768 à +32767	short
INT32	-2,147,483,648 à +2,147,483,647	long
FLOAT32	approx. 3.4E-38 à 3.4E+38 4 octets (représentation IEEE)	float
FLOAT64	approx. 1.7E-308 à 1.7E+308 8 octets (représentation IEEE)	double

Fig. 5.7 Redéfinitions des types non-abstraits utilisés

Les nouveaux types non-abstraits sont utilisés pour la réalisation des constructeurs de types abstraits de données et pour la réalisation des traducteurs de pseudocode schématique. Ces nouveaux types non-abstraits sont définis lorsqu'on inclut le fichier "rgltypes.h".

Tous les types abstraits de données s'utilisent de manière semblable. Il faut d'abord créer une instanciation d'un type abstrait de données. Ensuite, lorsqu'on utilise un constructeur, il faut toujours lui indiquer l'instanciation à laquelle l'opération s'applique. À la fin, une fonction de fermeture permet de libérer la mémoire associée à une instanciation. Cette approche nous permet d'utiliser plusieurs instanciations d'un même type abstrait de données simultanément.

L'information conservée par un type abstrait de données est fonction de l'implantation où elle est utilisée. La constitution (type) de cette information est libre de toutes contraintes de dimension et par conséquent est prise en charge par l'utilisateur. En d'autre termes, toutes les implantations des constructeurs des types abstraits de données voient l'information de l'utilisateur à travers un pointeur sur une donnée de type inconnu.

L'implantation des types abstraits de données impose que la dimension de l'information qui est conservée soit constante pour une instanciation donnée. Les constructeurs gèrent l'espace réservé pour l'information.

La librairie des constructeurs des types abstraits de données s'utilise avec un fichier inclus du nom de "rgl.h", qui contient l'information sur le contenu des types "STACK", "QUEUE", "LIST", "LISTIN", "GRAPH" et "GRAPHIN". Ce fichier contient aussi la déclaration du type de retour et des types des paramètres des fonctions de la librairie.

Afin de mieux illustrer ce qui vient d'être mentionné, voici une brève description des fonctions qui sont disponibles pour gérer une PILE.

1. Pile

La pile a été le premier élément de la librairie de types abstraits de données de haut niveau à être réalisé. Le nom de toutes les fonctions s'appliquant à ce type abstrait de données débute par "stk". La fonction d'instanciation s'appelle "stkopen" et elle demande en paramètre la dimension d'un élément de la pile. Cette fonction retourne un pointeur sur une structure "STACK". La fonction qui sert à libérer une instanciation d'une pile s'appelle "stkclose", elle ne retourne rien et doit avoir comme paramètre une instanciation ("STACK *"). Des fonctions pour modifier le contenu de la pile sont aussi disponibles, les fonctions "stkpush" et "stkpop" servent respectivement à ajouter et à retirer un élément de la pile. La fonction "stkempty" sert à interroger la pile pour savoir si elle est vide. Deux dernières fonctions servant à voir le contenu du dessus de la pile, "stktop", et servant à voir le $n^{ème}$ élément de la pile, "stkview", sont aussi disponibles.

VI. RÉALISATION DES TRADUCTEURS

> La simplicité de la solution d'un problème complexe est proportionnelle à la quantité de travail investi pour la trouver.
>
> Alain GRENIER

Les traducteurs de pseudocode schématique sont réalisés en langage C. Ils utilisent les types non-abstraits de données redéfinis, les types abstraits de base du langage et les types abstraits de données de haut niveau tels que définis précédemment. Ils sont contruits à l'aide des outils logiciels LEX, YACC, PREP et SCHEMACODE. Les programmes sont contenus dans des fichiers, le nom du fichier indique son contenu. Le nom d'un fichier commence par le nom de la variante utilisée, par exemple "VMD" pour la variante de construction de traducteurs directs; suivi d'un "_" (soulignement) et d'une lettre identifiant le langage cible du traducteur, "C" pour le langage C, "A" pour le langage assembleur 6502 et "D" pour le langage Ada. Le tout est à nouveau suivi d'un "_" et d'un dernier caractère défini comme suit:

C	Programme principal et sous-programmes en langage C;
1	Table principale en langage LEX;
2	Table secondaire en langage LEX;
Y	Schéma de transformation en langage YACC;
P	Schéma de transformation en langage PREP;
H	Fichier inclus pour un traducteur (déclaration des variables globales).

Ces fichiers ne sont pas nécessairement tous utilisés pour un traducteur selon une variante. Les fichiers utilisés pour un traducteur selon une variante donnée sont présents dans le systéme sous ces extensions:

.SPC Fichier d'édition par SCHEMACODE;

.C Fichier source traduit par SCHEMACODE (traducteur C) ou engendré par LEX ou YACC;

.LXI Fichier traduit par SCHEMACODE (traducteur C) en langage LEX;

.Y Fichier traduit par SCHEMACODE (traducteur C) ou engendré par PREP en langage YACC;

.ATR Fichier traduit par SCHEMACODE (traducteur C) en langage PREP;

.H Fichier inclus traduit par SCHEMACODE (traducteur C) ou produit par YACC.

Toutes les modifications sont toujours faites au niveau de SCHEMACODE.

A. Traducteurs construits selon la variante directe

Nous présentons dans cette section l'application de la variante directe de la méthode de construction de traducteurs. La variante est appliquée à construire un traducteur en langage C, un traducteur en langage Ada et un autre en langage assembleur 6502.

1. Traducteur C direct

Le traducteur C traduit un programme PCS contenu dans un fichier dont le nom est de la forme "programme.pcs" en un programme, qui suit la syntaxe du langage C, contenu dans un fichier dont le nom est de la forme de "programme.c".

a) Identification des terminaux

La première étape consiste à identifier les symboles terminaux du PCS et leurs attributs. Les terminaux de la grammaire du PCS sont:

1)	DRAF	Début de raffinement
2)	FRAF	Fin de raffinement
3)	TITRE	Titre du raffinement
4)	COMNA	Commentaire narratif
5)	AFFEC	Énoncé en langage cible
6)	TELQ	Énoncé traduit à partir de la colonne 1
7)	CONTI	Continuation

8)	DSI	Début d'une branche SI d'une conditionnelle
9)	DSISI	Début d'une branche SINON SI
10)	DSINO	Début d'une branche SINON
11)	FCOND	Fin d'une structure conditionnelle
12)	DREP	Début d'une structure répétitive
13)	FREP	Fin d'une structure répétitive
14)	DCS	Début d'une condition de sortie
15)	FCS	Fin d'une condition de sortie

Chaque symbole terminal représente une ligne dans le fichier PCS. Le texte de cette ligne est l'attribut de ce terminal. Les terminaux FRAF, DSINO, FCOND, FREP, et FCS n'ont pas de texte associé.

b) Réalisation de l'analyseur lexical

La seconde étape consiste à réaliser un analyseur lexical qui reconnaîtra les terminaux du PCS. Nous utilisons l'outil LEX, le générateur d'analyseurs lexicaux, pour réaliser cet analyseur. Les expressions régulières servant à reconnaître les symboles terminaux ont été présentées précédemment. L'analyse lexicale se fait en deux temps, d'abord on doit reconnaître le symbole terminal de la ligne courante et ensuite nous devons déterminer son attribut. Pour ce faire, nous concevons deux tables de transition d'états pour l'automate déterministe de LEX; une table pour reconnaître le symbole et l'autre, son attribut. La table de transition qui détermine l'attribut (le texte) d'un terminal, doit pouvoir identifier ceux qui se terminent par l'opérateur de concaténation "^" lorsque suivi d'une continuation. Dans ce cas, le

symbole suivant (la continuation) a un deuxième attribut, "concat", qui prend la valeur prédéfinie "TRUE". Dans le cas contraire, l'attribut "concat" de la continuation a la valeur "FALSE". Lorsqu'un texte se termine par l'opérateur de concaténation et qu'il est suivi par une continuation, l'attribut (texte) de ce symbole est une copie du texte de la ligne, le caractère de concaténation en moins.

Voici le programme LEX qui nous permet de reconnaître les terminaux du PCS:

```
espace      = " "
fin_ligne   = "\n"
autre       = [\000-\377]

%%

"A"     espace          { terminal = AFFEC;     lexswitch(table_2); }
"+"     espace          { terminal = CONTI;     lexswitch(table_2); }
"I"     espace          { terminal = DSI;       lexswitch(table_2); }
"S"     espace          { terminal = DSISI;     lexswitch(table_2); }
"X"     espace          { terminal = DCS;       lexswitch(table_2); }
"D"     espace          { terminal = DRAF;      lexswitch(table_2); }
"T"     espace          { terminal = TITRE;     lexswitch(table_2); }
"C"     espace          { terminal = COMNA;     lexswitch(table_2); }
"<"     espace          { terminal = TELQ;      lexswitch(table_2); }
"F"                     { return FRAF;     }
"L"                     { return DSINO;    }
"E"                     { return FCOND;    }
"R"                     { return DREP;     }
"Z"                     { return FREP;     }
"Q"                     { return FCS;      }
fin_ligne
autre                   { return INVALIDE; }

%%
```

Fig. 6.1 Programme LEX pour reconnaître les symboles terminaux du PCS (Table #1, fichier: VMD_C_1.LXI)

```
fin_ligne    = "\n"
texte        = [\040-\377]*

%%

texte / fin_ligne "+"
        {
                /*
                    - Copier le texte dans l'attribut texte
                    - Vérifier la présence de l'opérateur de concaténation
                */
                lexswitch(table_1);
                return terminal;
        }

texte
        {
                /*
                    - Copier le texte dans l'attribut texte
                */
                lexswitch(table_1);
                return terminal;
        }

%%
```

Fig. 6.2 Programme LEX pour reconnaître les symboles terminaux du PCS (Table #2, fichier: VMD_C_2.LXI)

c) Réalisation de l'analyseur syntaxique

La troisième étape consiste à réaliser un analyseur syntaxique à l'aide des règles de grammaire du PCS. Nous utilisons l'outil YACC et le générateur d'analyseurs syntaxiques pour réaliser cet analyseur. Les règles de production de la grammaire du PCS, base de l'analyseur syntaxique, ont été présentées précédemment. Ces règles sont directement utilisables, il suffit de se conformer à la syntaxe de YACC en spécifiant le symbole de départ et en énumérant les symboles

terminaux et leur type. Voici le programme en langage YACC qui produit l'analyseur syntaxique de notre traducteur VMD (fichier VMD_C_Y.Y):

```
%{
    typedef struct {
            UNS8 concat;
            UNS8 texte[79]; } COMM_LEX_TXT;
}%
    %start algorithme
    %union { COMM_LEX_TXT comm_lex_txt; }

    %token <comm_lex_txt> AFFEC CONTI DSI DSISI DCS
    %token <comm_lex_txt> DRAF TITRE COMNA TELQ
    %token FRAF DSINO FCOND DREP FREP FCS INVALIDE
%%
    algorithme                      :  raffinement
    raffinement                     :  DRAF TITRE list_structure FRAF
    list_structure                  :
                                    |  list_structure structure
    structure                       :  séquentielle
                                    |  conditionnelle
                                    |  répétitive
    séquentielle                    :  élément_structural
    élément_structural              :  COMNA
                                    |  commentaire_op
                                    |  AFFEC continuée
                                    |  TELQ
    list_ele_structural             :
                                    |  list_ele_structural élément_structural
    commentaire_op                  :  raffinement
    continuée                       :
                                    |  continuée CONTI
    conditionnelle                  :  DSI continuée list_ele_structural
                                       list_alt_cond alternative FCOND
    list_alt_cond                   :
                                    |  list_alt_cond alternative_conditionnelle
    alternative_conditionnelle      :  DSISI continuée list_ele_structural
    alternative                     :  DSINO list_ele_structural
    répétitive                      :  DREP list_ele_répétitive FREP
    list_ele_répétitive             :  ele_répétitive
                                    |  list_ele_répétitive ele_répétitive
    ele_répétitive                  :  condition de sortie
                                    |  élément_structural
    condition_de_sortie             :  DCS continuée list_ele_structural FCS
```

Fig. 6.3 Programme YACC pour reconnaître le PCS

d) Identification des équivalents sémantiques

La quatrième étape consiste à identifier dans le langage cible, l'équivalent sémantique de chaque symbole terminal au sein des règles de production. On définit que les équivalents sémantiques sont composés entre autres, de chaînes de texte nommées comme suit:

```
deb_comop       =       "/* Raffinement # "
fin_comop       =       " */"
deb_comma       =       "/* "
fin_comma       =       " */"
deb_cond        =       "if ( "
fin_cond        =       " ) { "
deb_alt_cond    =       " } else if ( "
fin_alt_cond    =       " ) { "
deb_alt         =       " } else { "
fin_str_cond    =       " }"
deb_str_rep     =       "while(1) { "
deb_cs          =       "if ( "
fin_cs          =       " ) { "
fin_str_cs      =       " break; }"
fin_str_rep     =       " }"
```

Fig. 6.4 Chaînes de texte composant les équivalents sémantiques (VMD_C)

Voici les équivalents sémantiques qui observent la syntaxe du langage C. Les règles de production et les équivalents sémantiques des symboles terminaux sont présentés sous la forme d'un schéma de transformation:

```
raffinement          ->     DRAF
                                    { print deb_comop texte fin_comop }
                            TITRE
                                    { print deb_comma texte fin_comma }
                            list_structure FRAF
```

```
élément_structural      ->  COMNA
                                    { print deb_comna texte fin_comna }
                        |   commentaire_op
                        |   AFFEC
                                    { print texte }
                            continuée
                        |   TELQ
                                    { print texte }

continuée               ->  ε
                        |   continuée CONTI
                                    { print texte }

conditionnelle          ->  DSI
                                    { print deb_cond texte }
                            continuée
                                    { print fin_cond }
                            list_ele_structural
                            list_alt_cond alternative
                            FCOND
                                    { print fin_str_cond }

alternative_conditionnelle  ->  DSISI
                                    { print deb_alt_cond texte }
                            continuée
                                    { print fin_alt_cond }
                            list_ele_structural

alternative             ->  ε
                        |   DSINO
                                    { print deb_alt }
                            list_ele_structural

répétitive              ->  DREP
                                    { print deb_str_rep }
                            list_ele_répétitive
                            FREP
                                    { print fin_str_rep }

condition_de_sortie     ->  DCS
                                    { print deb_cs texte }
                            continuée
                                    { print fin_cs }
                            list_ele_structural
                            FCS
                                    { print fin_str_cs }
```

Fig. 6.5 Schéma de transformation avec équivalents sémantiques (VMD_C)

Le mot "texte" utilisé dans les actions du schéma de transformation représente le texte (attribut) du symbole terminal qui le précède. Seules les règles de production qui ont des actions (équivalents sémantiques) ont été représentées ici.

e) Établissement des actions de traduction

La cinquième étape consiste à établir les actions à effectuer pour traduire les règles de production dans leur équivalent en langage cible en se servant des attributs synthétisés des symboles terminaux et non-terminaux. Dans l'étape précédente, nous avons associé aux symboles terminaux des textes de transformation. Dans cette étape, il s'agit de raffiner ces textes en y ajoutant des directives. Ces directives permettent au code source produit de se présenter dans une forme acceptable par rapport à l'indentation et aux changements de ligne. Pour y arriver, nous allons nous doter d'une fonction qui accepte un nombre variable de paramètres où chaque paramètre est une chaîne de texte. L'implantation d'une telle fonction implique que le dernier paramètre soit spécial, il indique la fin de la liste des paramètres, nous utilisons la constante "NULL" pour ce dernier paramètre.

Les directives sont spécifées à l'aide de caractères spéciaux contenus dans les chaînes de texte. Voici les caractères spéciaux et leurs définitions reconnus par la fonction EMIT:

Caractère	Définition
'\n'	indique un changement de ligne;
'\b'	diminue l'indentation d'un pas pour la ligne courante si le curseur est placé avant la colonne d'indentation ou pour la ligne suivante si le curseur est positionné après la colonne d'indentation;
'\t'	augmente l'indentation d'un pas pour la ligne courante si le curseur est avant la colonne d'indentation ou pour la ligne suivante dans le cas contraire;
'\020'	force l'écriture du texte en début de ligne (en première colonne), il faut qu'aucun caractère n'est été écrit depuis le dernier '\n';
'\021'	indique qu'il faut se déplacer à la colonne spécifiée par l'indentation pour poursuivre l'écriture du texte.

