Le Langage MODULA-2: Concepts et expérience

Pieter H. Hartel

Vakgroep Informatica
Universiteit van Amsterdam

Résumé

Bien que Modula-2 soit un langage de programmation d'application générale, il est plus particulièrement conçu pour le développement de systèmes d'exploitation. Il a hérité de la plupart des caractéristiques de PASCAL. Le langage d'origine Modula a donné le concept de module, les facilités d'accès à la machine au bas niveau, et une syntaxe plus cohérente.

En comparaison avec ADA, Modula-2 est un langage simple mais efficace. Notamment les concepts de "generics" et de type de données abstraites générales sont absents de Modula-2. Le concept de modularisation permet de développer un système hiérarchique. Les parties dépendantes de la machine peuvent être regroupées dans des modules spécifiques.

Pour développer un programme en Modula-2, on a besoin d'un environnement de programmation particulier. Le système de fichiers hiérarchisé de UNIX*) permet d'organiser de façon systématique le grand nombre de fichiers impliqués dans une compilation typique d'un système ecrit en Modula-2.

1. Introduction

Depuis 1970 le langage Pascal est devenu un des langages de programmation les plus répandus. Le langage a été conçu pour l'enseignement plus que pour le développement des grands logiciels. Il n'est en effet pas très satisfaisant dans ce domaine. Le langage Modula par contre émergea des expériences en multiprogrammation. Il a été mis en oeuvre vers 1975. En 1977 un projet de recherche lancé à l'Institut für Informatik à Zürich sous la direction du Prof. Niklaus Wirth envisageait le développement d'un système ou le matériel et le logiciel seraient integrés. Ce système baptisé Lilith serait programmé en Modula-2. La définition de Modula-2 a été fixée en 1980 (WIR82).

*) UNIX est une marque déposée des Laboratoires BELL.

1.1 Rigidité au niveau du type de données

La famille de langages du type Pascal ont en commun la rigidité au niveau du type des données. En FORTRAN il est par exemple parfaitement légal de comparer des pommes à des poires. En Modula-2 le compilateur refuse d'accepter une programmation comme celle de la figure 1.

```
VAR Cartesiennes: RECORD
                     x, y: REAL
                  END;
    Polaires:     RECORD
                     r, theta: REAL
                  END;

BEGIN
   IF Cartesiennes = Polaires THEN
```

Figure 1: programmation erronée au niveau du type des données

La comparaison est considerée sans signification et rejetée par le compilateur. En Modula-2 il existe des mécanismes pour échapper au controle du compilateur. Notamment des paramètres de tableaux dynamiques permettent de passer des chaines de différentes tailles aux procédures.

1.2 Modules

Plusieurs langages modernes permettent au programmeur de grouper des définitions de données avec des définitions d'opérations sur ces données. On appelle l'ensemble de ces définitions un type abstrait de données. Dans le langage Modula-2 un module peut servir à délimiter un type abstrait de données. Ceci peut être illustré comme suit: Une pile est une collection d'éléments sur laquelle un certain ordre est défini. On ne peut qu'ajouter des éléments en haut de la pile. Le seul élément qui puisse être supprimé est aussi celui du haut. La spécification syntaxique d'une pile en Modula-2 serait celle de la figure 2.

```
DEFINITION MODULE LaPile;
   EXPORT QUALIFIED Element, Ajouter, Enlever;
   TYPE Element;
   PROCEDURE Ajouter (x: Element);
   PROCEDURE Enlever (VAR x: Element);
END LaPile.
```

Figure 2: Spécification syntaxique d'une pile

Ce qui n'est pas spécifié dans ce module de définition est la façon dont les éléments sont chargés en mémoire, s'il y a des restrictions sur le nombre des éléments, etc. Il suffit que la mise en oeuvre soit conforme à la définition de pile pour que l'ordre des opérations soit assuré. C'est à dire que l'on ne peut ni ajouter ni supprimer de éléments au milieu de la pile, car on n'a pas accès à la structure interne du type abstrait de données. La mise en oeuvre de la pile serait alors celle de la figure 3 (la procédure 'Enlever' n'étant pas montrée).

```
IMPLEMENTATION MODULE LaPile;
   CONST MaxPile = 100;
   TYPE  Element =  ..... ;
   VAR Pile: ARRAY [1 .. MaxPile] OF Element;
       PointeurDePile: [0 .. MaxPile];

   PROCEDURE Ajouter (x: Element);
      BEGIN
         IF PointeurDePile = MaxPile THEN
            HALT
         ELSE
            INC (PointeurDePile);
            Pile [PointeurDePile] := x
         END
      END Ajouter;

.....

BEGIN
   PointeurDePile := 0
END LaPile.
```

