QUELQUES N0TI0NS SUR A D A

Michel CANAL

HISTORIQUE

C'est en 1974 que le département Américain de la défense (D.O.D.) conclut à la nécessité d'améliorer les outils de programmation et la méthodologie: en effet le budget se montait à 3 milliards de dollars et il y avait environ 400 langages et dialectes. A la suite de cette analyse économique le cahier des charges est défini progressivement de 1975 à 1978 (STRAWMAN, WOODENMAN, TINMAN, IRONMAN, STEELMAN). A l'issue d'une première sélection quatre projets de langage furent retenus :

- GREEN (CII Honeywell Bull)
- BLUE (Softech)
- RED (intermetrics)
- YELLOW (SRI)

C'est le Français GREEN qui l'emporta et devint Ada en mai 1979. Ada fut alors soumis à une phase de test et d'évaluation puis à une proposition de norme qui aboutit à la norme ANSI en janvier 1983 et, en 1986, à la norme ISO et à la norme AFNOR (NZ 65700).

L'équipe de développement initiale comprenait cinq Français (ICHBIAH, HELIARD, ROUBINE, ABRIAL et GAILLY), deux Américains (HILFINGER et LEDGARD) quatre Anglais (BARNES, WICHMANN, WOODGER et FIRTH) et un Allemand (KRIEG-BRUCKNER).

CHAMP D'APPLICATION

Ada se voulait un langage général pour 1983 à 20xx:

Un langage algorithmique, modulaire, temps réel et permettant l'écriture de systèmes. I1 n'est pas rare de voir un programme Ada d'un million de lignes, un compilateur Ada, par exemple, fait environ cinq cent mille lignes. On voit donc qu'il est nécessaire que la vérification soit intense aussi bien en compilation qu'en exécution.

En Ada tous les objets sont déclarés (on parle de types ou de sous-types). Ada dispose bien entendu des types numériques et des structures de contrôle modernes.

Les modules ou ensembles de modules (paquetages) sont compilables séparément mais avec des dépendances par rapport aux autres sans qu'il soit nécessaire de faire une édition de lien. Tout sous-programme sera déclaré, ainsi que ses paramètres et son résultat; lors d'une utilisation de ce sous-programme le compilateur vérifiera la concordance des paramètres.

Fiabilité, portabilité, lisibilité et maintenabilité sont les Points forts d'Ada sans pour autant perdre en vitesse ou en puissance.

On trouve actuellement des compilateurs Ada sur IBM PC-AT et compatibles, sur COMPAQ 386, sur poste 68000 (Sun, Appolo, etc.).

Examinons comment se présente la programmation en Ada :

TYPES ET SOUS-TYPES

Tous les objets, qu'ils soient variables ou constants, doivent être déclarés et toute opération sur un objet doit en préserver le type. Voici quelques exemples courants :

- type NOM-de-JOUR is (LUN, MAR, MER, JEU, VEN, SAM, DIM) ;
- type FEU is (ROUGE ,ORANGE ,VERT);
- type ENTIER is range –1000 .. 1000; -- entier de -1000 à +1000
- type GENRE is (M,F);
- type REEL is digits 8; -- réel 8 chiffres significatifs
- type DATE is -- type article
  - record
    - JOUR : INTEGER range 1..31 ;
    - MOIS : STRING(1..10) ;
    - ANNEE : INTEGER range 1900 .. 2100;

- end record ;

– type DATE FIN is new DATE; -- type dérivé
– type TEXTE is access STRING; -- type accès
– type STRING is array (POSITIVE range <>) of CHARACTER;
– type VECTEUR is array (INTEGER range <>) of REAL;
– subtype JOUR _OUVRABLE is NOM_DE_JOUR range LUN..VEN;
– subtype AGE is ENTIER range 0..120;
– subtype LIGNE is STRING (1..40);
– subtype RANGEE is VECTEUR (0..100);

En fait les sous-types permettent de définir des sous-ensembles de types par une contrainte d'intervalle, de précision, de discriminant ou d'indice.