Fig. 6.6 Caractères spéciaux reconnus dans les chaînes de texte

Les commandes '\020' et '\021' sont nécessaires entre autres, en présence d'étiquettes.

Nous allons réécrire les équivalents sémantiques et toutes les règles de production selon la syntaxe de YACC en tenant compte de tous les attributs des symboles terminaux et en incluant des directives de formattage de la fonction EMIT.

```
%{
#include "vmd_c_h.h"

UNS8 * deb_comop              =     "/* Raffinement # " ;
UNS8 * fin_comop              =     " */\n" ;
UNS8 * deb_comna              =     "/* " ;
UNS8 * fin_comna              =     " */\n" ;
UNS8 * deb_cond               =     "if ( \t" ;
UNS8 * fin_cond               =     " )\n{\t\n" ;
UNS8 * deb_alt_cond           =     "\b}\n\belse if ( \t" ;
UNS8 * fin_alt_cond           =     " )\n{\t\n" ;
UNS8 * deb_alt                =     "\b}\n\belse\n\t{\t\n" ;
UNS8 * fin_str_cond           =     "\b}\n\b";
UNS8 * deb_str_rep            =     "while(1)\n\t{\t\n" ;
UNS8 * deb_cs                 =     "if ( \t" ;
UNS8 * fin_cs                 =     " )\n{\t\n" ;
UNS8 * fin_str_cs             =     "break;\n\b}\n\b" ;
UNS8 * fin_str_rep            =     "\b}\n\b";
UNS8 * newline                =     "\n";
UNS8 * deb_ligne              =     "\020";
%}

%start programme
%union
{
 COMM_LEX_TXT                 comm_lex_txt;
}
%token <comm_lex_txt> AFFEC CONTI DSI DSISI DCS
%token <comm_lex_txt> DRAF TITRE COMNA TELQ
%token FRAF DSINO FCOND DREP FREP FCS
%token INVALIDE

%%

programme                     :
                              |     raffinement

raffinement                   :     DRAF TITRE
                                    {
                                        emit(deb_comop,$1.texte,fin_comop,NULL);
                                        emit(deb_comna,$2.texte,fin_comna,NULL);
                                    }
                                    list_structure FRAF

list_structure                :
                              |     list_structure structure

structure                     :     séquentielle
                              |     conditionnelle
                              |     répétitive

séquentielle                  :     élément_structural
```

```
élément_structural      :   COMNA
                            {
                                emit(deb_comna,$1.texte,fin_comna,NULL);
                            }
                        |   commentaire_op
                        |   AFFEC
                            {
                                emit($1.texte,NULL);
                            }
                            continuée
                            {
                                emit(newline,NULL);
                            }
                        |   TELQ
                            {
                                emit(deb_ligne,$1.texte,newline,NULL);
                            }

list_ele_structural     :
                        |   list_ele_structural élément_structural

commentaire_op          :   raffinement

continuée               :
                        |   continuée CONTI
                            {
                                if($2.concat==FALSE) emit(newline,NULL);
                                emit($2.texte,NULL);
                            }

conditionnelle          :   DSI
                            {
                                emit(deb_cond,$1.texte,NULL);
                            }
                            continuée
                            {
                                emit(fin_cond,NULL);
                            }
                            list_ele_structural
                            list_alt_cond alternative FCOND
                            {
                                emit(fin_str_cond,NULL);
                            }

list_alt_cond           :
                        |   list_alt_cond alternative_conditionnelle

alternative_conditionnelle  :   DSISI
                                {
                                    emit(deb_alt_cond,$1.texte,NULL);
                                }
                                continuée
                                {
                                    emit(fin_alt_cond,NULL);
                                }
                                list_ele_structural
```

```
alternative             :
                        |   DSINO
                            {

                            }
                        list_ele_structural

répétitive              :   DREP
                            {

                            }
                        list_ele_répétitive FREP
                            {
                                emit(fin_str_rep,NULL);
                            }

list_ele_répétitive     :   ele_répétitive
                        |   list_ele_répétitive ele_répétitive

ele_répétitive          :   condition_sortie
                        |   élément_structural

condition_sortie        :   DCS
                            {
                                emit(deb_cs,$1.texte,NULL);
                            }
                        continuée
                            {
                                emit(fin_cs,NULL);
                            }
                        list_ele_structural FCS
                            {
                                emit(fin_str_cs,NULL);
                            }

%%
```

Fig. 6.7 Analyseur syntaxique en langage YACC du traducteur C direct

f) Intégration des composantes

La sixième et dernière étape consiste à intégrer l'analyseur lexical et syntaxique au traducteur. Afin que le traducteur soit fonctionnel, il faut intégrer: l'analyseur lexical, l'analyseur syntaxique, le sous-programme "emit", un sous-programme de lecture de PCS ("lexgetc"), un sous-programme de traitement d'erreurs de syntaxe pour YACC ("yyerror") et le programme principal qui ouvre les fichiers et appelle l'analyseur syntaxique.

2. Traducteur Ada direct

Le traducteur Ada direct traduit un programme PCS contenu dans un fichier dont le nom est de la forme "programme.pcs" en un programme qui suit la syntaxe du langage Ada contenu dans un fichier dont le nom est de la forme "programme.pkg".

Le traducteur Ada direct est très semblable au traducteur C direct. Pour la plupart des étapes de la VMD, les résultats sont les mêmes pour le traducteur Ada direct. La composante du traducteur Ada qui s'occupe de la reconnaissance (identification des terminaux, réalisation de l'analyseur lexical, réalisation de l'analyseur syntaxique) est identique à celle du langage C. Pour le traducteur Ada direct, nous présentons les étapes de construction dont les résultats sont différents du langage C.

a) Identification des équivalents sémantiques

La principale différence entre un traducteur C direct et un traducteur Ada direct se situe au niveau des chaînes de texte qui composent les équivalents sémantiques. Nous retrouvons dans le langage Ada les mêmes chaînes de texte, mais avec un contenu différent:

```
deb_comop       =       "-- Raffinement # "
fin_comop       =       ""
deb_comna       =       "-- "
fin_comna       =       ""
deb_cond        =       "if ( "
fin_cond        =       " ) then "
deb_alt_cond    =       "elsif ( "
fin_alt_cond    =       " ) then "
deb_alt         =       "else "
fin_str_cond    =       "end if ;"
deb_str_rep     =       "loop"
deb_cs          =       "if ( "
fin_cs          =       " ) then "
fin_str_cs      =       "exit ; end if ;"
fin_str_rep     =       "end loop ;"
```

Fig. 6.8 Chaînes de texte composant les équivalents sémantiques (VMD_D)

Le schéma de transformation qui présente les règles de production et les équivalents sémantiques des symboles terminaux est le même que pour le traducteur C direct.

b) Établissement des actions de traduction

Les actions de traduction pour le traducteur Ada direct sont les mêmes que celles du traducteur C direct à l'exception du contenu des chaînes de texte qui composent les équivalents sémantiques. Voici les chaînes de texte qui incluent les directives de formatage selon la syntaxe de YACC:

```
UNS8 * deb_comop        =       "-- Raffinement # " ;
UNS8 * fin_comop        =       "\n" ;
UNS8 * deb_comma        =       "-- " ;
UNS8 * fin_comma        =       "\n" ;
UNS8 * deb_cond         =       "if ( \t" ;
UNS8 * fin_cond         =       " ) then\n" ;
UNS8 * deb_alt_cond     =       "\belsif ( \t" ;
UNS8 * fin_alt_cond     =       " ) then\n" ;
UNS8 * deb_alt          =       "\belse\n\t" ;
UNS8 * fin_str_cond     =       "\bend if ;\n";
UNS8 * deb_str_rep      =       "loop\n\t" ;
UNS8 * deb_cs           =       "if ( \t" ;
UNS8 * fin_cs           =       " ) then\n" ;
UNS8 * fin_str_cs       =       "exit ;\n\bend if ;\n" ;
UNS8 * fin_str_rep      =       "\bend loop ;\n";
UNS8 * newline          =       "\n";
UNS8 * deb_ligne        =       "\020";
```

Fig. 6.9 Chaînes de texte des équivalents sémantiques avec directives de formatage (VMD_D)

L'intégration des composantes se fait comme le traducteur C direct.

3. Traducteur assembleur 6502 direct

Le traducteur assembleur 6502 traduit un programme PCS contenu dans un fichier dont le nom est de la forme "programme.pcs" en un programme qui suit la syntaxe du langage assembleur 6502 contenu dans un fichier dont le nom est de la forme "programme.asm".

Il n'existe pas, à proprement dit, de standard pour la forme que doit prendre un programme assembleur 6502. Pour être le plus général possible et pour être utilisable avec la plupart des assembleurs disponibles sur le marché, nous convenons que:

- les étiquettes, lorsqu'elles sont présentes, se situent en début de ligne;
- les instructions ne se retrouvent jamais en début de ligne, elles sont toujours précédées d'au moins une espace;
- les étiquettes sont symboliques et commencent par une lettre de a-z ou A-Z, elles peuvent contenir des chiffres.

Le traducteur assembleur 6502 direct est semblable au traducteur C direct et au traducteur Ada direct. La composante du traducteur assembleur 6502 qui s'occupe de la reconnaissance (identification des terminaux, réalisation de l'analyseur lexical, réalisation de l'analyseur syntaxique) est identique à celle du langage C. Pour le traducteur assembleur 6502 direct, nous présentons les étapes de construction dont le résultat est différent du langage C.

a) Identification des équivalents sémantiques

Les équivalents sémantiques doivent tenir compte du fait qu'en assembleur, l'usage d'étiquettes est requis, car il n'existe pas de structure conditionnelle ni de structure répétitive. Ces dernières doivent être construites à l'aide d'instructions de test et de branchement.

Nous allons nous doter d'une pile contenant les étiquettes. Nous avons besoin d'une fonction qui crée une étiquette ("makelab") et de trois fonctions de gestion de la pile, soit "push" pour mettre l'étiquette courante ("label") dans la pile, "pop" pour retirer l'étiquette du dessus de la pile et la mettre dans l'étiquette courante et "view" pour voir un élément de la pile. Les fonctions de gestion de la pile correspondent à une vue différente des fonctions du type abstrait de données de haut niveau: pile. Elles sont réalisées sous forme de "MACRO".

Voici les équivalents sémantiques qui observent la syntaxe du langage assembleur 6502. Les règles de production et les équivalents sémantiques des symboles terminaux sont présentés sous la forme d'un schéma de transformation:

```
raffinement              ->   DRAF
                                        { print ";Raffinement # " texte }
                              TITRE
                                        { print "; " texte }
                              list_structure FRAF

élément_structural       ->   COMNA
                                        { print "; " texte }
                         |    commentaire_op
                         |    AFFEC
                                        { print texte }
                              continuée
                         |    TELQ
                                        { print texte }
```

```
continuée                  ->   ε
                           |    continuée CONTI
                                        { print texte }

conditionnelle             ->   DSI
                                        { print texte }
                                continuée
                                     {
                                  / *Créer une étiquette de fin de condition */
                                        makelab
                                  / *Conserver cette étiquette */
                                        push
                                  / *Créer une étiquette pour l'alternative */
                                        makelab
                                        push
                                  / *Sauter à la partie vraie de la condition */
                                        print " *+5"
                                  / *Sauter à l'alternative */
                                        print "jmp " label
                                     }
                                list_ele_structural
                                list_alt_cond alternative
                                FCOND
                                     {
                                  / *Récupérer l'étiquette alternative */
                                        pop
                                        print label
                                  / *Récupérer l'étiquette de fin de str. */
                                        pop
                                        print label
                                     }

alternative_conditionnelle ->   DSISI
                                     {
                                  / *Sauter à la fin de la structure */
                                        print "jmp " view(2)
                                  / *Récupérer l'étiquette alternative */
                                        pop
                                        print label texte
                                     }
                                continuée
                                     {
                                  / *Sauter à la partie vraie de l'alt. */
                                        print " *+5"
                                  / *Créer une étiquette pour */
                                  / *la prochaine alternative */
                                        makelab
                                        push
                                  / *Sauter à la prochaine alternative */
                                        print "jmp " label
                                     }
                                list_ele_structural
```

```
alternative          ->   €
                     |    DSINO
                               {
                          /* Sauter à la fin de la structure */
                                    print "jmp " view(2)
                          /* Récupérer l'étiquette alternative */
                                    pop
                                    print label
                          /* Créer une étiquette pour ajuster la pile*/
                                    makelab
                                    push
                               }
                          list_ele_structural

répétitive           ->   DREP
                               {
                          /* Créer une étiquette de fin de répétition */
                                    makelab
                                    push
                          /* Créer une étiquette de début de rép. */
                                    makelab
                                    push
                          /* Sortir l'étiquette de début de boucle */
                                    print label
                               }
                          list_ele_répétitive
                          FREP
                               {
                          /* Récupérer l'étiquette de début de rép. */
                                    pop
                          /* Sauter au début de la boucle */
                                    print "jmp " label
                          /* Récupérer l'étiquette de fin de rép. */
                                    pop
                                    print label
                               }

condition_de_sortie  ->   DCS
                               { print texte }
                          continuée
                               {
                          /* Créer une étiquette de fin de condition */
                                    makelab
                                    push
                          /* Sauter à la partie vraie de la condition */
                                    print " *+5"
                          /* Sauter à la fin de la cond. de sortie */
                                    print "jmp " label
                               }
```

```
list_ele_structural
FCS
    {
  / *Sauter à la fin de la répétition */
            print "jmp " view(3)
  / *Récupérer l'étiquette de fin de cond. */
            pop
            print label
    }
```

Fig. 6.10 Schéma de transformation avec équivalents sémantiques (VMD_A)

Le mot "texte" utilisé dans les actions du schéma de transformation représente le texte (attribut) du symbole terminal qui le précède. Seules les règles de production qui ont des actions (équivalents sémantiques) ont été représentées ici.

b) Établissement des actions de traduction

Pour des raisons esthétiques nous introduisons une nouvelle variable ("label_pending"). La variable est mise à "TRUE" quand une étiquette a été produite dans le fichier ".asm" et elle est remise à "FALSE" lorsque l'on peut sortir une étiquette sans changer de ligne au préalable.

L'intégration des composantes se fait comme pour le traducteur C direct.

B. Traducteurs construits selon la variante complète

Dans cette section, nous réalisons des traducteurs de PCS selon la variante complète de la méthode de construction de traducteurs. Les mêmes langages cibles, soit le langage C, le langage Ada et le langage assembleur 6502, ont été repris pour mettre en évidence les distinctions entre les variantes de la méthode. Plusieurs étapes de construction, réalisées selon la variante complète, donnent les mêmes résultats que pour la variante directe. Ces résultats ne sont pas repris dans cette section.

1. Traducteur C complet

Nous allons suivre les étapes de la variante complète de la méthode de construction de traducteurs en les appliquant au langage C. Les résultats des étapes d'identification des symboles terminaux du PCS et de réalisation de l'analyseur lexical, faisant partie de la reconnaissance, sont identiques aux résultats de ces mêmes étapes selon la variante directe.

a) Réalisation de l'analyseur syntaxique

La réalisation de l'analyseur syntaxique est faite avec l'outil PREP pour permettre aux étapes subséquentes de véhiculer plus aisément les attributs hérités et synthétisés des symboles non-terminaux. Cette étape doit aussi charger en mémoire le programme PCS. À l'aide d'une liste (type abstrait de données de haut niveau) nous chargeons le fichier PCS en mémoire. Chacun des éléments de la liste

représente une ligne de fichier PCS, c'est-à-dire un symbole terminal. Ces symboles ont trois attributs déclarés comme ceci:

```
typedef struct {
            INT16 statut;
            INT16 règle;
            UNS8 *texte_ptr;
        } SYNTAX;
```

Fig. 6.11 Composition de la structure attribut

Le "statut", au départ, ne sert qu'aux continuations pour indiquer si elles sont jointes, la "règle" contient le nom du symbole terminal de la ligne et le "texte_ptr" est un pointeur sur le texte de la ligne. Ces trois constituants sont regroupés dans une structure accessible par une variable de type "SYNTAX".