Figure 3: Mise en oeuvre de la pile par tableau

Dans le cas où un tableau n'est pas considéré comme le moyen le plus efficace pour mettre en oeuvre une pile, on peut sans changer la spécification remplacer la totalité du module d'exécution de la figure 3 par celui donné par la figure 4.

```
IMPLEMENTATION MODULE LaPile;
   TYPE Element = ..... ;
   TYPE LienDePile = POINTER TO EnregistrementDePile;
        EnregistrementDePile = RECORD
                                  donnee: Element;
                                  lien: LienDePile
                               END;
   VAR Pile: LienDePile;

   PROCEDURE Ajouter (x: Element);
      VAR tete: LienDePile;
   BEGIN
      NEW (tete);
      tete^.donnee := x;
      tete^.lien := Pile;
      Pile := tete
   END Ajouter;

.....

BEGIN
   Pile := NIL
END LaPile.
```

Figure 4: Mise en oeuvre de la pile par liste

1.2.1. La compilation séparée

Le syntaxe du langage Modula-2 permet de séparer physiquement des modules, par exemple dans des fichiers distincts. Cela permet au concepteur d'un grand logiciel de cacher tous les modules d'exécution. Par contre les modules de définition seront mis à la disposition des utilisateurs du système (par exemple les programmeurs d'application).

Comme la spécification n'est que syntaxique, des commentaires appropiés devront être présents pour que les utilisateurs puissent se servir du module. Le découpage d'un grand logiciel en modules a plusieurs avantages:

- La maintenance d'un tel système, constitué de modules bien isolés, est plus facile.
- Les parties de programme dépendantes de la machine peuvent être regroupées dans des modules spécifiques.

Le compilateur donne ensuite la garantie de cohérence entre les différents modules.

1.3 Programmation de bas niveau

Bien qui la rigidité des types de données soit souvent un avantage, il existe des circonstances dans lesquelles celle-ci est gênante. Souvent le matériel impose certaines structures sur les données. Dans un langage de système, des facilités pour exprimer ces structures doivent exister.

La syntaxe de Modula-2 permet au programmeur de spécifier l'adresse d'une variable dans la mémoire. Ceci permet d'accéder par exemple aux registres des unités d'entrée/sortie de la plupart des micros et des minis avec "memory mapped I/O". Il est évident qu'une telle facilité n'est à utiliser qu'au plus bas niveau d'un système d'exploitation.

La possibilité de transformer le type d'une variable est encore plus dangereuse. Dans la programmation de la figure 5 l'indentificateur du type 'PointeurVersEntier' est utilisé comme fonction. Celà indique au compilateur de transformer le type de l'argument (dans ce cas 'INTEGER' en 'PointeurVersEntier'). L'affectation a comme résultat que la variable 'janus' pointe vers l'adresse 1 de la mémoire. Ensuite l'écriture de la valeur '-1' à cette adresse peut avoir un effet particulier, cela dépend de la machine sur laquelle le programme est executé.

```
MODULE Essai;
   TYPE PointeurVersEntier = POINTER TO INTEGER;
   VAR janus: PointeurVersEntier;

BEGIN
   janus := PointeurVersEntier (1);
   janus^ := -1
END Essai.
```

Figure 5: Transformation du type de données

1.4 Utilisation des co-programmes

Une des tâches d'un système d'exploitation est de gérér les entrées/sorties d'un ordinateur. Souvent cela est fait par l'intermédiaire des interruptions. Le concept de co-programme (coroutine en anglais) est à la base de la notion d'interruptions, ce qui est illustré dans la figure 6.