Quand des objets apparaissent de manière imprévisible, quand plusieurs noms se réfèrent au même objet, quand des objets sont liés les uns aux autres il est souvent nécessaire d'utiliser des types ACCES. Voici un exemple avec des dépendances mutuelles :

– type PERSONNE (SEXE : GENRE); -- déclaration incomplète
– type VOITURE; -- déclaration incomplète
– type NOM_DE_PERSONNE is access PERSONNE;
– type NOM_DE_VOITURE is access VOITURE;
– type VOITURE is
  - record
    · NOMBRE : INTEGER;
    · PROPRIETAIRE : NOM-D_E_PERSONNE;
  - end record;
– type PERSONNE(SEXE : GENRE) is
  - record
    · NOM STRING (1..20);
    · NAISSANCE : DATE;
    · AGE : INTEGER range 0..130;
    · VEHICULE : NOM DE VOITURE;
  - case SEXE is

- when M => FEMME : NOM_DE_PERSONNE (SEXE => F);
- when F => MARI : NOM_DE_PERSONNE (SEXE => M);

- end case;

– end record;

Il est même possible d'envisager des types à accès récursif.

OBJETS

DEMAIN : NOM_DE_JOUR;

MILIEU : constant NOM_DE_JOUR := JEU;

DERNIER: constant NOM_DE_JOUR := NOM_DE_JOUR'LAST;

PREMIER: constant NOM_DE_JOUR NOM_DE_JOUR'FIRST;

MICHEL : PERSONNE (M); -- Michel sera toujours du genre M

EPI: TEXTE := new STRING'("Enseignement Public et Informatique");

INSTRUCTIONS

D: DATE;

...

If D.JOUR = 31 and D. MOIS=DEC then

D.JOUR .= 1;

D.MOIS .= JAN;

D.ANNEE := D.ANNEE+1;

end if;

for I in 2..193 loop

– ECRIS_DANS_LE_CADRE (I, LIGNE, COLONNE);
– TAMIS.INPUT (I);

end loop;

Il faut remarquer à propos des boucles qu'il est impossible de modifier le compteur (ici I) pendant la boucle. Il n'est pas nécessaire de

déclarer I et on a l'assurance que I prendra effectivement toutes les valeurs de 2 à 193.

- loop
  - GET (CARACTERE);
  - exit when CARACTERE = '#';
- end loop;
- ECHANGE:
  - declare
    - TEMP : constant INTEGER := V;
  - begin
    - V := U; U := TEMP;
  - end SWAP;

STRUCTURE DES PROGRAMMES

En Ada il y a quatre formes d'unités de programmes : les sous-programmes, les paquetages, les tâches et les unités génériques. Les unités de programmes peuvent être compilées séparément ce qui permet de se créer une bibliothèque. Les unités peuvent être emboîtées comme les structures de bloc d' Algol.

Les SOUS-PROGRANMES sont semblables à ce qu'on trouve en Pascal ou en LSE, on retrouve les deux formes (procédures et fonctions). Un sous-programme Ada se compose d'une partie déclarative et d'un corps. En voici deux exemples classiques :

```
procedure EQUATION (A ,B , C : in REAL;
        RACINE -1   ,RACINE_2 : out REAL;
        OK : out B00LEAN) is
    D: constant REAL := B**2-4.0*A*C;
-   begin
    - if D<0.0 then
      · RACINE_1  := 0.0;
      · RACINE_2 := 0.0;
      · OK := FALSE;
      · return
```

- end if;
- RACINE _1 := (-B+SQRT(D)) / (2.0*A);
- RACINE-2 := (-B-SQRT(D)) / (2.0*A);
- OK := TRUE;
- end EQUATION;

On remarque au passage la nécessité de préciser par "in" et/ou "out" le sens de passage des paramètres. Il n'est pas possible de lire un "out" ou d'écrire un "in".

type VECTEUR is array (INTEGER range <>) of REAL;

function PRODUIT (X, Y : VECTEUR) return REAL is

- TOTAL : REAL :=0.0;

begin

- VERIFIER (X'FIRST=Y'FIRST and X'LAST=Y'LAST);
- for I in X'RANGE loop
  - TOTAL := TOTAL + X(I)*Y(I);
- end loop;
- return TOTAL;

end PRODUIT;

avec les déclarations

- A, B : VECTEUR (1..N);
- R : REAL ;

on peut appeler la fonction ainsi :

- R := PRODUIT (A, B);

Les PAQUETAGES servent à décrire les situations suivantes :

- Collections nommées de déclarations -
- Groupes de sous-programmes
- Types privés

Les paquetages se composent de deux parties, l'une visible et l'autre privée. La partie visible contient la déclaration et certaines spécifications; la partie privée contient le reste des spécifications et le corps du paquetage. La partie visible contient toutes les informations

nécessaires à une autre unité de programme. Il est alors possible de recompiler le corps d'un paquetage sans qu'il soit nécessaire de recompiler toutes les unités qui l'utilisent.