Deux fonctions de base, implantées sous forme de "MACRO", "set_symb" et "add_symb", sont nécessaires pour charger le fichier PCS en mémoire. La fonction "set_symb" demande trois paramètres, le premier contient le "statut", le deuxième est associé à la "règle" et le troisième au "texte_ptr".

Voici une section du programme PREP qui réalise l'analyseur syntaxique et qui charge un PCS dans une liste.

```
%%

raffinement              :    draf titre list_structure fraf

élément_structural       :    comna
                         |    commentaire_op
                         |    affec continuée
                         |    telq

continuée                :
                         |    continuée conti

affec                    :    AFFEC
                                  { set_symb(0,AFFEC,$1.TEXTE);
                                    add_symb; }

comna                    :    COMNA
                                  { set_symb(0,COMNA,$1.TEXTE):
                                    add_symb; }

conti                    :    CONTI
                                  { set_symb($1.concat?LIGNE_CONCAT:0,CONTI,
                                          $1.TEXTE);
                                    add_symb; }

draf                     :    DRAF
                                  { set_symb(0,DRAF,$1.TEXTE);
                                    add_symb; }

fraf                     :    FRAF
                                  { set_symb(0,FRAF,NULL);
                                    add_symb; }

telq                     :    TELQ
                                  { set_symb(0,TELQ,$1.texte);
                                    add_symb; }

titre                    :    TITRE
                                  { set_symb(0,TITRE,$1.texte);
                                    add_symb; }
%%
```

Fig. 6.12 Section du programme PREP qui réalise l'analyseur syntaxique (VMC_C)

b) Identification des structures complètes du PCS

L'information nécessaire pour reconnaître les structures complètes du PCS se présente sous deux aspects:

- du point de vue de la structure complète;
- du point de vue de l'élément de structure complète.

Nous devons connaître la composition d'une structure complète lorsqu'on la rencontre. Lorsqu'on traite ses composantes nous devons connaître leurs implications. L'information sur la structure complète est contenue dans le symbole qui débute cette structure, soit "DSI" pour une conditionnelle et "DREP" pour une répétitive. La "DCS" est considérée comme un élément d'une répétitive et non pas comme une structure complète. Du point de vue de la structure complète, les informations conservées pour une conditionnelle sont :

- présence de branches SINON-SI (COND_SINON_SI)
- présence d'une branche SINON (COND_SINON)

Les informations conservées pour une structure répétitive sont:

- présence de plusieurs CS (PLUSIEURS_CS)
- présence d'une seule CS (UNE_CS)
- présence d'une CS en début de structure (CS_AU_DÉBUT)
- présence d'une CS en fin de structure (CS_À_LA_FIN)

- présence d'un énoncé exécutable dans la CS (CS_UNE_ACTION)
- présence de plusieurs énoncés exécutables dans la CS (CS_PLUSIEURS_ACTIONS)

Les informations "CS_UNE_ACTION" et "CS_PLUSIEURS_ACTIONS" se rapportent à la CS placée au début de la structure répétitive si elle est présente, sinon, à la CS placée à la fin, sinon, à la première CS rencontrée dans la structure, ou sinon, lorsqu'il n'y a pas de CS, l'information ne s'applique tout simplement pas.

Du point de vue des éléments ou branches d'une structure, des informations doivent être construites et conservées. Pour les branches "DSI", "DSISI", "DSINO", "DREP" et "DCS" deux informations sont reconnues:

- présence de plusieurs énoncés exécutables (PLUSIEURS_ACTIONS)
- présence d'un seul énoncé exécutable (UNE_ACTION)

Ce que nous entendons par un énoncé exécutable c'est un symbole "AFFEC" ou "TELQ" contenu dans une branche. Ce que nous entendons par plusieurs énoncés exécutables c'est une ou plusieurs conditionnelles ou répétitives, et/ou une série d'"AFFEC" ou de "TELQ".

Pour les branches "DSI" et "DSISI", deux informations sont reconnues:

- présence d'un "SINON-SI" qui suit (SINON_SI_QUI_SUIT)
- présence d'un "SINON" qui suit (SINON_QUI_SUIT)

Pour une "DCS" deux informations sont reconnues:

- la CS est en début de répétitive (DÉBUT_DE_REP)
- la CS est en fin de répétitive (FIN_DE_REP)

Il est possible que certaines combinaisons de ces informations n'aient pas de sens et il est aussi possible que dans certains cas, il y ait des dépendances entre les informations. Cette redondance est utile pour rendre la transcription aussi indépendante que possible du contexte.

c) Reconnaissance des stuctures complètes

Nous n'allons pas décrire en détail les actions à effectuer pour reconnaître les structures complètes. Nous mentionnons quand même que les informations recueillies à l'étape précédente sont conservées dans le "statut" du symbole. La reconnaissance de ces informations se fait au moment du chargement du programme PCS dans la liste.

d) Identification des équivalents sémantiques

De la même façon que pour la VMD, les équivalents sémantiques des structures complètes sont composés des chaînes de texte suivantes:

```
deb_comop       =       "/* Raffinement # "
fin_comop       =       " */"
deb_comma       =       "/* "
fin_comma       =       " */"
deb_cond        =       "if ( "
fin_cond        =       " ) { "
deb_alt_cond    =       " } else if ( "
fin_alt_cond    =       " ) { "
deb_alt         =       " } else { "
fin_str_cond    =       " }"
deb_str_rep     =       "for(;;) { "
fin_str_rep     =       " }"
csdeb_deb_rep1  =       "while(!( "
csdeb_deb_rep2  =       " )) { "
csfin_deb_rep   =       "do { "
csfin_fin_rep1  =       " } while (!( "
csfin_fin_rep2  =       " ));"
a_deb_cs        =       "if ( "
a_fin_cs        =       " ) { "
a_fin_str_cs    =       " break; }"
deb_cs          =       "if ( "
fin_str_cs      =       " ) break;"
```

Fig. 6.13 Chaînes de texte composant les équivalents sémantiques (VMC_C)

Nous ne présentons pas le schéma de transformation complet. Nous allons plutôt décrire les conditions dans lesquelles les chaînes de texte définies plus haut sont utilisées.

Pour une répétitive.

A) En présence d'une condition de sortie placée au début et sans action

-> au début ne rien faire, le cas sera traité à la première CS,

-> à la fin utiliser "fin_str_rep".

B) En présence d'une condition de sortie placée à la fin et sans action

-> au début utiliser "csfin_deb_rep",

-> à la fin ne rien faire, le cas sera traité à la dernière CS.

C) Autrement

-> au début utiliser "deb_str_rep",

-> à la fin utiliser "fin_str_rep".

Pour une CS.

A) Lorqu'elle est présente en début de répétitive de type A)

-> utiliser "csdeb_deb_rep1" et "csdeb_deb_rep2".

B) Lorqu'elle est présente en fin de répétitive de type B)

-> utiliser "csfin_fin_rep1" et "csfin_fin_rep2".

C) Autrement, et la CS a une ou plusieurs actions associées

-> utiliser "a_deb_cs", "a_fin_cs" et "a_fin_str_cs".

D) Autrement (sans action)

-> utiliser "deb_cs" et "fin_str_cs".

Le comportement des séquentielles et des conditionnelles est équivalent à celui de la VMD.

e) Établissement des actions de traduction

La technique utilisée pour parcourir la liste des symboles et traduire ces symboles est basée sur ce qui a été présenté à la section "Parcours d'un arbre syntaxique" du chapitre sur la "Théorie des langages". Il s'agit d'un ensemble de fonctions récursives dont les noms sont semblables aux noms des symboles non-terminaux, qui réalisent la transcription. Le corps d'une fonction peut se baser sur le fait que la représentation interne du programme à traduire est exempt d'erreur de syntaxe, ainsi que sur le fait que les éléments de la liste contiennent le nom des symboles terminaux qu'ils représentent. Les fonctions déplacent le curseur de la liste en fonction de ce qu'elles ont à traiter et laissent la liste à la position suivante du dernier élément traité. À titre d'exemple voici la fonction "continuée":

```
void continuée ()
{
SYNTAX * elem;

    - Prendre l'information de l'élément courant
    elem = GET_ELE;
    *   elem->regle != CONTI
        *
    - Traiter le texte de la continuation
        ! ( elem->statut & LIGNE_CONCAT )
        emit(newline,NULL);
    emit(elem->texte_ptr,NULL);
    NEXT;
}
```

Fig. 6.14 Fonction "continuée" servant à traduire une continuation (VMC_C)

L'intégration des composantes de l'analyseur se fait comme pour la VMD.

2. Traducteur Ada complet

Entre le traducteur C complet et le traducteur Ada complet, il existe une seule différence. Cette différence se situe au niveau de la constitution des chaînes de texte qui composent les équivalents sémantiques des structures complètes.

a) Identification des équivalents sémantiques

En langage Ada, par opposition au langage C, il n'existe pas d'énoncé spécifique qui traduit une structure répétitive composée d'une condition de sortie sans action placée à la fin. Ce cas est donc traité par la structure répétitive générale. Voici la composition des chaînes de texte qui composent les équivalents sémantiques pour le langage Ada:

```
deb_comop       =       "-- Raffinement # "
fin_comop       =       ""
deb_comma       =       "-- "
fin_comma       =       ""
deb_cond        =       "if ( "
fin_cond        =       " ) then "
deb_alt_cond    =       "elsif ( "
fin_alt_cond    =       " ) then "
deb_alt         =       "else "
fin_str_cond    =       "end if ;"
deb_str_rep     =       "loop"
fin_str_rep     =       "end loop ;"
csdeb_deb_rep1  =       "while (not( "
csdeb_deb_rep2  =       " )) loop"
a_deb_cs        =       "if ( "
a_fin_cs        =       " ) then "
a_fin_str_cs    =       "exit ; end if ;"
deb_cs          =       "exit when ( "
fin_str_cs      =       " );"
```

Fig. 6.15 Chaînes de texte composant les équivalents sémantiques (VMC_D)

3. Traducteur assembleur 6502 complet

La "reconnaissance" des traducteurs complets consiste à reconnaître les structures complètes du PCS. Pour le traducteur assembleur 6502 complet, cette information est interprétée différemment qu'en langage C ou Ada. Dans ce langage, il n'existe pas de structure conditionnelle ou répétitive, c'est pourquoi les informations recueillies vont servir à améliorer la forme ou la présentation du code source traduit.

Le traducteur assembleur 6502 complet présente la même architecture que les autres traducteurs complets, c'est-à-dire que la transcription, fonction du langage cible, est réalisée à l'aide du même ensemble de fonctions récursives, mais appliquée au langage assembleur 6502.

Pour ce traducteur, on reprend les actions définies pour la VMD et on les utilise dans un contexte de VMC, où on connaît d'avance la composition de la structure en présence.

a) Identification des équivalents sémantiques

Le fait de reconnaître la composition des structures complètes permet un meilleur usage des étiquettes. Le code source traduit présente moins d'étiquettes pour les structures conditionnelles. Le traitement des branches "DSI" et "DSISI" vérifie si elles sont les dernières branches de la structure et dans ce cas n'engendre pas d'étiquette inutile. Le traitement des conditions de sortie "DSI" peut aussi profiter du fait qu'il sait si elles possèdent des actions à l'intérieur et produire le code en conséquence.

C. Traducteurs construits selon la variante ciblée

Contrairement aux autres variantes de la méthode de construction de traducteurs, celle-ci peut créer une infinité de traducteurs différents pour un même langage cible. Cela vient du fait qu'il y a de la latitude au niveau de la reconnaissance des multiples syntaxes d'un langage cible associés à la traduction d'une structure complète. Il s'agit de déterminer une façon de dissocier les différents équivalents sémantiques d'une même structure complète à l'aide des attributs ("texte_ptr") disponibles. Pour différencier les cas, on peut se baser sur la syntaxe spécifique de chaque possibilité ou proposer une nouvelle syntaxe particulière qui permettra de lever l'ambiguïté.

1. Traducteur C ciblé

Les résultats des six premières étapes de la réalisation d'un traducteur ciblé en langage C sont identiques aux résultats de ces mêmes étapes pour la VMC. Ces six étapes sont:

- l'identification des symboles terminaux du PCS et leurs attributs;
- la réalisation d'un analyseur lexical qui reconnaîtra tous ces terminaux;
- la réalisation d'un analyseur syntaxique qui charge un fichier PCS dans une liste en mémoire;
- l'identification des différentes structures complètes du PCS;

- la détermination des actions à effectuer pour reconnaître les structures complètes du PCS à l'intérieur de l'analyseur syntaxique et les attacher aux symboles terminaux chargés en mémoire;
- l'intégration des actions à l'analyseur syntaxique.

a) Identification des différentes syntaxes

L'étape suivante consiste à identifier dans le langage cible les différentes syntaxes qui peuvent être associées à une même structure complète. On se souvient que pour le langage C, la VMC ne nous permettait pas d'utiliser l'instruction "for", pour son usage habituel, afin de traduire une répétitive, ainsi que l'instruction "switch" pour traduire une conditionnelle. L'usage de ces deux instructions de structure est régi par une syntaxe particulière et spécifique. Dans une approche ciblée, il est possible de traduire une répétitive et une conditionnelle à l'aide de ces instructions, mais sous certaines conditions. L'approche ciblée nous permet aussi d'améliorer la traduction de certaines expressions de la forme "! (expression)" utilisées dans les conditions de sortie traduites par un "while(!(texte))". Les deux "! " qui se suivent peuvent être éliminés.

Ce sont ces trois caractéristiques ciblées que notre traducteur C ciblé nous permet d'utiliser lors de l'élaboration d'un programme PCS.

b) Choix des critères d'association

Il s'agit ici de déterminer les critères que le traducteur applique sur les attributs ("texte_ptr") afin de décider l'équivalent sémantique à utiliser lorsqu'il y a ambiguïté.

Une expression en langage C ne peut contenir de ";", on se sert de cette spécification pour reconnaître la boucle "for". L'instruction "for" est utilisée pour traduire une répétitive lorsqu'elle présente une condition de sortie située en début de structure et que son expression contient un ";".

Pour éliminer un double "!" lors de la traduction de certaines expressions de condition de sortie, le traducteur retire le "!" d'une expression si cette expression débute par l'expression régulière suivante: ([ ]*![ ]*(").

Le traducteur peut introduire des erreurs si le programmeur utilise cette forme d'expression sans précaution. Par exemple dans le cas de l'expression "! (valide) || erreur" qui serait traduite par "while (valide) || erreur". Dans ce cas le compilateur indiquera une erreur de syntaxe.

Pour l'instruction "switch", le traducteur reconnaît sa syntaxe si l'expression de la première branche conditionnelle débute par l'expression régulière : ([ ]*case[ ]). Le traducteur ajoute un "break;" à la fin de chaque branche, par conséquent, il n'est pas possible d'utiliser la forme, peu commune, suivante:

```
case 'A'
        premier_caractère = 'a';
case 'Z'
        dernier_caractère = 'z';
        break;
```

Fig. 6.16 Utilisation peu commune de l'instruction "switch"

Par contre, il est possible d'écrire ceci:

```
case '1':  case '2':  case '3':
symbole = CHIFFRE123;
```

Fig. 6.17 Exemple d'utilisation d'un "switch" en pseudocode schématique (VMI_C)

Il est aussi possible de regrouper dans des raffinements, les parties d'un "switch". Dans ce cas, il faut écrire:

```
switch (erreur)
{
[01] Traiter les messages d'erreur de 0-99
[02] Traiter les messages d'erreur de 100+
}
```

Fig. 6.18 Instruction "switch" traitée dans plus d'un raffinement (VMI_C)

Dans ce dernier cas, les accolades ({}) sont obligatoires, sinon il y aura une erreur de syntaxe à moins que ce code soit dans un autre "switch" et que les cas "erreurs 100+" aient un sens pour ce "switch".

c) Établissement des actions de traduction

Afin que le traducteur soit performant, les critères de différentiation des équivalents sémantiques faisant intervenir le contenu de l'attribut sont vérifiés seulement en présence des structures visées par cette différentiation.

