Version 1.0: août 1988.

SENTIERS

Un logiciel Modula-2 pour l'arithmétique sur les grands entiers

Pierre L'Ecuyer, Gaétan Perron et François Blouin

Département d'informatique
Université Laval

Résumé

SENTIERS (un acronyme pour "Super Entiers") est un ensemble de modules écrits en Modula-2, permettant de travailler avec des entiers de taille arbitrairement grande, sans perte de précision. Ces modules précompilés fournissent des types de données prédéfinis et des procédures que l'on peut utiliser à partir d'un programme Modula-2. Ces outils permettent de créer de grands entiers, d'effectuer des entrées-sorties et des opérations arithmétiques de toutes sortes, de factoriser ou de tester la primalité, de générer des grands entiers au hasard, etc. Présentement, SENTIERS est implanté sous VAX/VMS, MS-DOS et SUN/OS.

Table des matières

1 Introduction 2

2 Exemples de programmes utilisateurs 4

2.1 Calcul de N factoriel 4

2.2 Multiplication par "diviser-pour-régner" 4

2.3 Quelques exemples de temps d'exécution 7

3 Implantation 9

3.1 Organisation des modules 9

3.2 Gestion de la mémoire 9

3.3 Problèmes de visibilité 10

3.4 Pourquoi pas une approche fonctionnelle ? 11

ANNEXES 13

A. Définition fonctionnelle des modules de SENTIERS 13

SUPINT 14

SUPCONV 17

SUPARITH 19

SUPRAND 23

SUPFACT 24

B. Guide d'utilisation sur VAX/VMS. 26

C. Guide d'utilisation sous MS-DOS avec TopSpeed Modula-2. 27

D. Guide d'utilisation sur SUN avec SUN/Modula-2. 27

Références 28

1 Introduction

SENTIERS est un logiciel permettant d'effectuer des opérations arithmétiques de base, des comparaisons, des entrées/sorties, etc. pour des entiers de taille arbitrairement grande, sans perte de précision. Il est constitué d'un ensemble de modules précompilés, écrits en Modula-2 [7, 8, 16], et qui fournissent des types de données prédéfinis et des procédures que l'on peut utiliser à partir d'un autre programme Modula-2.

Les grands entiers sont représentés dans des vecteurs de chiffres en base $b = 2^{(\beta-2)/2}$, où β est une constante dépendant de l'implantation, et qui correspond habituellement au nombre de bits dans un mot sur l'ordinateur utilisé. Par exemple, pour une machine qui représente les entiers sur 32 bits, on prendra $b = 2^{15}$.

Le module `SUPINT` fournit les outils de base pour créer des grands entiers, les détruire, effectuer des affectations et des entrées/sorties. Il s'occupe aussi de gérer l'utilisation de la mémoire pour les représenter. Chaque variable représentant un grand entier doit être déclarée de type `SuperInteger`, puis initialisée (avant sa première utilisation) en appelant la procédure `Create`. Lors des opérations (arithmétiques ou autres), le système s'occupe d'agrandir au besoin (automatiquement) la taille des vecteurs de chiffres utilisés, de façon à ce qu'il n'y ait jamais de débordement (du moins tant qu'il reste de la mémoire disponible).

Le module `SUPCONV` permet d'effectuer des conversions entre des variables de type `SuperInteger` et des variables ordinaires représentées en entier ou en point flottant.

`SUPARITH` est un module servant à effectuer des opérations arithmétiques de base et des comparaisons sur les grands entiers. Des fonctions booléennes permettent de comparer deux grands entiers, ou de savoir si un grand entier est négatif, positif ou nul. On peut changer le signe ou prendre la valeur absolue d'un grand entier, additionner, soustraire, multiplier ou diviser deux grands entiers. On peut aussi élever un grand entier à une puissance entière (positive), multiplier deux grands entiers ou élever un grand entier à une puissance dans l'arithmétique modulo un grand entier (ou un entier), ou calculer le plus grand commun diviseur de deux grands entiers. Des procédures spéciales permettent d'améliorer l'efficacité lorsqu'on multiplie ou divise par une puissance de la base, ou lorsqu'on fait un calcul modulo une puissance de la base.

Enfin, `SUPRAND` permet de générer au hasard des grands entiers, selon une loi uniforme discrète entre deux bornes. `SUPFACT` permet de tester la primalité d'un grand entier, ou de le décomposer en facteurs premiers.