Voici deux exemples de paquetages tirés de la norme ANSI :

```
package TEMPS PASSE                    is
  – type JOUR                  is (L UN, MAR_MER, JEU, VEN, SAM,
    DIM);
  – type TEMPS                 is delta 0.01 range 0.0 .. 24.0;
  – type HORAIRE               is array (JOUR) of TEMPS;
  – HEURES_DE_TRAVAIL : HORAIRE;
  – HEURES_NORMALES : constant HORAIRE :=
    - (LUN..MAR|JEU..VEN => 8.0, MER|SAM => 3.5 DIM => 0.0);
end TEMPS PASSE;

package NOMBRES_RATIONNELS is
  – type RATIONNEL is
    - record
      · NUMÉRATEUR : INTEGER;
      · DÉNOMINATEUR: POSITIVE;
    - end record;
function ÉGAL          (X, Y : RATIONNEL)    return BOOLEAN;
function "/"           (X, Y : INTEGER)      return RATIONNEL;
function "+"           (X, Y : RATIONNEL)    return RATIONNEL;
function "-"           (X, Y : RATIONNEL)    return RATIONNEL;
function "*"           (X, Y : RATIONNEL)    return RATIONNEL;
function "/"           (X, Y : RATIONNEL)    return RATIONNEL;
end;

package body NOMBRES RATIONNELS is
  – procedure MEME DÉNOMINATEUR (X, Y : in out RATIONNEL)
    is begin
    - -- réduction au même dénominateur;
    - ...
```

- end;••

function EGAL(X, Y : RATIONNEL) return BOOLEAN is U, V : RATIONNEL;

begin

- U .= X;
- V := Y;
- MEME_DENOMINATEUR (U, V);
- return U.NUMERATEUR = V.NUMERATEUR;

end ÉGAL;

function "/" (X,Y : INTEGER) return RATIONNEL is

- -- cette fonction permet de construire les rationnels
- -- à partir de deux entiers

begin

- if Y>0 then
  - return (NUMÉRATEUR => X, DÉNOMINATEUR => Y);
- else
  - return (NUMÉRATEUR => -X, DÉNOMINATEUR => -Y);
- end If;

end "/";

function "+" (X,Y: RATIONNEL) return RATIONNEL is ... end "+";

function "-" (X,Y: RATIONNEL) return RATIONNEL is... end

function "*" (X,Y: RATIONNEL) return RATIONNEL is... end "*";

function "/" (X,Y: RATIONNEL) return RATIONNEL is... end "/";

end NOMBRES RATIONNELS;

Vous avez dans ce paquetage un exemple de surcharge d'opérateurs; les opérateurs ainsi redéfinis cacheront les prédéfinis. Le compilateur distingue les opérateurs à utiliser grâce aux types des paramètres, il en est de même pour les sous-programmes, par exemple, le sous-programme ÉCRIRE ne sera pas le même suivant le type du paramètre à écrire.

Voici un autre exemple pour illustrer les TYPES PRIVES :

```
package SERRURIER is
   – type CLEF is limited private;
   – procedure ATTRIBUER (CLÉ: out CLEF);
private
   – type CLEF is access PORTE;
end SERRURIER;
   – -- la seule opération utilisable par les utilisateurs de CLEF
   – -- est : ATTRIBUER
   – -- même si CLEF est un type accès (comme ici),
   – -- il n'y a pas de new...
package body SERRURIER is
   – -- par contre, dans l'implémentation du paquetage
   – -- on sait que CLEF est un accès
   – CLEF COURANTE: CLEF := new PORTE;
   – ...
end;• •
```