2. Traducteur Ada ciblé

Le traducteur Ada ciblé est très semblable au traducteur C ciblé, il reconnaît la répétitive "for", la conditionnelle de type "case" et améliore la présentation des expressions débutant par "not" lorsque c'est possible. Les résultats des étapes de la construction de ce traducteur sont aussi inspirés des résultats de la construction du traducteur Ada complet. Nous explorons plus en détail les critères servant à reconnaître l'équivalent sémantique qui traduit le mieux la volonté du programmeur parmi les équivalents sémantiques qui peuvent être associés à une même structure complète.

a) Choix des critères d'association

La structure répétitive qui a une condition de sortie placée au début dont l'expression contient le texte " in ", est traduite à l'aide de l'instruction "for".

Lorsque l'expression de la première branche d'une structure conditionnelle débute par l'expression régulière suivante: ([ ]*when[ ]), le traducteur considère que la conditionnelle est un "case" et suit cette syntaxe pour traduire toute la structure.

Si le traducteur, lorsqu'il traduit une répétitive, choisit d'utiliser la chaîne "csdeb_deb_rep1_not" et que le texte de l'expression de la condition de sortie débute par ([ ]*not[ ]*"("), alors il utilise plutôt la chaîne "csdeb_deb_rep1" et il retire le "not" de l'expression.

Dans le contexte du Ada, il est aussi possible que le traducteur introduise des erreurs de syntaxe ou de logique si le programmeur ne tient pas compte de sa composante ciblée.

Comme le traducteur C, le traducteur Ada utilise les critères de différentiation seulement en présence des structures visées.

3. Traducteur assembleur 6502 ciblé

Un traducteur ciblé pour un langage qui est aussi près de son code que le langage assembleur est un atout qui rend le pseudocode schématique extrêmement intéressant.

a) Identification des différentes syntaxes

Pour un langage qui ne possède pas de structure, cette étape de la VMI concerne plutôt les différentes façons pour le traducteur de traduire une expression en fonction de l'endroit où elle est utilisée.

Dans certains cas, le traducteur a besoin de créer des artifices pour que le cheminement principal soit exécuté lorsque la condition est vraie. Un traducteur ciblé a la possibilité de choisir une instruction qui a le sens inverse pour éviter cet artifice. Voici un exemple:

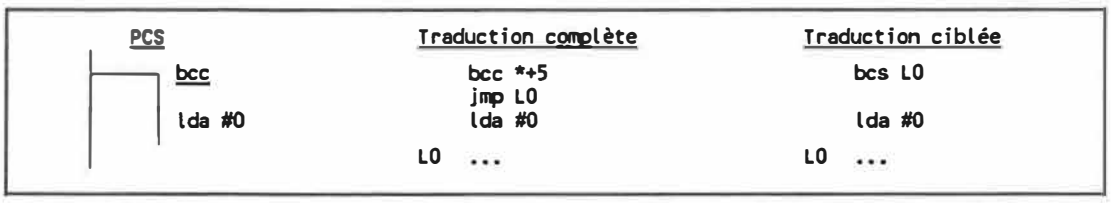


Fig. 6.19 Traduction complète et ciblée d'une conditionelle (VMI_A)

Le traducteur assembleur 6502 ciblé doit toujours tenter d'éviter les artifices. Par contre, dans ce langage, il devient impossible d'éviter ce manège dans le cas où

le branchement ordonne de faire un saut vers l'avant de plus de 127 octets ou vers l'arrière de moins de 128 octets.

b) Choix des critères d'association

Pour forcer le traducteur à utiliser un artifice, le programmeur montre son intention par l'intermédiaire de l'expression, en la débutant avec un "j" plutôt qu'un "b". Voici un exemple pour forcer un artifice:

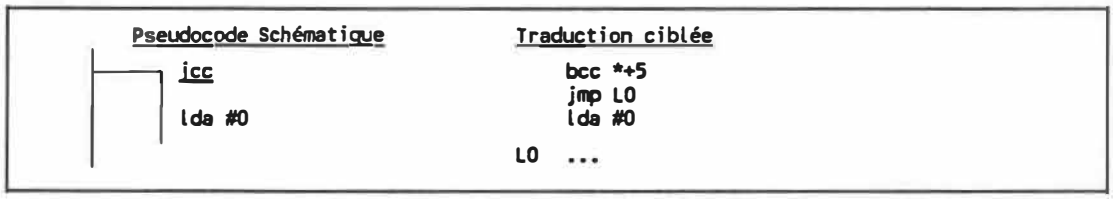


Fig. 6.20 Traduction complète et ciblée d'une conditionelle (VMI_A)

Le programmeur a la responsabilité de déterminer les endroits où les artifices sont nécessaires. Cette information lui sera transmise par son assembleur. Il faut porter une attention particulière aux modifications d'un logiciel , afin de remettre des "b" lorsque c'est à nouveau possible.

Comme le traducteur Ada et le traducteur C, le traducteur assembleur 6502 ciblé demande une plus grande implication du programmeur pour l'interprétation des expressions.

VII. EXPÉRIMENTATIONS

A. Présentation des résultats

1. Les traducteurs C

L'évaluation des performances des traducteurs en langage C est faite à l'aide du compilateur Microsoft C version 4.0 sur un COMPAQ DESKPRO 386/20 en disque mémoire («Ram Disk»). Voici les résultats d'évaluation des critères obtenus dans les conditions spécifiées.

Traduction	**Directe**	**Complète**	**Ciblée**
Les critères qualitatifs visés			
Indentation	oui	oui	oui
Structuration	oui	oui	oui
Disposition des commentaires	oui	oui	oui
Les critères qualitatifs comparatifs pour l'utilisateur			
Disponibilité des instructions de structure	non	non	oui
Les critères qualitatifs comparatifs pour le traducteur			
Usage d'énoncés appropriés	non	oui	oui
Présence de code superflu	oui	non	non

Tableau 7.1 Résultats qualitatifs pour les traducteurs de langage C

Traduction	Directe	Complète	Ciblée
Les critères quantitatifs associés à la traduction			
Le temps de traduction (sans écriture)			
séquentielle	ref	+1%	+1%
conditionnelle	ref	+3%	+3%
répétitive	ref	+6%	+6%
Le temps de traduction (avec écriture)			
séquentielle	ref	0%	0%
conditionnelle	ref	+3%	+3%
répétitive	ref	-10%	-10%
Espace mémoire de traduction[1]			
séquentielle	n/a	+164%	+164%
conditionnelle	n/a	+201%	+201%
répétitive	n/a	+268%	+268%
Les critères quantitatifs associés au code produit			
Le nombre de lignes du programme traduit			
séquentielle	ref	0%	0%
conditionnelle	ref	0%	0%
répétitive	ref	-39%	-39%
Le nombre de caractères du programme			
séquentielle	ref	0%	0%
conditionnelle	ref	0%	0%
répétitive	ref	-31%	-31%
Le temps de compilation			
séquentielle	ref	0%	0%
conditionnelle	ref	0%	0%
répétitive	ref	-28%	-28%
Le temps d'exécution			
séquentielle	ref	0%	0%
conditionnelle	ref	0%	0%
répétitive	ref	-9%	-9%

Tableau 7.2 Résultats quantitatifs pour les traducteurs de langage C

[1]Nombre d'octets de mémoire utilisé divisé par la dimension du fichier PCS à traduire.

Conditionnelle à 20 branches	**"if else"**	**"switch case"**
Les critères quantitatifs associés à la traduction		
Le temps de traduction (avec écriture)	ref	-13%
Espace mémoire de traduction	ref	0%
Les critères quantitatifs associés au code produit		
Le nombre de lignes du programme traduit	ref	-21%
Le nombre de caractères du programme	ref	-18%
Le temps de compilation	ref	-47%
Le temps d'exécution	ref	+520%
Conditionnelle à 250 branches	**"if else"**	**"switch case"**
Les critères quantitatifs associés à la traduction		
Le temps de traduction (avec écriture)	ref	-15%
Espace mémoire de traduction	ref	0%
Les critères quantitatifs associés au code produit		
Le nombre de lignes du programme traduit	ref	-24%
Le nombre de caractères du programme	ref	-22%
Le temps de compilation	ref	-19%
Le temps d'exécution	ref	-92%

Tableau 7.3 Résultats quantitatifs pour le code ciblé en langage C

On constate que le traducteur complet et le traducteur ciblé prennent plus de temps à traduire un programme que le direct si on ne considère pas l'écriture. Avec l'écriture, le ciblé prend moins de temps lorsque le code source traduit est plus petit. Globalement, le traducteur complet et le traducteur ciblé prendront plus ou moins de temps à traduire un programme tout dépendant de la composition en structures conditionnelles et répétitives qui permettent une forme condensée.

On constate aussi que le traducteur complet et le traducteur ciblé prennent de 2 à 4 fois plus de mémoire que la dimension du programme à traduire; tandis que l'utilisation de la mémoire pour le traducteur direct n'est pas fonction du programme à traduire. Les différences entre l'usage de la mémoire pour les séquentielles, les conditionnelles et les répétitives sont dûes à la constitution même de la grammaire. Les symboles "FCOND" pour la conditionnelle ainsi que "DREP", "FREP" et "FCS" pour la répétitive, occupent de l'espace dans un programme PCS, donc en mémoire, mais ne produisent pas nécessairement de code.

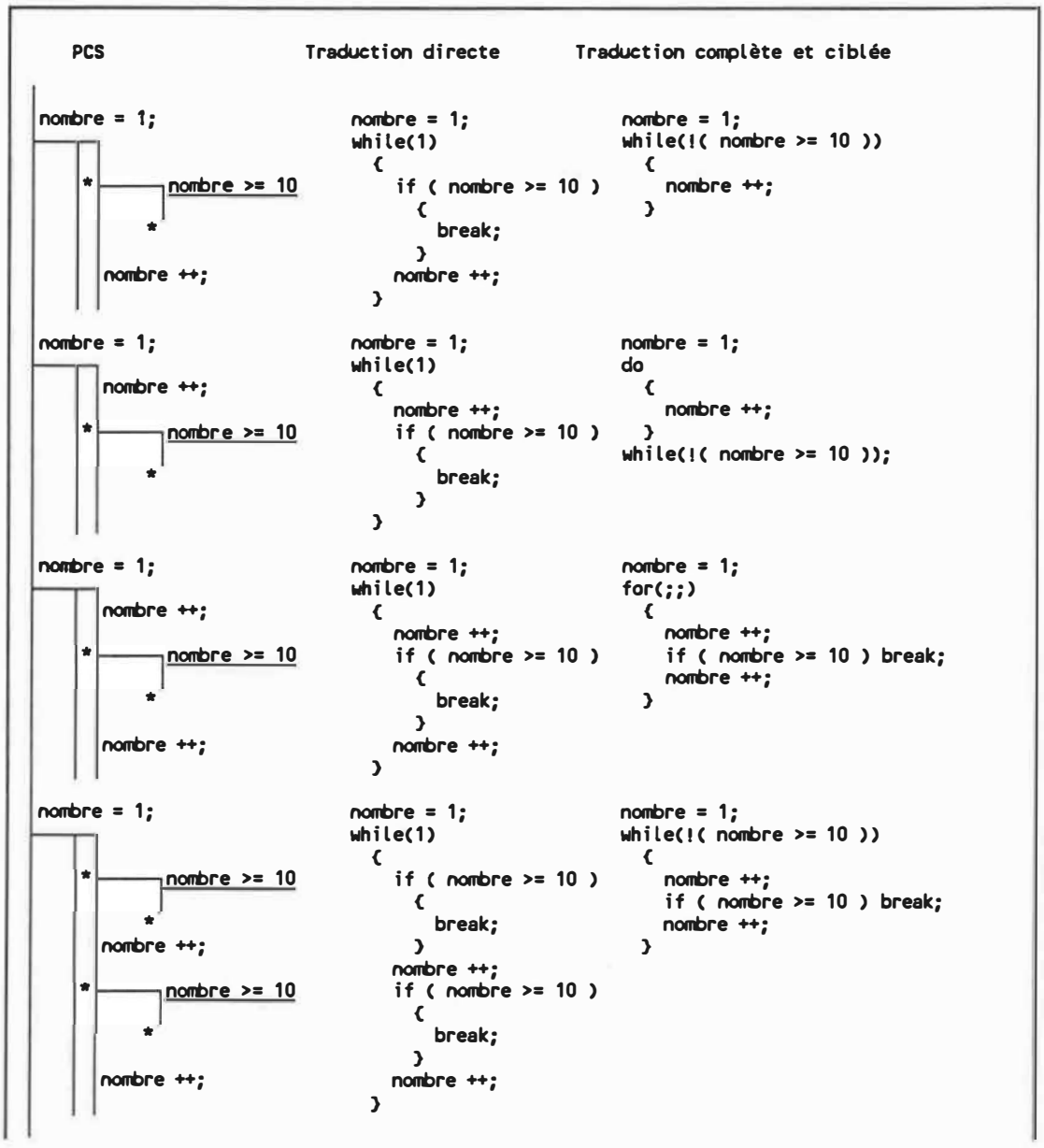
Le fait que l'exécution d'un programme utilisant des répétitives (pouvant être traduites par des formes condensées) prenne moins de temps que le même programme n'utilisant pas de forme condensée est, à notre avis, fonction des possibilités d'optimisation du compilateur.

On constate que le choix du "switch" pour traduire une conditionnelle ne donne pas nécessairement un programme plus rapide d'exécution.

Pour obtenir ces résultats, nous avons répété plusieurs fois une même section de code dans le programme à traduire. Le choix d'un gros fichier source nous a permis d'obtenir des temps suffisamment longs pour mesurer avec une marge d'erreur inférieure à 1% les critères en question.

Ces mesures quantitatives sont extrêmement liées à la forme du programme à traduire. C'est pourquoi, malgré une certaine exactitude, elles ne peuvent être considérées qu'à titre indicatif.

Voici une section du programme qui a servi afin d'obtenir les résultats d'évaluation des critères quantitatifs associés au code produit pour la répétitive.



PCS

```
nombre = 1;
*  nombre >= 10
   *
nombre ++;
```

```
nombre = 1;
*  nombre ++;
   nombre >= 10
   *
```

```
nombre = 1;
*  nombre ++;
   nombre >= 10
   *
nombre ++;
```

```
nombre = 1;
*  nombre >= 10
   *
nombre ++;
*  nombre >= 10
   *
nombre ++;
```

Traduction directe

```
nombre = 1;
while(1)
  {
    if ( nombre >= 10 )
      {
        break;
      }
    nombre ++;
  }
```

```
nombre = 1;
while(1)
  {
    nombre ++;
    if ( nombre >= 10 )
      {
        break;
      }
  }
```

```
nombre = 1;
while(1)
  {
    nombre ++;
    if ( nombre >= 10 )
      {
        break;
      }
    nombre ++;
  }
```

```
nombre = 1;
while(1)
  {
    if ( nombre >= 10 )
      {
        break;
      }
    nombre ++;
    if ( nombre >= 10 )
      {
        break;
      }
    nombre ++;
  }
```

Traduction complète et ciblée

```
nombre = 1;
while(!( nombre >= 10 ))
  {
    nombre ++;
  }
```

```
nombre = 1;
do
  {
    nombre ++;
  }
while(!( nombre >= 10 ));
```

```
nombre = 1;
for(;;)
  {
    nombre ++;
    if ( nombre >= 10 ) break;
    nombre ++;
  }
```

```
nombre = 1;
while(!( nombre >= 10 ))
  {
    nombre ++;
    if ( nombre >= 10 ) break;
    nombre ++;
  }
```

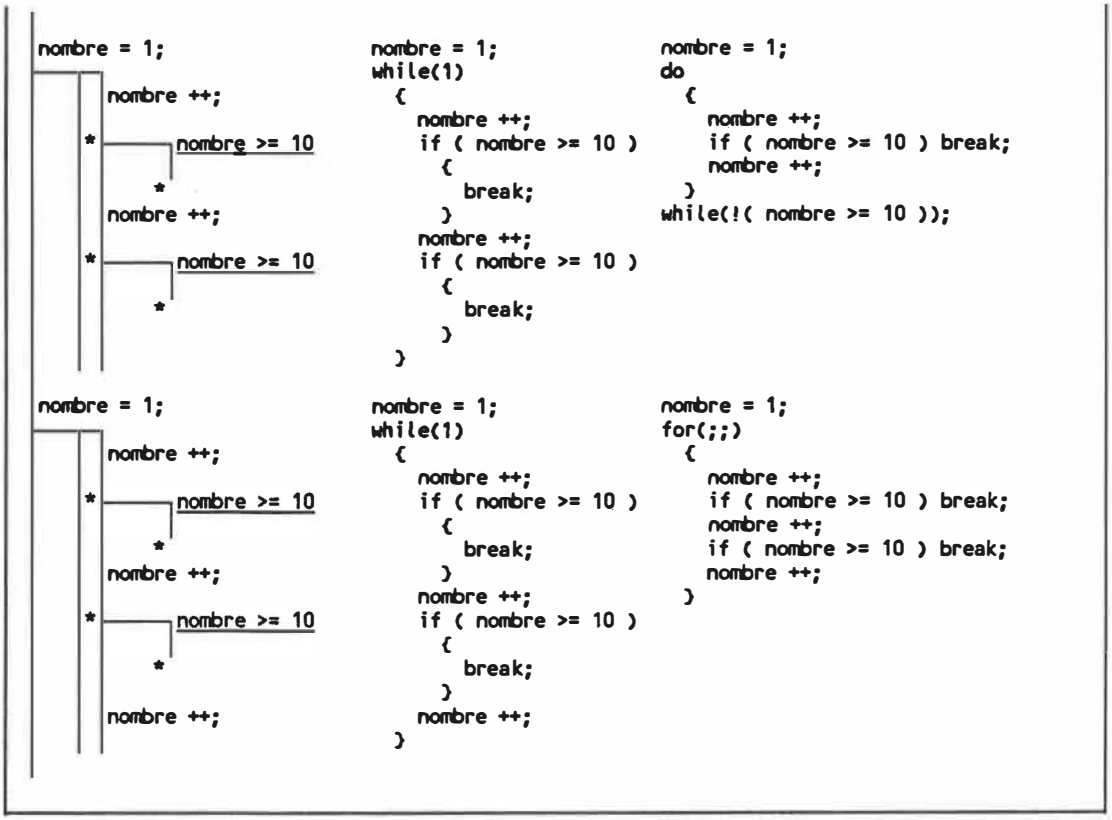


Fig. 7.1 Section de code servant à évaluer les critères quantitatifs en C.