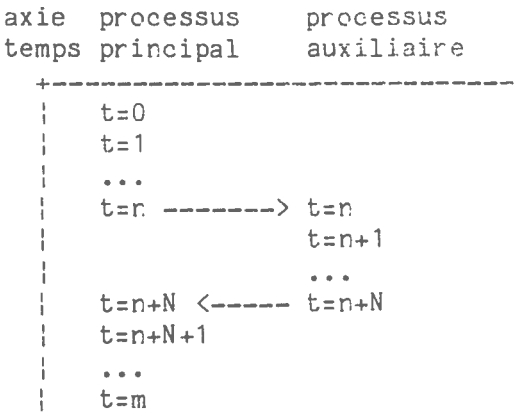

Figure 6: Interruptions traitées comme transfert de controle

Au moment de l'interruption à t=n le contrôle est transferé du processus principal (à gauche) au processus auxiliaire (à droite) qui sert à traiter l'interruption. Par exemple en cas de lecture d'une ligne RS232 il suffit de stocker le caractère reçu dans la mémoire, et de remettre la ligne en marche. Dès que les données sont sauvées, le controle est transféré au processus principal. Ce dernier ne s'aperçoit pas de ce qui ce passe (sauf en cas de chronométrage en temps réel).

Dans le langage Modula-2 le concept de co-programme est incorporé. Des procédures standards existent, pour transférer le contrôle d'un co-programme à un autre.

2. Modula-2 en comparaison avec Pascal

Modula-2 représente un grand progrès comparé à Pascal. En effet les points suivants sont importants:

- Le concept de module permet l'écriture de grands logiciels.
- La syntaxe est plus systématique. De nouvelles structures de contrôle sont incorporées, notamment la boucle générale (LOOP ... END) et la sortie des procédures (RETURN).
- Le concept de processus sous forme de co-programmes permet de gérér les entrées/sorties en utilisant des interruptions.
- Le langage Modula-2 permet de briser les règles rigides sur la cohérence des types.
- La notion de procédures constantes, qui sont les déclarations de procédures en Pascal, a été généralisée. Des types de procédures peuvent être déclarés.
- Des paramètres de tableaux dynamiques permettent l'écriture de librairies alors que cela n'est pas possible en Pascal.
- Le jeu de caractères ISO est recommandé pour l'écriture de programmes Modula-2.

Tous les concepts incorporés dans le langage Pascal ne sont pas présents en Modula-2. Il n'existe pas d'étiquettes, ni de "GOTO" en Modula-2. Cela rend l'écriture de programmes structurés plus facile (DIJ68).

On ne trouve pas le système d'entrées/sorties (Read, Write, FILE etc.) en Modula-2. En effet il n'existe pas de système d'entrées/sorties dans le langage. Par contre le programmeur dispose d'une librairie de modules, qui permettent la lecture et l'écriture des fichiers. Ainsi la dépendance à une machine est concentrée dans certains modules, ce qui rend le langage proprement dit plus portable.

Ce qui n'est pas prévu dans le langage Modula-2 c'est la possibilité (qui existe dans la plupart des compilateurs de Pascal) d'écrire une partie d'un programme dans un autre langage.

3. Modula-2 en comparaison avec ADA

En ce moment il n'existe que 3 compilateurs pour l'ensemble du langage ADA. Ces compilateurs sont parfois lents mais toujours énormes.

Pour le langage Modula-2 il existe déjà une dizaine d'executions complètes. Il est évident qu'il manque des concepts importants en Modula-2.

- Dans le langage ADA on retrouve complètement la notion du type abstrait de données, y compris la définition et le "overloading" des opérateurs. En Modula-2 il n'existe que la possibilité de définir des procédures.

- Le concept de "generics" d'ADA n'est pas incorporé en Modula-2.

- Pour des applications scientifiques et numériques il est important de pouvoir spécifier la précision des calculs. En Modula-2 cela n'est pas possible, tandis que ADA propose toute la gamme de précisions.

- Bien que la notion de tâches, présentes en ADA, soit déjà plus élaborée que ne le sont les co-programmes de Modula-2, il n'existe pas de support pour des systèmes multi-processeurs.

- Le traitement d'exceptions n'est pas incorporé dans le langage Modula-2.

4. Mise en oeuvre de Modula-2 sous UNIX

Pour qu'on puisse bénéficier de tous les avantages d'un langage modulaire tel que Modula-2 on a besoin d'un environnement de programmation particulier.

La gestion d'un grand nombre de fichiers impliqués dans la compilation d'un système en Modula-2 nécessite un ensemble d'outils et une base de données.