Un sous-programme Ada est une unité de programme avec paramétrage à l'exécution. Une UNITÉ GENERIQUE Ada est aussi une unité de programme mais avec paramétrage la compilation. Les unités génériques sont nécessaires car certaines formes de paramètres (types, sous-programmes) doivent être vérifiées à la compilation. Les unités génériques permettent d'éviter les redondances inutiles, souvent sources d'erreurs, dans l'écriture des programmes. Ainsi, grâce au concept de généricité, Ada est le premier langage permettant l'écriture de composants logiciels à l'échelle industrielle. Par exemple, les problèmes de tri entrent parfaitement dans ce cadre. Dans le principe, l'algorithme de tri est le même, quel que soit le type d'objet à trier. On utilise en Ada un paquetage générique auquel on précise le type des éléments à trier et la relation d'ordre entre ces éléments. On évite ainsi de réécrire plusieurs procédures identiques qui ne diffèrent que par le type des éléments à trier.

Comme dans l'exemple suivant :

```
procedure ÉCHANGE (GAUCHE, DROIT : in out NOM) is
      AVANT_A_GAUCHE : constant NOM := GAUCHE;
```

```
begin
   – GAUCHE := DROITE;
   – DROITE := AVANT_A_GAUCHE;
end;
procedure ÉCHANGE (GAUCHE, DROITE : in out CODE) is
…
end;
procedure ÉCHANGE (GAUCHE, DROITE : in out EMPLOI) is
…
end;
```

La logique de la procédure ne dépend pas du type de choses échangées. On écrira alors

```
generic
   – type ITEM is private;
procedure ÉCHANGE (GAUCHE, DROITE : in out ITEM);
   – -- et le corps générique
procedure ECHANGE (GAUCHE, DROITE : in out ITEM) is
   – AVANT_A_GAUCHE : constant ITEM := GAUCHE;
begin
   – GAUCHE := DROITE;
   – DROITE := AVANT A GAUCHE;
end;
   – -- ITEM est un type formel générique. puis on écrira:
procedure TROC is nev ECHANGE (INTEGER);
procedure TROC is new ECHANGE (COULEUR);
procedure TROC is new ECHANGE (ITEM => CODE);
procedure TROC is nev ECHANGE (CODE)
   – -- trois procédures de TROC sont surchargées.
      - procedure TROC (GAUCHE, DROIT : in out CODE);
```

- ...
- -- et maintenant

TROC (TEINTE A, TEINTE B); -- pour les couleurs

TROC (DROITE => TON _CODE, GAUCHE => MON CODE); -- pour les codes

-- remarquez ici cette utilisation très commode en Ada lorsqu'on

-- a oublié l'ordre des paramètres, mais ni leurs noms, ni leurs

-- types

-- etc.

Enfin les TACHES, quatrième forme d'unité de programme, permettent l'exécution en parallèle de plusieurs processus. Il existe aussi des objets de type tâche qui s'exécutent en parallèle; le type tâche est un type limité (ni affectation, ni égalité) mais il peut être passé en paramètre. Voici un exemple de type tâche:

task type CONTOLEUR_VISU is

- entry LIRE (C: out CHARACTER);
- entry ECRIRE (C: in CHARACTER);

end;

LE TIEN,LE MIEN : CONTROLEUR VISU;

On aura une tâche pour chacun avec un comportement identique.

Voici maintenant un exemple de traitement parallèle:

task DISTRIBUTEUR is

- entry PREND (CETTE LIGNE : in LIGNE);
- entry REND (UN CARACTERE : out CHARACTER);

end DISTRIBUTEUR;

task body DISTRIBUTEUR is

```
TAMPON : LIGNE;
```

begin

- loop
  - accept PREND (CETTE_LIGNE : in LIGNE) do

```
        · TAMPON := CETTE_LIGNE;
    - end PREND;
    - for INDICE in TAMPON'RANGE loop
        · accept REND (UN_CARACTERE : out, CHARACTER) do
                UN_CARACTERE := TAMPON (INDICE);
        · end;
    - end loop;
  – end loop;
end DISTRIBUTEUR;
```

Ce distributeur peut être appelé par d'autres tâches, certaines lui fournissent des lignes :

PREND (CETTE LIGNE => "Information pour les autres tâches."); d'autres tâches consomment les caractères en appelant l'entrée REND. I1 y a une file d'attente par entrée. Chaque instruction accept permet un rendez-vous entre la tâche qui l'exécute et la première de la file. L'instruction accept est exécutée en commun puis les tâches s'exécutent à nouveau indépendamment en parallèle.