La section de code est répétée 25 fois et elle est comprise dans une boucle de 10000 itérations. Les résultats obtenus pour le temps d'exécution de ce programme sont les suivants:

	Traduction directe	Traduction complète et ciblée
Valeur enregistrée	15930 ms	14500 ms
Valeur relative	ref	-9%

Tableau 7.4 Temps d'exécution du programme d'évaluation en langage C

Le temps d'exécution enregistré est obtenu en faisant la moyenne des temps mesurés pour l'exécution du programme à, au moins, cinq reprises. Dans ce cas-ci, la même valeur a été mesurée cinq fois. Les différences entre les temps observés sont de l'ordre d'une dizaine de millisecondes. La fluctuation est dûe au système d'exploitation.

L'annexe contient un programme traduit par les traducteurs direct, complet et ciblé en langage C et un programme conçu pour le traducteur ciblé. Les codes sources traduits démontrent bien les différences selon les approches utilisées.

Une autre façon de comparer les différents types de traducteurs C entre eux, consiste à identifier les énoncés de structure que le traducteur utilise, par rapport à tous les énoncés de structure que le langage C possède:

Énoncés de structure du C		Utilisés par le traducteur		
		direct	complet	ciblé
séquentielle:		oui	oui	oui
conditionnelle:	if, if else	oui	oui	oui
	switch case	non	non	oui
répétitive:	for	non	non	oui
	while	oui	oui	oui
	do while	non	oui	oui
	break	oui	oui	oui
	continue	non	non	non
		---	---	---
		50%	63%	88%

Tableau 7.5 Énoncés du langage utilisés par le traducteur C

On constate que le traducteur ciblé utilise presque tous les énoncés de structure du langage.

D'un autre point de vue on peut évaluer le langage C en fonction des structures du pseudocode schématique. Prenons un sous-ensemble des structures complètes du pseudocode schématique et regardons s'il existe, dans le langage C, des énoncés qui représentent directement ces structures. Ainsi on peut caractériser l'apport du pseudocode schématique pour le langage C.

Structure complète du PCS		Énoncé qui la représente
séquentielle:		oui
conditionnelle:	Si	oui
	Si Sinon	oui
	Si Sinon-Si Sinon	oui
répétitive	avec CS au début	oui
	avec CS sans action au centre	non
	avec CS avec actions au centre	non
	avec CS à la fin	oui
	avec plusieurs CS sans action	non
	avec plusieurs CS avec actions	non

		60%

Tableau 7.6 Structures du pseudocode schématique disponibles en énoncés de langage C

Cette mesure montre l'apport du pseudocode schématique pour le langage C. Elle peut aussi servir à indiquer le niveau de difficulté associé à la construction d'un traducteur pour ce langage.

2. Les traducteurs Ada

L'évaluation des performances des traducteurs en langage Ada est faite à l'aide du compilateur Janus/Ada version 1.6.1 de RR Software sur un COMPAQ DESKPRO 386/20 en disque mémoire («Ram Disk»). Voici les résultats d'évaluation des critères obtenus dans les conditions spécifiées.

Traduction	**Directe**	**Complète**	**Ciblée**
Les critères qualitatifs visés			
Indentation	oui	oui	oui
Structuration	oui	oui	oui
Disposition des commentaires	oui	oui	oui
Les critères qualitatifs comparatifs pour l'utilisateur			
Disponibilité des instructions de structure	non	non	oui
Les critères qualitatifs comparatifs pour le traducteur			
Usage d'énoncés appropriés	non	oui	oui
Présence de code superflu	oui	non	non

Tableau 7.7 Résultats qualitatifs pour les traducteurs de langage Ada

Traduction	Directe	Complète	Ciblée
Les critères quantitatifs associés à la traduction			
Le temps de traduction (sans écriture)			
séquentielle	ref	0%	0%
conditionnelle	ref	+2%	+2%
répétitive	ref	+8%	+8%
Le temps de traduction (avec écriture)			
séquentielle	ref	+1%	+1%
conditionnelle	ref	+2%	+2%
répétitive	ref	-9%	-9%
Espace mémoire de traduction			
séquentielle	n/a	+218%	+218%
conditionnelle	n/a	+232%	+232%
répétitive	n/a	+280%	+280%
Les critères quantitatifs associés au code produit			
Le nombre de lignes du programme traduit			
séquentielle	ref	0%	0%
conditionnelle	ref	0%	0%
répétitive	ref	-34%	-34%
Le nombre de caractères du programme			
séquentielle	ref	0%	0%
conditionnelle	ref	0%	0%
répétitive	ref	-28%	-28%
Le temps de compilation			
séquentielle	ref	0%	0%
conditionnelle	ref	0%	0%
répétitive	ref	-22%	-22%
Le temps d'exécution			
séquentielle	ref	0%	0%
conditionnelle	ref	0%	0%
répétitive	ref	-9%	-9%

Tableau 7.8 Résultats quantitatifs pour les traducteurs de langage Ada

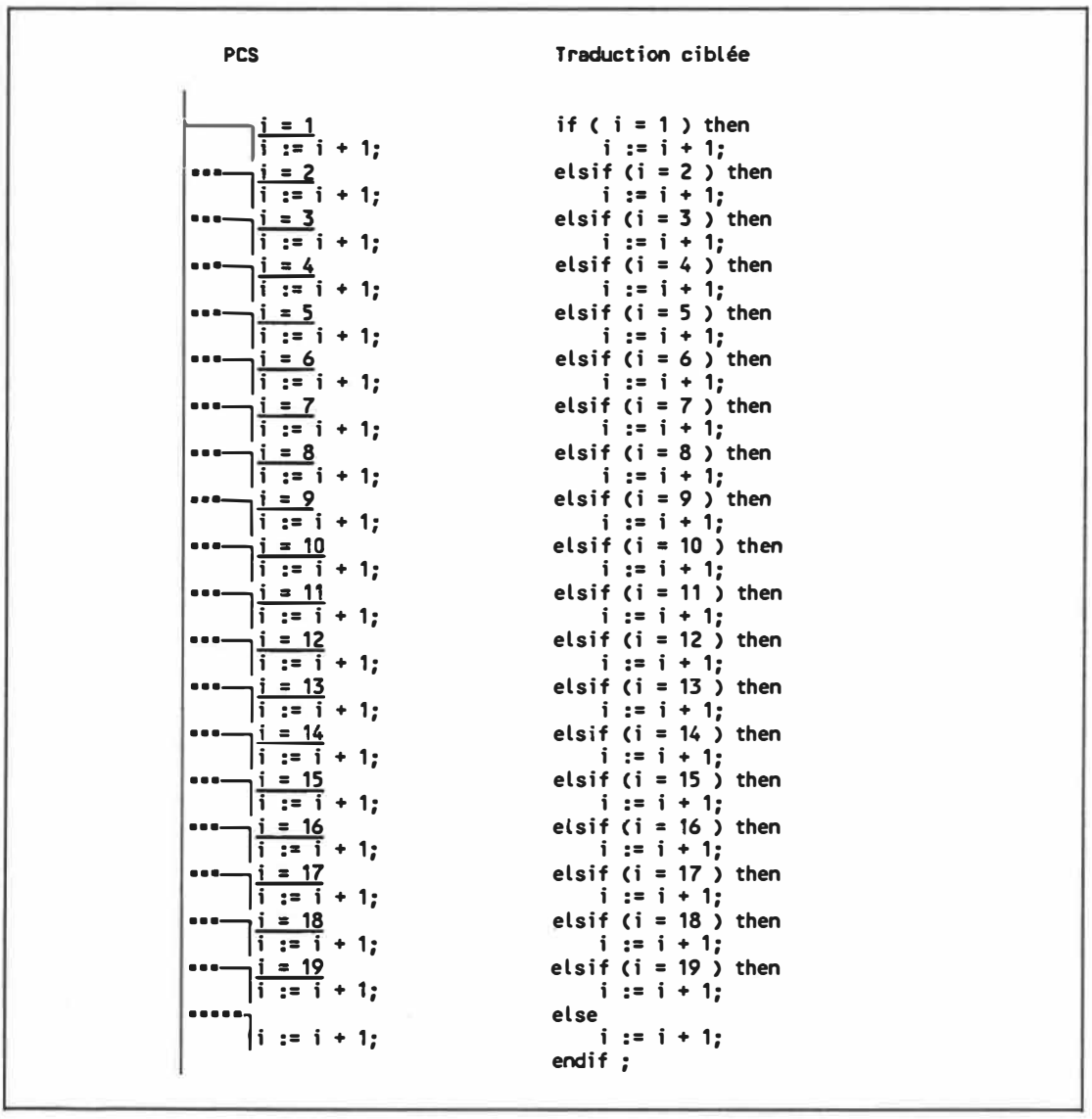
Conditionnelles à 20 branches	**"if else"**	**"case when"**
Les critères quantitatifs associés à la traduction		
Le temps de traduction (avec écriture)	ref	-2%
Espace mémoire de traduction	ref	0%
Les critères quantitatifs associés au code produit		
Le nombre de lignes du programme traduit	ref	+2%
Le nombre de caractères du programme	ref	-22%
Le temps de compilation	ref	-46%
Le temps d'exécution	ref	-71%
Conditionnelles à 250 branches	**"if else"**	**"case when"**
Les critères quantitatifs associés à la traduction		
Le temps de traduction (avec écriture)	ref	-2%
Espace mémoire de traduction	ref	0%
Les critères quantitatifs associés au code produit		
Le nombre de lignes du programme traduit	ref	0%
Le nombre de caractères du programme	ref	-26%
Le temps de compilation	ref	-48%
Le temps d'exécution	ref	-97%

Tableau 7.9 Résultats quantitatifs pour le code ciblé en langage Ada

Pour les traducteurs Ada, on constate à peu près les mêmes performances que pour le langage C.

Par contre, ici, on peut conclure que l'usage de composantes ciblées adéquates est toujours un avantage qui peut même amener une amélioration de 97% de la rapidité du code produit.

De la même manière que pour les évaluations de performance des traducteurs C, nous avons répété plusieurs fois une petite section de code et nous l'avons évaluée. Voici une section du programme qui nous a servi à mesurer les critères quantitatifs associés au code produit pour le traducteur ciblé. Les résultats permettent de comparer la structure conditionnelle "if else" et "case when" de 20 branches.



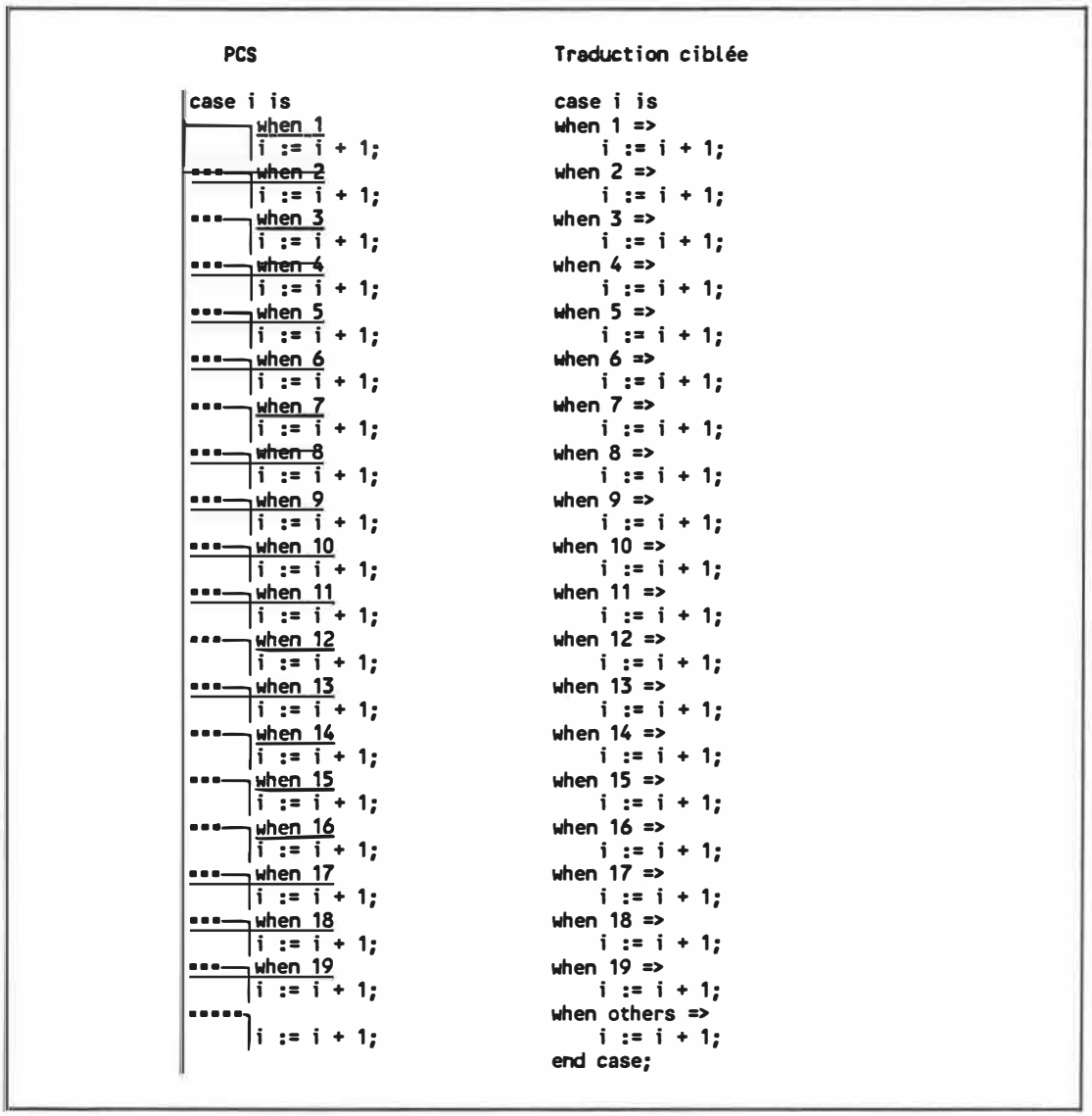
PCS

```
i = 1
i := i + 1;
i = 2
i := i + 1;
i = 3
i := i + 1;
i = 4
i := i + 1;
i = 5
i := i + 1;
i = 6
i := i + 1;
i = 7
i := i + 1;
i = 8
i := i + 1;
i = 9
i := i + 1;
i = 10
i := i + 1;
i = 11
i := i + 1;
i = 12
i := i + 1;
i = 13
i := i + 1;
i = 14
i := i + 1;
i = 15
i := i + 1;
i = 16
i := i + 1;
i = 17
i := i + 1;
i = 18
i := i + 1;
i = 19
i := i + 1;
i := i + 1;
```