Sous UNIX le système de gestion des fichiers peut servir comme base de données. La plupart des modules sont séparés en une partie "spécification" et une partie "exécution". Chacune se trouve sur un fichier, la partie spécification du module 'x' sur le fichier 'x.def' et la partie execution sur le fichier 'x.mod'. Pour chaque module compilé sont crées 4 fichiers:

- Le fichier 'x.sym' contient la table de symboles qui permet au compilateur de vérifier la cohérence des opérations.
- Le fichier 'x.lst' contient la liste créee par le compilateur.

- Le fichier 'x.lnk' contient le code, éventuellement sous forme intermédiaire tel que 'M-code'.

- Le fichier 'x.ref' contient la table de symboles pour le metteur au point (debugger en anglais).

Après la compilation, l'éditeur de lien lie tous les fichiers '.lnk' impliqués dans le programme et crée un fichier '.lod' qui contient l'image partielle en mémoire du programme, ainsi que le fichier '.map' dans lequel on retrouve la table de chargement. Finalement le chargeur crée le fichier 'a.out' qui peut être exécuté par le "shell" d'UNIX. Le système de fichiers UNIX permet de stocker des fichiers dans un arbre de catalogues, de sous-catalogues etc. (figure 7).

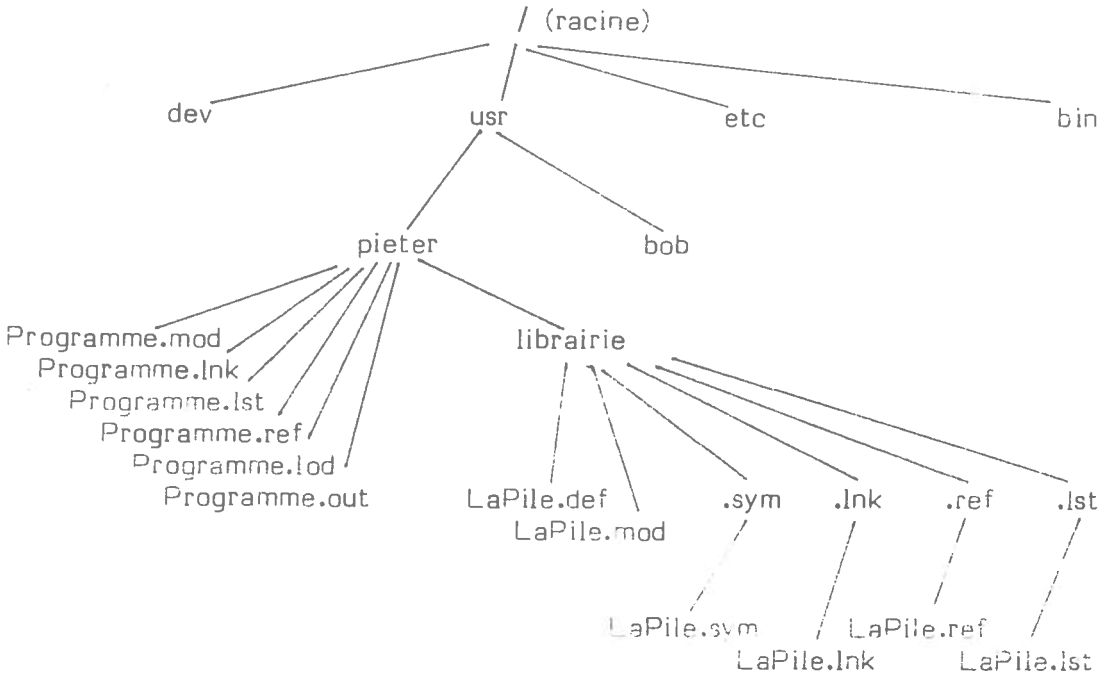

Figure 7: Arbre de fichiers

Les systèmes comme par exemple, le compilateur Modula-2 et sa librairie d'entrées/sorties, contiennent des centaines de modules. Donc le programmeur responsable du système doit se souvenir de l'endroit ou se trouvent ces fichiers. Il existe actuellement des outils standards et spécialisés sous UNIX qui permettent de travailler sur un grand nombre de fichiers.

5. Projets liés à Modula-2

Un certain nombre de projets sont réalisés ou en cours de réalisation pour évaluer le langage Modula-2.

5.1 Input/Output Tools

Un langage permettant la description de l'interface homme-machine de façon générale a été développé et mis en oeuvre à l'Université de Nimègue. La gestion de multiples entrées/sorties simultanées est possible dans ce langage (BOS83) et les co-programmes de Modula-2 forment la base de sa mise en oeuvre.