ENTREES ET SORTIES

Il est difficile de parler d'un langage sans parler des entrées-sorties. En Ada, elles sont définies en utilisant des paquetages génériques, des sous-programmes homographes ou des sous-programmes avec des paramètres par défaut; donc, par extension, sans traits spécifiques. Cela évite les problèmes rencontrés dans les langages ayant des traits spécifiques comme FORTRAN ou PASCAL ou qui ont moins de fonctionnalités comme ALGOL60. En fait Ada comporte deux paquetages génériques pour les fichiers binaires SEQUENTIAL_I0 et DIRECT I0 et un paquetage pour les fichiers de textes TEXT_I0. Ce dernier contient quatre paquetages génériques: INTEGER_I0, FLOAT_I0, FIXED_IO et ENUMERATION_I0.

Voici, à titre d'exemple, un petit programme qui affiche tous les fichiers d'un disque MS-DOS, répertoire par répertoire :

```
with TEXT_IO,DOS;

use DOS; --  paquetage d'interfaçage avec MS-DOS
```

```
procedure FILES is
  – procedure LIST SUBTREES( TREE : STRING) is
    - S : SEARCH_RECORD;
    - OK: BOOLEAN;
  – begin
    - FIND_FIRST(
      · TREE &
        "\*.*",S,OK,ATTR_DIREC+ATTR_SYSTEM+ATTR_HIDDEN)
        ;
    - while OK loop
      · if S. FILE_NAMX(1)/='.' then -- avoid . and ..
      · declare
      · F: constant STRING := TREE & '\'
                                  & GET_STRING (S.FILE NAMX);
  – begin
    - if S.FILE_ATTR = ATTR_DIREC then
      · LIST_SUBTREES (F);
    - else
      · TEXT_IO.PUT_LINE (F);
    - end if;
  – end;
  – end if;
  – FIND_NEXT (S, OK);
  – end loop;
  – end;
  – begin
    - LIST SUBTREES(GET-PARMS);
  – exception
    - when DOS ERROR => TEXT_IO.PUT_LINE("Directory not
      found.");
  – end;
```

EXCEPTION

On trouve en Ada un traitement des erreurs comparable au "ON ERROR GOTO" du BASIC. En effet, certaines situations empêchent l'accomplissement normal d'une action, la violation d'une contrainte, les débordements, la division par zéro ou la singularité d'une matrice, par exemple. Une exception est un nom associé à une telle situation. Lever une exception c'est signaler qu'une telle situation est survenue et traiter une exception c'est y répondre par l'exécution d'actions appropriées. Certaines exceptions sont d'ailleurs prédéfinies dans le paquetage STANDARD: CONTRAINT ERROR, NUMERIC *ERROR, P*ROGRAM ERROR, STORAGE_ERROR et TASKING FRROR et peuvent être levées implicitement par Ada. Lorsqu'une exception est levée dans une suite d'instructions , l'exécution de la suite est abandonnée; s'il y a un traite-exception, pour cette exception, l'exécution se poursuit par celle des instructions du traite-exception sinon l'exception est propagée au niveau d'appel dans le cas de blocs, de sous-programmes ou de paquetages, mais pas dans le cas de tâches qui ont un traitement particulier. Ce mécanisme donne l'assurance qu'on ne détruit pas la structuration du programme (on ne risque pas de revenir n'importe où, comme cela est possible en BASIC) et cela fait un chemin d'exécution plus clair. Exemple :

```
begin
   – -- séquence d'instructions
exception
   – when SINGULAR 1 NUMERIC ERROR => PUT("Matrice singulière");
   – when others => PUT(" Erreur fatale");
   – raise ERROR;
end;
```

ou encore:

```
...
loop
   –    begin
      - NEW LINE;
      - PUT("  Jour: ");
```

- GET(DATE.JOUR);
- exit; -- non exécutée si une exception est levée

– exception

- when others =>
  - -- récupération de l'exception levée
  - -- si le jour saisi n'est pas correct
  - PUT LINE (" Erreur : jour non compris entre 1 et 31");
  - SKIP LINE;
  - -- et on reste dans la boucle, on va donc relire le jour.
- end;

end loop;

...