Traduction ciblée

```
if ( i = 1 ) then
    i := i + 1;
elsif (i = 2 ) then
    i := i + 1;
elsif (i = 3 ) then
    i := i + 1;
elsif (i = 4 ) then
    i := i + 1;
elsif (i = 5 ) then
    i := i + 1;
elsif (i = 6 ) then
    i := i + 1;
elsif (i = 7 ) then
    i := i + 1;
elsif (i = 8 ) then
    i := i + 1;
elsif (i = 9 ) then
    i := i + 1;
elsif (i = 10 ) then
    i := i + 1;
elsif (i = 11 ) then
    i := i + 1;
elsif (i = 12 ) then
    i := i + 1;
elsif (i = 13 ) then
    i := i + 1;
elsif (i = 14 ) then
    i := i + 1;
elsif (i = 15 ) then
    i := i + 1;
elsif (i = 16 ) then
    i := i + 1;
elsif (i = 17 ) then
    i := i + 1;
elsif (i = 18 ) then
    i := i + 1;
elsif (i = 19 ) then
    i := i + 1;
else
    i := i + 1;
endif ;
```

Fig. 7.2 Section de code pour évaluer la structure "if else" de 20 branches en Ada



PCS

```
case i is
    when 1
        i := i + 1;
    when 2
        i := i + 1;
    when 3
        i := i + 1;
    when 4
        i := i + 1;
    when 5
        i := i + 1;
    when 6
        i := i + 1;
    when 7
        i := i + 1;
    when 8
        i := i + 1;
    when 9
        i := i + 1;
    when 10
        i := i + 1;
    when 11
        i := i + 1;
    when 12
        i := i + 1;
    when 13
        i := i + 1;
    when 14
        i := i + 1;
    when 15
        i := i + 1;
    when 16
        i := i + 1;
    when 17
        i := i + 1;
    when 18
        i := i + 1;
    when 19
        i := i + 1;
    
        i := i + 1;
```

Traduction ciblée

```
case i is
when 1 =>
    i := i + 1;
when 2 =>
    i := i + 1;
when 3 =>
    i := i + 1;
when 4 =>
    i := i + 1;
when 5 =>
    i := i + 1;
when 6 =>
    i := i + 1;
when 7 =>
    i := i + 1;
when 8 =>
    i := i + 1;
when 9 =>
    i := i + 1;
when 10 =>
    i := i + 1;
when 11 =>
    i := i + 1;
when 12 =>
    i := i + 1;
when 13 =>
    i := i + 1;
when 14 =>
    i := i + 1;
when 15 =>
    i := i + 1;
when 16 =>
    i := i + 1;
when 17 =>
    i := i + 1;
when 18 =>
    i := i + 1;
when 19 =>
    i := i + 1;
when others =>
    i := i + 1;
end case;
```

Fig. 7.3 Section de code servant à évaluer la structure "case when" de 20 branches en Ada.

La section de code est répétée 20 fois et elle est comprise dans une boucle de 30000 itérations. La valeur de "i" est établie à "1" au début de la boucle. Voici les résultats obtenus pour le nombre de lignes et le nombre de caractères traduits:

	Conditionnelle "if else"		Conditionnelle "case when"	
	Absolue	Relatif	Absolue	Relatif
Nombre de lignes	883	ref	903	+2%
Nombre de caractères	16749	ref	13049	-22%

Tableau 7.10 Dimension du code traduit pour les conditionnelles ciblées en Ada.

L'annexe contient un programme traduit par les traducteurs direct, complet et ciblé en langage Ada et un programme conçu pour le traducteur ciblé.

Pour le langage Ada on peut aussi comparer les différents types de traducteurs entre eux, en identifiant les énoncés de structure que chaque traducteur utilise:

Énoncés de structure du Ada		Utilisés par le traducteur		
		direct	complet	ciblé
séquentielle:		oui	oui	oui
conditionnelle:	if	oui	oui	oui
	case when	non	non	oui
répétitive:	loop	oui	oui	oui
	for	non	non	oui
	while	non	oui	oui
	exit	oui	oui	oui
	exit when	non	oui	oui
		---	---	---
		50%	75%	100%

Tableau 7.11 Énoncés du langage utilisés par le traducteur Ada

On constate que le traducteur ciblé utilise tous les énoncés de structure du langage.

On peut aussi évaluer le langage Ada en fonction des structures du pseudocode schématique:

Structure complète du PCS		Énoncé qui la représente
séquentielle:		oui
conditionnelle:	Si	oui
	Si Sinon	oui
	Si Sinon-Si Sinon	oui
répétitive	avec CS au début	oui
	avec CS sans action au centre	oui
	avec CS avec actions au centre	non
	avec CS à la fin	non
	avec plusieurs CS sans action	oui
	avec plusieurs CS avec actions	non

		70%

Tableau 7.12 Structures du pseudocode schématique disponibles en énoncé de langage Ada

Cette mesure démontre l'apport du pseudocode schématique pour le langage Ada. Comparativement au C, on constate que le langage Ada contient plus d'énoncés de structure s'approchant de ceux du pseudocode schématique.

3. Les traducteurs assembleur 6502

L'évaluation des performances des traducteurs assembleur 6502 est faite sur un COMPAQ DESKPRO 386/20 avec le macro assembleur ASM740 de Mitsubishi. Pour l'évaluation de la performance du code produit, nous avons utilisé un COMMODORE 64 avec l'assembleur PAL de Brad Templeton. Voici les résultats d'évaluation des critères obtenus dans les conditions spécifiées.

Traduction	**Directe**	**Complète**	**Ciblée**
Les critères qualitatifs visés			
Indentation	oui	oui	oui
Structuration	oui	oui	oui
Disposition des commentaires	oui	oui	oui
Les critères qualitatifs comparatifs pour l'utilisateur			
Disponibilité des instructions de structure	non	non	oui
Les critères qualitatifs comparatifs pour le traducteur			
Usage d'énoncés appropriés	non	non	oui
Présence de code superflu	oui	non	non

Tableau 7.13 Résultats qualitatifs pour les traducteurs de langage assembleur 6502

Traduction	Directe	Complète	Ciblée
Les critères quantitatifs associés à la traduction			
Le temps de traduction (sans écriture)			
séquentielle	ref	0%	0%
conditionnelle	ref	0%	+7%
répétitive	ref	0%	+6%
Le temps de traduction (avec écriture)			
séquentielle	ref	0%	0%
conditionnelle	ref	0%	-5%
répétitive	ref	0%	-14%
Espace mémoire de traduction			
séquentielle	n/a	+225%	+225%
conditionnelle	n/a	+340%	+340%
répétitive	n/a	+426%	+426%
Les critères quantitatifs associés au code produit			
Le nombre de lignes du programme traduit			
séquentielle	ref	0%	0%
conditionnelle	ref	-4%	-21%
répétitive	ref	0%	-35%
Le nombre de caractères du programme			
séquentielle	ref	0%	0%
conditionnelle	ref	-2%	-17%
répétitive	ref	+3%	-40%
Le temps de compilation			
séquentielle	ref	0%	0%
conditionnelle	ref	-2%	-19%
répétitive	ref	-7%	-40%
Le temps d'exécution			
séquentielle	ref	0%	0%
conditionnelle	ref	0%	-20%
répétitive	ref	0%	-36%

Tableau 7.14 Résultats quantitatifs pour les traducteurs de langage assembleur 6502

Pour les traducteurs assembleur 6502, on constate à peu près les mêmes performances que pour les autres langages.

Par contre, ici, on peut conclure que l'usage adéquat de composantes ciblées est toujours un avantage qui peut même se concrétiser en une amélioration de 36% au niveau de la rapidité du code produit.

De la même manière que pour les autres évaluations de performances, nous avons répété plusieurs fois une petite section de code et nous l'avons évaluée.

Voici une section du programme qui a servi pour mesurer les critères quantitatifs associés au code produit pour la conditionnelle, relativement au temps d'exécution.

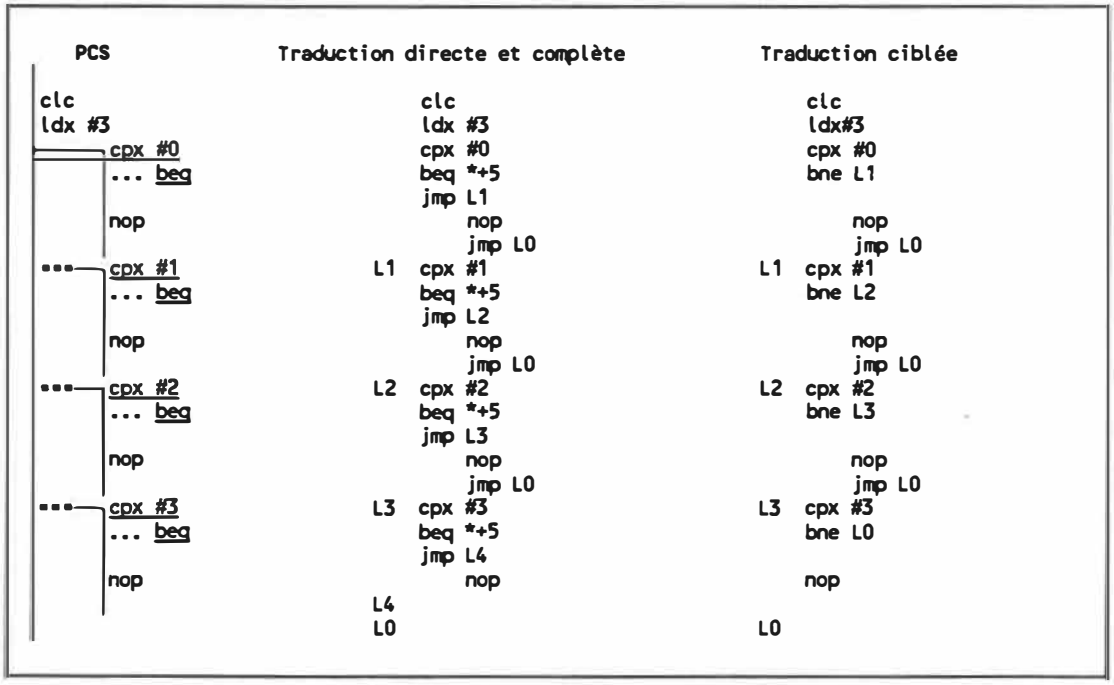


Fig. 7.4 Section de code servant à mesurer le temps d'exécution de la conditionnelle en assembleur 6502.

La section de code est comprise dans une boucle de 65535 itérations. Le résultat obtenu pour le temps d'exécution est présenté ici:

	Traduction directe et complète	Traduction ciblée
Valeur enregistrée	146 ds	117 ds
Valeur relative	ref	-20%

Tableau 7.15 Temps d'exécution du programme d'évaluation du temps d'exécution de la conditionnelle en assembleur 6502.

Dans ce cas particulier, où le corps de la boucle est relativement petit, on peut déterminer les performances comparatives des différentes approches en déduisant le temps d'exécution d'une boucle sans son contenu. Sur le COMMODORE 64, la boucle sans contenu prend 71 dixièmes de seconde pour s'exécuter. Si on soustrait ce nombre au temps enregistré, on constate une augmentation des performances pour l'approche ciblée de près de 39%.

L'annexe contient un programme traduit par les traducteurs direct, complet et ciblé en langage assembleur 6502.

En assembleur, il n'existe pas vraiment de structure conditionnelle ou répétitive comme en C ou en Ada car la notion de portée, nécessaire pour identifier une structure, n'est pas présente. On ne peut donc pas comparer les différentes approches de traduction du traducteur par rapport aux énoncés de structure du langage.

Il est aussi difficile d'évaluer l'apport du pseudocode schématique sur le langage assembleur, puisqu'il existe seulement des énoncés séquentiels, comparatifs et de branchement. Les structures complètes (séquentielles exeptées) sont inexistantes dans ce langage. On peut donc dire que pour le langage assembleur 6502, l'apport de structure par le pseudocode schématique ajoute une dimension lui permettant de se comparer à des langages de plus haut niveau.

CONCLUSION

Il faut réfléchir longuement pour écrire peu de mots.

Anonyme

La méthode de construction de traducteurs de pseudocode schématique proposée est appliquée aux langages C, Ada et assembleur 6502. La méthode est aussi applicable à tout langage orienté procédure possédant des énoncés de tests et de branchements et/ou des énoncés de structures conditionnelles et de structures répétitives.

Les trois variantes de la méthode associées aux trois genres de traductions répondent aux différents besoins des concepteurs et des utilisateurs.

La variante directe, basée sur les règles de production de la grammaire du pseudocode schématique, demeure la variante la plus facile à réaliser. La forme du code produit, par contre, diffère plus du code écrit à la main et les performances peuvent en être affectées. Ces éléments représentent une source potentielle d'insatisfaction.

La variante complète qui propose de recourir aux structures complètes du pseudocode schématique pour traduire un programme, permet une amélioration sensible de la forme du code produit et, par conséquent, entraîne potentiellement de meilleures performances par rapport à la variante directe.

La variante ciblée, orientée sur les énoncés de structure du langage cible, permet l'utilisation de structures inaccessibles autrement. Cette caractéristique, lorsqu'utilisée à bon escient, produit une forme de code qui se rapproche le plus du code écrit à la main et entraîne une amélioration des performances. Par contre, cette variante va parfois à l'encontre de la philosophie du pseudocode schématique qui tente d'abstraire les détails d'implantation.

Nous devons concéder que pour l'utilisateur, l'approche ciblée est favorisée. Afin d'éviter d'aller à l'encontre de la philosophie du pseudocode schématique nous suggérons au concepteur d'un traducteur ciblé de privilégier l'identification de la forme sémantique appropriée à l'aide du contexte plutôt qu'à l'aide de directives au traducteur.

Nous favorisons l'approche ciblée à cause des performances qui lui sont reliées. La présence d'instructions plus spécifiques comme le "switch" en langage C, sert à indiquer au compilateur qu'il existe une meilleure façon de compiler ce type de structure conditionnelle. Nous croyons que les performances de l'approche ciblée peuvent être égalées par les autres approches à condition que les compilateurs compensent par l'optimisation.

Les traducteurs directs, complets et ciblés pour les langages C, Ada et assembleur 6502 ont été comparés à l'aide d'un ensemble de programmes. Ces derniers ont contribué à mettre en évidence les caractéristiques de chacun des traducteurs. L'apparence et les performances du code traduit sont supérieures pour les traducteurs ciblés. Les traducteurs directs brillent par leur simplicité. Les

traducteurs complets utilisent le maximum d'informations accessibles sans consulter le texte du programme. Cette particularité à pour effet d'établir leur performance quelque part entre l'approche directe et l'approche ciblée.

En résumé, les variantes de la méthode de construction de traducteurs de pseudocode schématique proposent de recourir aux principes, techniques et outils utilisés pour la réalisation de compilateurs. Cette démarche permet de produire rapidement des traducteurs de pseudocode schématique efficaces.

Pour la poursuite de recherches dans le domaine de la traduction du pseudocode schématique, une approche plus globale devrait être envisagée. Cette approche utiliserait une forme paramétrisée du langage cible orienté procédure, et permettrait ainsi à l'utilisateur de construire, lui-même, son traducteur en fonction de ses besoins particuliers.

BIBLIOGRAPHIE

Aho, A.V. [1980]. "Translator Writing Systems: Where Do They Now Stand?," *Computer* **13:8**, 9-14.