5.2 La programmation concurrente

En collaboration avec l'Université de Nimègue un système a été développé pour l'enseignement de la programmation "concurrente" (BEN82). Notamment les concepts de moniteurs, sémaphores, l'échange de messages, et le mécanisme de rendez-vous ADA sont utilisés.

5.3 Projet en médecine

En collaboration avec l'hopital Antoni van Leeuwenhoek à Amsterdam un système de gestion d'une expérience est en cours de réalisation. A part un materiel VME/68000 tout à fait standard une interface histogramme sera dévéloppée.

5.3.1 L'expérience

Un rayon laser passe à travers un tuyau transportant du liquide contenant des cellules. Trois détecteurs sont installés pour mesurer le "temps de vol" des cellules, la diffusion vers l'avant et la diffusion perpendiculaire. Les trois quantités analogiques sont converties en quantités digitales de 6 bits chacune. L'ensemble de 18 bits est présenté comme adresse dans une mémoire de 256 K mots (de 16 bits). Le contenu de l'adresse est incrémenté par l'interface par des accès mémoire directs (DMA).

5.3.2 Le logiciel

Un programme Modula-2 construira une image sur l'écran "bitmap" a partir des données dans la mémoire histogramme. Cette image donnera des indications à l'opérateur pour poursuivre son expérience.

5.4 Projets pour le futur

Afin de pouvoir développer de grands logiciels en Modula-2 il est essentiel que l'environnement de programmation, par exemple celui de UNIX soit amelioré.

Actuellement le développement croisé du logiciel n'est pas au point. Le télé-chargement de code se fait à travers une ligne RS232 à 9600 bauds. Dans l'ordinateur-cible il n'existe qu'un moniteur simple, qui ne permet pas la mise au point symbolique. Pour un dévéloppement sérieux on doit pouvoir disposer d'un système plus sophistiqué.

Les librairies d'entrées/sorties, de fonctions mathématiques etc. sont peu nombreuses et non standardisées. Il est évident que cette standardisation est d'une grande importance pour garantir le succès du langage Modula-2.

6. Conclusions

Le langage Modula-2 permet d'acquérir de l'expérience pour la programmation du futur. Le langage est beaucoup plus simple qu'ADA, mais il en contient les éléments essentiels. En attendant ADA on peut se servir du langage Modula-2 pour le développement de systèmes d'exploitation et de contrôle. Des langages comme Pascal, FORTAN et assembleur sont à éviter pour l'écriture de grands logiciels dans ce domaine.

7. Remerciement

Je remercie Sylviane Szafran du College de France sans l'aide de laquelle cette contribution n'aurait pu être accomplie.

A. References

(BEN82)
Ben-Ari, M.
'Principles of Concurrent Programming'
Prentice Hall, Englewood cliffs, New Jersey

(DIJ68)
Dijkstra, E.W.
'Goto statement considered harmful'
CACM, 11, 3, Mars, 1968, pp. 147-148

(BOS83)
Bos, J. van den; Plasmeijer, M.J.; Hartel, P.H.
'Input-Output Tools: A Language Facility for Interactive and Real-Time Systems'
IEEE Transactions on Software Engineering, SE-9, 3, May 1983, pp. 247-259

(JEN76)
Jensen, K.; Wirth, N.
'Pascal user manual and report'
Springer-Verlag. New York, 1978.

(WIR82)
Wirth, N.
'Programming in Modula-2'
Springer-Verlag, Berlin, 1982.

FORUM SUR LA MICROINFORMATIQUE

EN PHYSIQUE NUCLÉAIRE

ET PHYSIQUE DES PARTICULES

ORGANISE CONJOINTEMENT PAR LE CEA-IRF ET LE CNRS-I.N2.P3

AU LABORATOIRE DE PHYSIQUE CORPUSCULAIRE

COLLEGE DE FRANCE, PARIS

les 19, 20 et 21 septembre 1983

Comité d'organisation :

G. FONTAINE (coordinateur), L.P.C., Collège de France

F.X. GENTIT, D.Ph.P.E., Saclay

R. MARBOT, L.P.N.H.E., Ecole Polytechnique

M. MENCIK, L.A.L., Orsay

A. MULLER, D.Ph.P.E., Saclay