On voit dans cet exemple que la vérification d'une entrée se fait très facilement grâce aux exceptions.

EN GUISE DE CONCLUSION

Nous espérons que ces quelques pages ont pu vous donner envie d'en savoir plus sur Ada. Voici une petite liste d'ouvrages qui peuvent vous y aider:

Ada: Une introduction avancée par N. Gehany	Chez Eyrolles
Manuel de Référence Ada	Chez Alsys
Ada:une introduction de H. Ledgard	Chez Masson
Programmez en Ada de J.G.P. Barnes	en cours de traduction

et en anglais un livre rempli d'exemples très clairs :

Rationale for the design of the Ada Programming language (Alsys).

Vous pourrez également en savoir plus auprès d'ALSYS, société française créée en 1980 par Jean Ichbiah, responsable de l'équipe de définition du langage Ada.

ALSYS (Applications Logicieles et SYStèmes) a ses bureaux au 29 Av. de Versailles, La Châtaigneraie, 78170 LA CELLE ST CLOUD

C'est grâce à elle, à un séminaire Ada que nous y avons suivi, et à la collaboration d'un de ses experts (Henri Dancy, ancien élève du lycée Corneille à La Celle St Cloud) que cet article a pu se faire. La plupart des exemples ci-dessus ont été compilés et testés sur un IBM-AT de la société ALSYS.

I1 existe à ce jour 101 compilateurs Ada dont la conformité à la norme a été vérifiée par le passage des 3000 tests de validation du D. O. D., ce qui leur assure une compatibilité inégalée.

En dernière nouvelle, il semblerait que la société Meridian Software System ait écrit un compilateur Ada pour IBM-PC XT mais nous ignorons s'il est commercialisé en France.

Pour terminer cet article nous vous proposons un exemple complet de programme Ada. Il nous a été présenté au cours du séminaire Ada par M. Pascal Plisson. L'exécution de ce programme donne une démonstration très visuelle du multi-tâche.

Michel CANAL

EXEMPLE DE TACHES PARALLÈLES

Ce programme est une démonstration de l'utilisation des tâches parallèles dans le langage Ada.

Cet exemple calcule les 44 premiers nombres premiers suivant l'algorithme de KAHN (cf. G. KAHN, D. Mac Queen "Coroutines and Networks of parallel Processes", INRIA).

Elle met en avant la facilité de programmation en Ada des processus parallèles et de leurs interactions, notamment :

- * les tâches,
- * les types tâches,
- * la communication par rendez-vous.

DESCRIPTION DE L'ALGORITHME

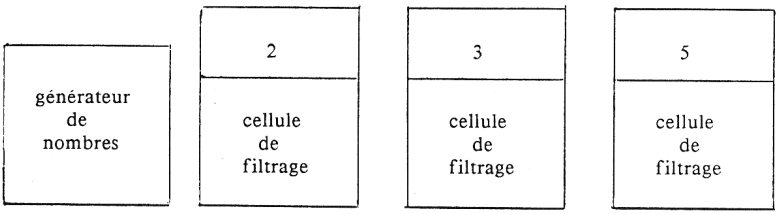
Un générateur de nombres génère des nombres compris entre 2 et N. Chaque nombre généré est examiné par un filtre qui laisse passer les nombres premiers et élimine ceux qui ne le sont pas.

Le filtre est composé de plusieurs cellules. Chaque cellule contient un diviseur pour tester si le nombre reçu est divisible par le diviseur ou non. Si c'est le cas, le nombre n'est pas premier (puisqu'il est divisible) et alors il est abandonné, Le filtre est constitué de plusieurs cellules disposées en séries, au travers desquelles se propage le nombre généré. S'il arrive en bout de chaîne, il est premier et il sert alors de diviseur à une nouvelle cellule de filtrage dont le diviseur est N.

Programme

Le générateur de nombres et les cellules de filtrage sont programmés par des tâches. Ainsi nous créons un réseau de processus qui évolue dynamiquement. En fin de démonstration, il y a 46 taches concurrentes qui communiquent entre elles.