Aho, A.V., B.W. Kernighan and P.J. Weinberger [1988]. *The AWK Programming Language*, AT&T Bell Laboratories, Murray Hill, New Jersey, USA.

Aho, A.V., R. Sethi and J.D. Ullman [1986]. *Compilers Principles, Techniques, and Tools*, Addison-Wesley, Reading, Massachusetts, USA.

Booch, G. [1987a]. *Software Engineering with Ada, Second Edition*, The Benjamin/Cummings Publishing Company, Menlo Park, California, USA.

Booch, G. [1987b]. *Software Components with Ada - Structures, Tools, and Subsystems* , The Benjamin/Cummings Publishing Company, Menlo Park, California, USA.

Brown, P.J. [1984]. *Starting with UNIX,* Addison-Wesley, Reading, Massachusetts, USA.

dBase III for your 16-bit PC [1984], Ashton-Tate, Culver, California, USA.

Forsyth, C.H. [1982]. *LEX - A Lexical Analyzer Generator*, University of Waterloo, Waterloo, Ontario, Canada.

Graham, S.L. [1980]. "Table-driven code generation," *Computer* **13:8**, 25-34.

Harbison, S.P. and G.L. Steele Jr. [1987]. *C: A Reference Manual*, Prentice-Hall, Englewood Cliffs, New Jersey, USA.

JANUS/Ada - Compiler Package - User Manual [1986], R.R. Software, Madison, Wisconsin, USA.

Johnson, S.C. [1978]. *YACC: Yet Another Compiler-Compiler*, Bell Laboratories, Murray Hill, New Jersey, USA.

Johnson, S.C. and M.E. Lesk [1978]. "Language Development Tools," *The Bell System Technical Journal* **57:6**, 2155-2174.

Katwijk, J.V. [1983]. "A preprocessor for YACC," *ACM SIGPLAN Notices* **18:10**, 12-15.

Katzan, H. [1978]. *FORTRAN 77,* Van Nostrand Reinold Company, New York, N. Y., USA.

Kernighan, B.W. and D.M. Ritchie [1978]. *The C Programming Language,* Prentice-Hall, Englewood Cliffs, New Jersey, USA.

Knuth, D.E. [1964]. "Backus Normal Form vs. Backus Naur Form," Communications of the ACM 7:12, 735-736.

Lesk, M.E. and E. Schmidt [1978]. *LEX - A Lexical Analyzer Generator,* Bell Laboratories, Murray Hill, New Jersey, USA.

Magnenat-Thalmann, N., D. Thalmann et J. Vaucher [1981]. *Le Langage Pascal ISO avec Pascal 6000 et Pascal UCSD*, Gaëtan Morin, Chicoutimi, Québec, Canada.

Microsoft C Compiler - User's Guide [1986], Microsoft Corporation, Redmond, Washington, USA.

Murach, M. [1975]. *Standard COBOL,* Science Research Associates, Chicago, USA.

Programming in VAX C [1985], Digital Equipment Corporation, Nashua, New Hampshire, USA.

Programming in VAX Pascal [1985], Digital Equipment Corporation, Nashua, New Hampshire, USA.

Ramamoorthy, C.V. and K. Siyan [1983]. "Software Engineering," *Encyclopedia of Computer Science and Engineering*, 2nd Ed., Van Nostrand Reihold Co., New York, N.Y., USA, 1349-1352.

Robillard, P.N. and D. Thalmann [1979]. "Grammar for schematic pseudocode," *Technical Report,* École Polytechnique de Montréal, Montréal, Québec, Canada.

Robillard, P.N. and D. Thalmann [1980]. "Complex problem solving using schematic pseudocode (SPC)," *Publication #373,* Université de Montréal, Montréal, Québec, Canada.

Robillard, P.N. and R. Plamondon [1981]. "Introduction to SCHEMACODE," *Technical Report EP81-R-20,* École Polytechnique de Montréal, Montréal, Québec, Canada.

Robillard, P.N. and P. Tan [1983]. "SCHEMACODE: Software and tool for FORTRAN and Pascal coding," *Technical Report EP83-R-38,* École Polytechnique de Montréal, Montréal, Québec, Canada.

Robillard, P.N. [1984]. "Schematic construct notation for programming and its computer automation by SCHEMACODE," *Technical Report EPM/RT-84-13,* École Polytechnique de Montréal, Montréal, Québec, Canada.

Robillard, P.N. [1985a]. *Le logiciel: de sa conception à sa maintenance,* Gaëtan Morin, Chicoutimi, Québec, Canada.

Robillard, P.N. [1985b]. "A Software Tool and a Schematic Notation that improve the use of programming languages," *A second Conference on Software Development Tools, Techniques, and Alternatives* (San Francisco, California December 2-5), IEEE Computer Society, Washington, D.C., USA, 149-158.

Robillard, P.N. [1986]. "Schematic pseudocode for program constructs and its computer automation by SCHEMACODE," *Communications of the ACM* 29:11, 1072-1089.

Robillard, P.N., J.B. Trouvé and A. Grenier [1987]. "A pre-processor for schematic pseudocode," *Second international conference on computers and applications* (Beijing, Peking, June 23-27), IEEE Computer Society, Washington, D.C., USA, 904-911.

Robillard, P.N., J.B. Trouvé, A. Grenier & A. Beaucage [1988]. "L'automatisation du raffinement successif et de sa documentation," *Journées Internationales sur le Génie Logiciel et ses Applications* (Toulouse, France, 5-9 décembre), EC2, Nanterre, France, 1081-1090.

Sammet, J.E. [1983]. "Software History," *Encyclopedia of Computer Science and Engineering*, 2nd Ed., Van Nostrand Reihold Co., New York, N.Y., USA, 1353-1359.

Scanlon, L.J. [1980]. *6502 Software Design*, Woward W. Sams & Co., Indianapolis, Indiana, USA.

SCHEMACODE [1987]. *SCHEMACODE, User Guide*, Schemacode International Inc., Montréal, Québec, Canada.

Thalman, D. et B. Levrat [1979]. *Conception et implantation de langages de programmation: Une introduction à la compilation*, Gaëtan Morin, Chicoutimi, Québec, Canada.

Templeton, B. [1982]. *Personnal Assembly Language,* Pro-Line Software, Mississauga, Ontario, Canada.

Thalmann, D. et N. Magnenat-Thalmann [1981]. *COBOL: Une approche structurée à la résolution de problèmes,* Gaëtan Morin, Chicoutimi, Québec, Canada.

VAX-11 FORTRAN Language Reference Manual, Digital Equipment Corporation, Nashua, New Hampshire, USA, 1982.

Wirth, N. [1976]. *Algorithms + Data Structures = Programs*, Prentice-Hall, Englewood Cliffs, New Jersey, USA.

ANNEXE

A1 - Programmes typiques à traduire en pseudocode schématique

A1.1 - Programme PCS en langage C

```
     R 0
0    1  -SCHEMACODE 1.5.2 - Recherche en Genie Logiciel
     2
     3  -Programme en C qui peut etre traduit par le traducteur
     4  - direct ou complet ou cible.
     5
     6  main()
     7  {
     8
     9  -01 Declaration des variables
1    2  int compteur;
0   10
    11  -02 Utilisation d'une structure conditionnelle
2    2      compteur = 1;
     3  ┌──┐compteur == 1
     4  │  │ printf("On est en presence du premier cas\n");
     5  ***┌compteur == 2
     6  │  │ printf("On est en presence du deuxieme cas\n");
     7  *****
     8  │  │ printf("On est en presence d'un autre cas\n");
     9  │  │
0   12  │  │
    13  -03 Utilisation d'une structure repetitive avec condition de sortie au debut
3    2  │  │ compteur = 1;
     3  │  │
     4  │  *┌──┐compteur == 10
     5  │  │  *
     6  │  │ printf("Le compte est %d\n",^
     7  │  │ ...compteur );
     8  │  │
     9  │  │ -Incrementer le compteur
    10  │  │ compteur++;
    11  │  │
0   14  │  │
    15  -04 Utilisation d'une structure repetitive avec condition de sortie au milieu
4    2  │  │ compteur = 1;
     3  ┌──┐
     4  │  │ printf("Le compte est %d\n",
     5  │  │ ...compteur );
     6  │  │
     7  │  *┌──┐compteur == 10
     8  │  │  *
     9  │  │ -Incrementer le compteur
    10  │  │ compteur++;
    11  │  │
0   16  │  │
    17  }
```

A1.2 - Programme PCS en langage Ada

```
     R 0
0    1 -SCHEMACODE 1.5.2 - Recherche en Genie Logiciel
     2
     3 -Programme en Ada qui peut etre traduit par le traducteur
     4 - direct ou complet ou cible.
     5
     6 Pragma RangeCheck(off); Pragma Debug(off); Pragma Arithcheck(Off);
     7
     8 With Text_IO; Use Text_IO;
     9
    10 Procedure P_Ada Is
    11
    12 -01 Declaration des variables
1   2
    3 Package Int_IO Is New Integer_IO(Integer);
    4
    5 compteur    : Integer ;
    6
0   13
    14 Begin
    15
    16 -02 Utilisation d'une structure conditionnelle
2   2
    3 compteur := 1;
    4
    5 ┌──┐compteur = 1
    6 │  │ put("On est en presence du premier cas");
    7 │  │ new_line;
    8 ...─┐compteur = 2
    9 │  │ put("On est en presence du deuxieme cas");
   10 │  │ new_line;
   11 .....
   12 │  │ put("On est en presence d'un autre cas");
   13 │  │ new_line;
   14 │  │
0  17 │  │
   18 -03 Utilisation d'une structure repetitive avec condition de sortie au debut
3   2 │  │
    3 compteur := 1;
    4 │  │
    5 ┌──┐
    6 │ *┌──┐compteur = 10
    7 │  │ *│
    8 │  │ put("Le compte est ");
    9 │  │ Int_IO.put(compteur);
   10 │  │ new_line;
   11 │  │
   12 │  │-Incrementer le compteur
   13 │  │compteur :=
   14 │  │...compteur + 1;
   15 │  │
0  19 │  │
   20 -04 Utilisation d'une structure repetitive avec condition de sortie au milieu
4   2 │  │
    3 compteur := 1;
    4 │  │
    5 ┌──┐
    6 │  │ put("Le compte est ");
    7 │  │ Int_IO.put(compteur);
    8 │  │ new_line;
    9 │  │
   10 │ *┌──┐compteur = 10
   11 │  │ *│
```

```
   12 |    | |-Incrementer le compteur
   13 |    | |compteur :=
   14 |    | |...compteur + 1;
   15 |    | |
0  21 |
   22 |End ;
      |_____|
```

A1.3 - Programme PCS en langage assembleur 6502

```
      R 0
      ┌
0   1 │-SCHEMACODE 1.5.2 - Recherche en Genie Logiciel
    2 │
    3 │-Programme en Assembleur 6502 qui peut etre traduit par le traducteur
    4 │- direct ou complet ou cible.
    5 │
    6 │*= $F000
    7 │
    8 │lda #0
    9 │clc
   10 │
   11 │┌──────bcc
   12 │       -Le carry est "clear"
   13 │       lda #1
   14 │•••┌───cmp #0
   15 │   │   ...beq
   16 │   │   -L'acumulateur est egale a 0
   17 │   │   lda #2
   18 │•••••
   19 │       -Autre cas
   20 │       lda #3
   21 │
   22 │┌──┐
   23 │   │   -Commentaire en debut de repetitive
   24 │   *┌──cmp #3
   25 │    │  ...beq
   26 │    *
   27 │   │   -Incrementer l'accumulateur
   28 │   │   clc
   29 │   │   adc #1
   30 │   │
   31 │   .end
      └
```

A2 - Code produit par les traducteurs directs (typique)

A2.1 - Programme source direct en langage C

```
/ *Fichier : p_c.c */
/ *Raffinement # 0 */
/ *SCHEMACODE 1.5.2 - Recherche en Genie Logiciel */

/ *Programme en C qui peut etre traduit par le traducteur */
/ * direct ou complet ou cible. */

main()
{

/ *Raffinement # 1 */
/ *Declaration des variables */
int compteur;

/ *Raffinement # 2 */
/ *Utilisation d'une structure conditionnelle */

compteur = 1;

if ( compteur == 1 )
  {
     printf("On est en presence du premier cas\n");
  }
else if ( compteur == 2 )
  {
     printf("On est en presence du deuxieme cas\n");
  }
else
  {
     printf("On est en presence d'un autre cas\n");
  }

/ *Raffinement # 3 */
/ *Utilisation d'une structure repetitive avec condition de sortie au debut */

compteur = 1;

while(1)
  {
    if ( compteur == 10 )
      {
        break;
      }
    printf("Le compte est %d\n",compteur );

    / *Incrementer le compteur */
    compteur++;
  }

/ *Raffinement # 4 */
/ *Utilisation d'une structure repetitive avec condition de sortie au milieu */

compteur = 1;

while(1)
  {
    printf("Le compte est %d\n",
    compteur );

    if ( compteur == 10 )
      {
        break;
      }
```

```
    /* Incrementer le compteur */
    compteur++;
  }

}
```

A2.2 - Programme source direct en langage Ada

```
-- Fichier : p_ada.pkg
-- Raffinement # 0
-- SCHEMACODE 1.5.2 - Recherche en Genie Logiciel

-- Programme en Ada qui peut etre traduit par le traducteur
--  direct ou complet ou cible.

Pragma RangeCheck(off); Pragma Debug(off); Pragma Arithcheck(Off);

With Text_IO; Use Text_IO;

Procedure P_Ada Is

-- Raffinement # 1
-- Declaration des variables

Package Int_IO Is New Integer_IO(Integer);

compteur    : Integer ;


Begin

-- Raffinement # 2
-- Utilisation d'une structure conditionnelle

compteur := 1;

if ( compteur = 1 ) then
     put("On est en presence du premier cas");
     new_line;
elsif ( compteur = 2 ) then
     put("On est en presence du deuxieme cas");
     new_line;
else
     put("On est en presence d'un autre cas");
     new_line;
end if ;

-- Raffinement # 3
-- Utilisation d'une structure repetitive avec condition de sortie au debut

compteur := 1;

loop
    if ( compteur = 10 ) then
        exit ;
    end if ;
     put("Le compte est ");
     Int_IO.put(compteur);
     new_line;

    -- Incrementer le compteur
    compteur := compteur + 1;
end loop ;

-- Raffinement # 4
-- Utilisation d'une structure repetitive avec condition de sortie au milieu

compteur := 1;

loop
     put("Le compte est ");
```

```
     Int_IO.put(compteur);
     new_line;

    if ( compteur = 10 ) then
        exit ;
    end if ;
    -- Incrementer le compteur
    compteur := compteur + 1;
end loop ;

End ;
```

A2.3 - Programme source direct en langage assembleur 6502

```
            ; Fichier : p_asm.asm
            ; Raffinement # 0
            ; SCHEMACODE 1.5.2 - Recherche en Genie Logiciel

            ; Programme en Assembleur 6502 qui peut etre traduit par le traducteur
            ;  direct ou complet ou cible.

        *= $F000

        lda #0
        clc

        bcc *+5
        jmp L1
                ; Le carry est "clear"
            lda #1
            jmp L0
L1      cmp #0
        beq *+5
        jmp L2
                ; L'acumulateur est egale a 0
            lda #2
            jmp L0
L2              ; Autre cas
            lda #3
L3
L0
L5              ; Commentaire en debut de repetitive
            cmp #3
            beq *+5
            jmp L6
                jmp L4
L6              ; Incrementer l'accumulateur
            clc
            adc #1
            jmp L5
L4      .end
```

A3 - Code produit par les traducteurs complets (typique)