5 n'est pas divisible par 3. Comme il n'y a pas d'autre cellule de filtrage après CF3, une nouvelle cellule est créée dont le diviseur est 5.



Toutes les tâches travaillent en parallèle.

Programme de démonstration

L'écran visualise le travail de chaque tâche et le cheminement de chaque nombre depuis le générateur de nombres jusqu'à la sortie du filtre.

On remarquera l'efficacité du programme, sa simplicité d'écriture et sa concision (41 instructions).

Exemple

Au départ nous avons :

Le générateur de nombres fournit 2.

2 est communiqué (par rendez-vous) à la cellule de filtrage dont le diviseur est 2 (CF2).

2 est divisible par 2, 2 n'est pas premier, il est abandonné.

Le générateur de .nombres fournit 3.

3 est communiqué à CF2.

3 n'est pas divisible par 2. Comme il n'y a pas d'autre cellule de filtrage après CF2, une nouvelle cellule est créée (création d'une tache), dont le diviseur est 3 (CF3).

Le générateur de nombres fournit 4.

4 est communiqué à CF2.

4 est divisible par 2, 4 n'est pas premier, il est abandonné.

Le générateur de nombres fournit 5.

5 est communiqué à CF2.

5 n'est pas divisible par 2, il est transféré à CF3.

– with CADRE;

```
  – use CADRE;
procedure NOMBRE PREMIER is
  – task GENERATEUR_DE_NOMBRES is
  – end GENERATEUR DE NOMBRES;
  – task type FILTRE is
    - entry INPUT (NOMBRE: in INTEGER);
  – end FILTRE;
  – type BUS is access FILTRE;
  – function CREE UNE TACHE(DIVISEUR:INTEGER) return BUS
    is
    - TAMIS: BUS;
  – begin
    - TAMIS := new FILTRE;
    - TAMIS.INPUT(DIVISEUR);
    - return TAMIS;
  – end CREE UNE TACHE;
    - -- La tâche GENERATEUR_DE_NOMBRES:
    - -- crée la première tâche FILTRE et
    - --  lui transfère séquentiellement les nombres de 2 à 193.
task body GENERATEUR_DE_NOMBRES is
  – LIGNE, COLONNE        INTEGER;
  – TAMIS  : BUS := nuit;
begin
  – CREE _UN_CADRE("N", LIGNE, COLONNE);
  – TAMIS := CREE_UNE TACHE(2);
  – for I in 2 .. 193 loop
    - ECRIS_DANS_LE_CADRE(I, LIGNE, COLONNE);
    - TAMIS.INPUT(I);
  – end loop;
  – EFFACE_INTERIEUR_DU_CADRE(LIGNE, COLONNE);
end GENERATEUR_DE_NOMBRES;
    -- La tâche FILTRE:
```

```
– -- Contient un diviseur (DIVISEUR) qui est premier;
– -- elle reçoit des nombres.
– -- Si le nombre reçu est divisible par DIVISEUR,
– -- alors ce nombre n'est pas un nombre premier, il est abandonné.
– -- Sinon elle le transfère au filtre suivant.
– -- Si ce filtre n'existe pas alors le nombre est premier
– -- elle crée un nouveau filtre qui a ce nombre comme diviseur.
task body FILTRE is
– DIVISEUR        : INTEGER := 0;
– X,Y,l      : INTEGER ;
– SUIVANT         :BUS    := null;
begin
– accept INPUT(NOMBRE : in INTEGER) do
  - DIVISEUR := NOMBRE;
  - CREE_UN_CADRE(DIVISEUR, Y, X);
– end INPUT;
– loop
  - select
    · accept INPUT(NOMBRE : in INTEGER) do
    ·    I:= NOMBRE
    · end INPUT;
    · ECRIS_DANS_LE_CADRE(I, Y, X);
    · if I mod DIVISEUR /= 0 then
     -  if SUIVANT = null then
            SUIVANT := CRÉE_UNE_TACHE(I);
  - else
            SUIVANT.INPUT(I);
  - end if;
– end if;
– EFFACE_INTERIEUR_DU_CADRE(Y, X);
  or
  - terminate;
```

```
   – end select;
     end loop;
   end FILTRE;
begin
   – null;
end NOMBRE_PREMIER;
```