A3.1 - Programme source complet en langage C

```
/ *Fichier : p_c.c */
/ *Raffinement # 0 */
/ *SCHEMACODE 1.5.2 - Recherche en Genie Logiciel */

/ *Programme en C qui peut etre traduit par le traducteur */
/ * direct ou complet ou cible. */

main()
{

/* Raffinement # 1 */
/ *Declaration des variables */
int compteur;

/ *Raffinement # 2 */
/ *Utilisation d'une structure conditionnelle */

compteur = 1;

if ( compteur == 1 )
  {
     printf("On est en presence du premier cas\n");
  }
else if ( compteur == 2 )
  {
     printf("On est en presence du deuxieme cas\n");
  }
else
  {
     printf("On est en presence d'un autre cas\n");
  }

/ *Raffinement # 3 */
/ *Utilisation d'une structure repetitive avec condition de sortie au debut */

compteur = 1;

while(!( compteur == 10 ))
  {
    printf("Le compte est %d\n",compteur );

    / *Incrementer le compteur */
    compteur++;
  }

/ *Raffinement # 4 */
/ *Utilisation d'une structure repetitive avec condition de sortie au milieu */

compteur = 1;

for(;;)
  {
    printf("Le compte est %d\n",
    compteur );

    if ( compteur == 10 ) break;
    /* Incrementer le compteur */
    compteur++;
  }

}
```

A3.2 - Programme source complet en langage Ada

```
-- Fichier : p_ada.pkg
-- Raffinement # 0
-- SCHEMACODE 1.5.2 - Recherche en Genie Logiciel

-- Programme en Ada qui peut etre traduit par le traducteur
--  direct ou complet ou cible.

Pragma RangeCheck(off); Pragma Debug(off); Pragma Arithcheck(Off);

With Text_IO; Use Text_IO;

Procedure P_Ada Is

-- Raffinement # 1
-- Declaration des variables

Package Int_IO Is New Integer_IO(Integer);

compteur    : Integer ;


Begin

-- Raffinement # 2
-- Utilisation d'une structure conditionnelle

compteur := 1;

if ( compteur = 1 ) then
     put("On est en presence du premier cas");
     new_line;
elsif ( compteur = 2 ) then
     put("On est en presence du deuxieme cas");
     new_line;
else
     put("On est en presence d'un autre cas");
     new_line;
end if ;

-- Raffinement # 3
-- Utilisation d'une structure repetitive avec condition de sortie au debut

compteur := 1;

while (not( compteur = 10 )) loop
     put("Le compte est ");
     Int_IO.put(compteur);
     new_line;

    -- Incrementer le compteur
    compteur := compteur + 1;
end loop ;

-- Raffinement # 4
-- Utilisation d'une structure repetitive avec condition de sortie au milieu

compteur := 1;

loop
     put("Le compte est ");
     Int_IO.put(compteur);
     new_line;
```

```
    exit when ( compteur = 10 );
    -- Incrementer le compteur
    compteur := compteur + 1;
end loop ;

End ;
```

A3.3 - Programme source complet en langage assembleur 6502

```
            ; Fichier : p_asm.asm
            ; Raffinement # 0
            ; SCHEMACODE 1.5.2 - Recherche en Genie Logiciel

            ; Programme en Assembleur 6502 qui peut etre traduit par le traducteur
            ;  direct ou complet ou cible.

        *= $F000

        lda #0
        clc

        bcc *+5
        jmp L1
                ; Le carry est "clear"
            lda #1
            jmp L0
L1      cmp #0
        beq *+5
        jmp L2
                ; L'acumulateur est egale a 0
            lda #2
            jmp L0
L2              ; Autre cas
            lda #3
L0
L4              ; Commentaire en debut de repetitive
            cmp #3
            beq *+5
                jmp *+6
                jmp L3
                ; Incrementer l'accumulateur
            clc
            adc #1
            jmp L4
L3      .end
```

A4 - Code produit par les traducteurs ciblés (typique)

A4.1 - Programme source ciblé en langage C

```
/ *Fichier : p_c.c */
/ *Raffinement # 0 */
/ *SCHEMACODE 1.5.2 - Recherche en Genie Logiciel */

/ *Programme en C qui peut etre traduit par le traducteur */
/ * direct ou complet ou cible. */

main()
{

/ *Raffinement # 1 */
/ *Declaration des variables */
int compteur;

/ *Raffinement # 2 */
/ *Utilisation d'une structure conditionnelle */

compteur = 1;

if ( compteur == 1 )
  {
     printf("On est en presence du premier cas\n");
  }
else if ( compteur == 2 )
  {
     printf("On est en presence du deuxieme cas\n");
  }
else
  {
     printf("On est en presence d'un autre cas\n");
  }

/ *Raffinement # 3 */
/ *Utilisation d'une structure repetitive avec condition de sortie au debut */

compteur = 1;

while(!( compteur == 10 ))
  {
    printf("Le compte est %d\n",compteur );

    / *Incrementer le compteur */
    compteur++;
  }

/ *Raffinement # 4 */
/ *Utilisation d'une structure repetitive avec condition de sortie au milieu */

compteur = 1;

for(;;)
  {
    printf("Le compte est %d\n",
    compteur );

    if ( compteur == 10 ) break;
    / *Incrementer le compteur */
    compteur++;
  }

}
```

A4.2 - Programme source ciblé en langage Ada

```
-- Fichier : p_ada.pkg
-- Raffinement # 0
-- SCHEMACODE 1.5.2 - Recherche en Genie Logiciel

-- Programme en Ada qui peut etre traduit par le traducteur
--  direct ou complet ou cible.

Pragma RangeCheck(off); Pragma Debug(off); Pragma Arithcheck(Off);

With Text_IO; Use Text_IO;

Procedure P_Ada Is

-- Raffinement # 1
-- Declaration des variables

Package Int_IO Is New Integer_IO(Integer);

compteur    : Integer ;


Begin

-- Raffinement # 2
-- Utilisation d'une structure conditionnelle

compteur := 1;

if ( compteur = 1 ) then
     put("On est en presence du premier cas");
     new_line;
elsif ( compteur = 2 ) then
     put("On est en presence du deuxieme cas");
     new_line;
else
     put("On est en presence d'un autre cas");
     new_line;
end if ;

-- Raffinement # 3
-- Utilisation d'une structure repetitive avec condition de sortie au debut

compteur := 1;

while (not( compteur = 10 )) loop
     put("Le compte est ");
     Int_IO.put(compteur);
     new_line;

    -- Incrementer le compteur
    compteur := compteur + 1;
end loop ;

-- Raffinement # 4
-- Utilisation d'une structure repetitive avec condition de sortie au milieu

compteur := 1;

loop
     put("Le compte est ");
     Int_IO.put(compteur);
     new_line;
```

```
    exit when ( compteur = 10 );
    -- Incrementer le compteur
    compteur := compteur + 1;
end loop ;

End ;
```

A4.3 - Programme source ciblé en langage assembleur 6502

```
          ; Fichier : p_asm.asm
          ; Raffinement # 0
          ; SCHEMACODE 1.5.2 - Recherche en Genie Logiciel

          ; Programme en Assembleur 6502 qui peut etre traduit par le traducteur
          ;  direct ou complet ou cible.

      *= $F000

      lda #0
      clc

      bcs L1
              ; Le carry est "clear"
          lda #1
          jmp L0
L1    cmp #0
      bne L2
              ; L'acumulateur est egale a 0
          lda #2
          jmp L0
L2            ; Autre cas
          lda #3
L0
L4            ; Commentaire en debut de repetitive
          cmp #3
          beq L3
              ; Incrementer l'accumulateur
          clc
          adc #1
          jmp L4
L3    .end
```

A5 - Programmes ciblés à traduire en pseudocode schématique

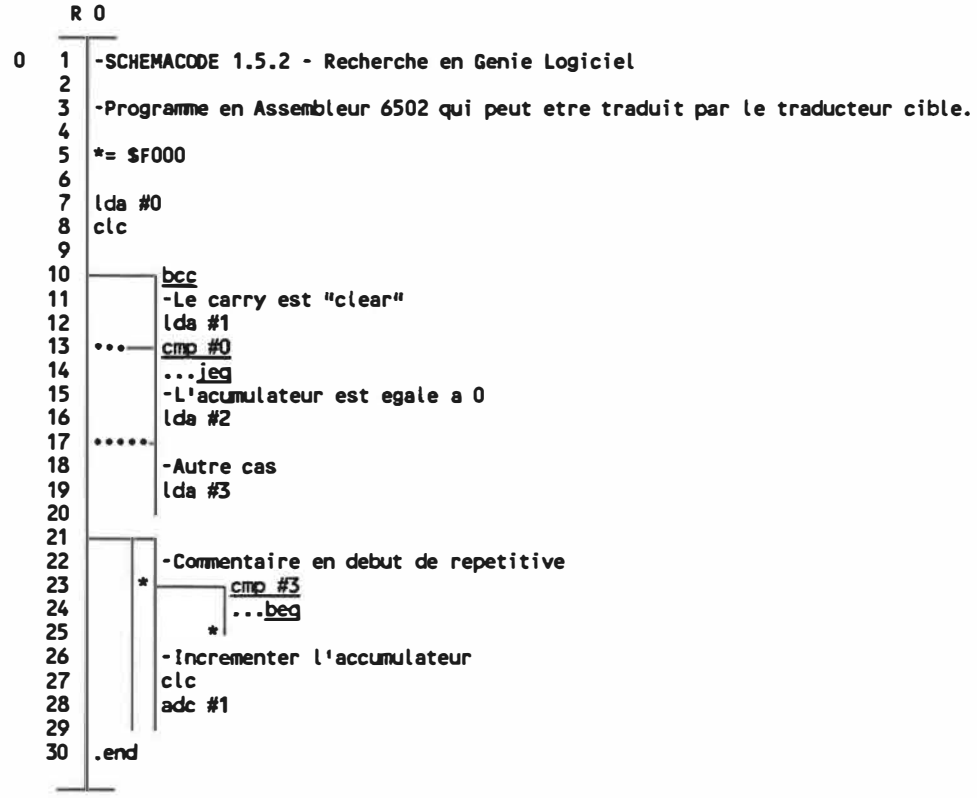
A5.1 - Programme PCS ciblé en langage C

```
     R 0
     ┌─
0    1  -SCHEMACODE 1.5.2 - Recherche en Genie Logiciel
     2
     3  -Programme en C qui peut etre traduit par le traducteur cible.
     4
     5  main()
     6  {
     7
     8  -01 Declaration des variables
1    2  int compteur;
0    9
    10  -02 Utilisation d'une structure conditionnelle
2    2
     3  compteur = 1;
     4
     5  switch (compteur)
     6  ┌─── case 1:
     7  │     printf("On est en presence du premier cas\n");
     8  ...── case '2': case '3':
     9  │     printf("On est en presence du deuxieme ou du troisieme cas\n");
    10  .....
    11  │     printf("On est en presence d'un autre cas\n");
    12  │
0   11  │
    12  -03 Utilisation d'une structure repetitive (for)
3    2  │
     3  │
     4  │  *┌─── compteur = 0; compteur <= 10 ; compteur++
     5  │   └*
     6  │  printf("Le compte est %d\n",
     7  │  ...compteur );
     8  │
     9  │
0   13  │
    14  -04 Utilisation d'une structure repetitive (!expression)
4    2  │
     3  compteur = 1;
     4  │
     5  ┌──┐
     6  │  printf("Le compte est %d\n",
     7  │  ...compteur );
     8  │  │
     9  │  *┌─── !( compteur == 1)
    10  │   └*
    11  │  -Incrementer le compteur
    12  │  compteur++;
    13  │
0   15  │
    16  }
     └─
```

A5.2 - Programme PCS ciblé en langage Ada

```
        R 0
0   1  -SCHEMACODE 1.5.2 - Recherche en Genie Logiciel
    2
    3  -Programme en Ada qui peut etre traduit par le traducteur cible.
    4
    5  Pragma RangeCheck(off); Pragma Debug(off); Pragma Arithcheck(Off);
    6
    7  With Text_IO; Use Text_IO;
    8
    9  Procedure P_Ada_c Is
   10
   11  -05 Declaration des variables
5   2
    3  Package Int_IO Is New Integer_IO(Integer);
    4
    5  compteur    : Integer ;
    6
0  12
   13  Begin
   14
   15  -02 Utilisation d'une structure conditionnelle
2   2
    3  compteur := 1;
    4
    5  case compteur is
    6      when 1
    7      put("On est en presence du premier cas");
    8      new_line;
    9  ...when 2
   10      put("On est en presence du deuxieme cas");
   11      new_line;
   12  .....
   13      put("On est en presence d'un autre cas");
   14      new_line;
   15
0  16
   17  -03 Utilisation d'une structure repetitive (for)
3   2
    3
    4  *       index in Integer range 1..10
    5          *
    6      put("Le compte est ");
    7      Int_IO.put(index);
    8      new_line;
    9
0  18
   19  -04 Utilisation d'une structure repetitive (not expression)
4   2
    3  compteur := 1;
    4
    5
    6      put("Le compte est ");
    7      Int_io.put(compteur);
    8      new_line;
    9
   10  *       not ( compteur = 1 )
   11          *
   12      -Incrementer le compteur
   13      compteur :=
   14      ...compteur + 1;
   15
0  20
   21  End ;
```

A5.3 - Programme PCS ciblé en langage assembleur 6502



A6- Code produit (ciblé)

A6.1 - Programme ciblé traduit en langage C

```
/ * Fichier: p_c_c.c */
/ *Raffinement # 0 */
/ *SCHEMACODE 1.5.2 - Recherche en Genie Logiciel */

/ *Programme en C qui peut etre traduit par le traducteur cible. */

main()
{

/ * Raffinement #1 */
/ *Declaration des variables */
int compteur;

/ *Raffinement # 2 */
/ *Utilisation d'une structure conditionnelle */

compteur = 1;

switch (compteur)
  {
  case 1:
     printf("On est en presence du premier cas\n");
    break;
  case '2': case '3':
     printf("On est en presence du deuxieme ou du troisieme cas\n");
    break;
  default:
     printf("On est en presence d'un autre cas\n");
    break;
  }

/ *Raffinement # 3 */
/ *Utilisation d'une structure repetitive (for) */

for(  compteur = 0; compteur <= 10 ; compteur++ )
  {
    printf("Le compte est %d\n",compteur );

  }

/ *Raffinement # 4 */
/ *Utilisation d'une structure repetitive (!expression) */

compteur = 1;

for(;;)
  {
    printf("Le compte est %d\n",
    compteur );

    if ( !( compteur == 1) ) break;
    / *Incrementer le compteur */
    compteur++;
  }

}
```

A6.2 - Programme ciblé traduit en langage Ada

```
-- Fichier : p_ada_c.pkg
-- Raffinement # 0
-- SCHEMACODE 1.5.2 - Recherche en Genie Logiciel

-- Programme en Ada qui peut etre traduit par le traducteur cible.

Pragma RangeCheck(off); Pragma Debug(off); Pragma Arithcheck(Off);

With Text_IO; Use Text_IO;

Procedure P_Ada_c Is

-- Raffinement # 5
-- Declaration des variables

Package Int_IO Is New Integer_IO(Integer);

compteur    : Integer ;


Begin

-- Raffinement # 2
-- Utilisation d'une structure conditionnelle

compteur := 1;

case compteur is
 when 1 =>
     put("On est en presence du premier cas");
     new_line;
when 2 =>
     put("On est en presence du deuxieme cas");
     new_line;
when others =>
     put("On est en presence d'un autre cas");
     new_line;
end case;

-- Raffinement # 3
-- Utilisation d'une structure repetitive (for)

for index in Integer range 1..10 loop
     put("Le compte est ");
     Int_IO.put(index);
     new_line;
end loop ;

-- Raffinement # 4
-- Utilisation d'une structure repetitive (not expression)

compteur := 1;

loop
     put("Le compte est ");
     Int_io.put(compteur);
     new_line;

    exit when (  not ( compteur = 1 ) );
    -- Incrementer le compteur
    compteur := compteur + 1;
end loop ;

End ;
```

A6.3 - Programme ciblé traduit en langage assembleur 6502

```
        ; Fichier : p_asm_c.asm
        ; Raffinement # 0
        ; SCHEMACODE 1.5.2 - Recherche en Genie Logiciel

        ; Programme en Assembleur 6502 qui peut etre traduit par le traducteur cible.

    *= $F000

    lda #0
    clc

    bcs L1
            ; Le carry est "clear"
        lda #1
        jmp L0
L1  cmp #0
    beq *+5
    jmp L2
            ; L'acumulateur est egale a 0
        lda #2
        jmp L0
L2          ; Autre cas
        lda #3
L0
L4          ; Commentaire en debut de repetitive
        cmp #3
        beq L3
            ; Incrementer l'accumulateur
        clc
        adc #1
        jmp L4
L3  .end
```