Les algorithmes de base pour effectuer des opérations sur des entiers de taille arbitrairement grande sont décrits en détails dans Knuth [9]. Collins [5] décrit un système permettant de manipuler des polynômes avec des coefficients entiers de taille arbitrairement grande, avec des techniques de gestion dynamique de la mémoire utilisant des listes chaînées. A noter ici qu'en se basant sur les présents modules de SENTIERS, on pourrait facilement ajouter un module permettant d'effectuer des opérations sur les polynômes. Certains outils sont aussi disponibles sous UNIX [4, 13] pour effectuer des calculs avec des entiers arbitrairement grands, en mode interactif. Brent [3] décrit par ailleurs une librairie FORTRAN (dont le

code source est disponible) offrant des outils pour effectuer un vaste répertoire d'opérations sur des nombres représentés en point flottant, avec une précision arbitrairement grande.

SENTIERS est basé sur une philosophie procédurale. L'objectif était d'avoir un ensemble d'outils efficaces et portables, utilisables à partir de programmes écrits en langage de haut niveau. Nous ne voulions pas un progiciel interactif. La rapidité d'exécution était un critère primordial, car pour nos applications, ces procédures devaient être utilisées pour consommer des centaines d'heures de CPU, en traitement par lots. A noter par ailleurs que l'on pourrait très bien (et assez facilement) implanter un progiciel interactif en utilisant les outils fournis par SENTIERS. Dans ce cas, il serait probablement avantageux d'améliorer la performance des procédures d'entrée/sortie, pour lesquelles nous n'avons mis que le minimum d'effort.

En principe, une approche fonctionnelle avec une notation infixe aurait fourni une interface plus intéressante. Pour effectuer une opération sur `A` et `B` et placer le résultat dans `C`, on aurait pu faire "`C := A Oper B`" au lieu de "`Oper (A, B, C)`", ce qui aurait permis de composer les opérations comme on le fait habituellement. Mais malheureusement, Modula-2 ne nous permet pas d'implanter les choses ainsi de façon suffisamment efficace et sécuritaire. Nous verrons pourquoi à la section 3. Le choix de Modula-2 comme langage de base s'est fait surtout parce qu'au moment du développement, ce langage était sans doute le plus intéressant parmi ceux produisant du code efficace et pour lesquels un compilateur était disponible sur les machines que nous utilisions. Nous aurions peut-être pu utilisé ADA si des compilateurs avaient été davantage disponibles. Un jour, on fera peut-être une version ADA. Dans ce langage on pourrait, par exemple, définir les opérations arithmétiques en utilisant une notation infixe et les opérateurs habituels.

Dans la prochaine section, on présente quelques exemples dont l'objectif est d'illustrer l'utilisation de SENTIERS. Dans la section 3, on discute de l'implantation. La définition fonctionnelle de chacun des modules est donnée à l'annexe A, et les modes d'utilisation sur VAX/VMS, MS-DOS, et SUN sont donnés aux annexes B, C et D, respectivement. A noter que l'utilisation de ces modules doit se faire à partir d'un programme Modula-2, et nécessite donc un compilateur Modula-2.

2 Exemples de programmes utilisateurs

Nous présentons dans cette section deux programmes Modula-2 utilisant SENTIERS. L'objectif est de familiariser le lecteur avec les outils disponibles et d'illustrer leur utilisation. En examinant ces exemples, on pourra se référer au besoin à l'annexe A.

2.1 Calcul de N factoriel

Le programme de la figure 1 demande à l'usager un entier N, le lit puis calcule et imprime N factoriel. On utilise trois grands entiers : l'un contient la constante 1, un autre contient la valeur de N, et le dernier est un tampon contenant la valeur courante du produit. Les trois variables correspondantes sont déclarées dans l'entête, et les grands entiers sont créés au début du programme, avant leur première utilisation. On initialise ensuite `Un` à 1, on lit la valeur de N en base 10 et si cette dernière valeur est négative, un message d'erreur est imprimé à l'écran. Sinon, on calcule N factoriel. A chaque itération de la boucle, on diminue N de 1 et on multiplie la valeur du tampon par la valeur courante de N. Finalement, on demande à l'usager dans quelle base il veut le résultat et celui-ci est alors imprimé dans la base voulue, 70 chiffres par ligne.

2.2 Multiplication par "diviser-pour-régner"

Le module de la figure 2 contient une procédure qui implante la technique "diviser-pour-régner" pour la multiplication des grands entiers. Cette technique est décrite à la section 4.7 de [2]. Elle prend un temps dans l'ordre de $n^{1.59}$ pour multiplier deux entiers de taille n (i.e. de n chiffres), comparativement à l'algorithme classique, utilisé dans SENTIERS, qui prend un temps dans l'ordre exact de n^2.

Soient u et v deux grands entiers de n chiffres (en base b), à multiplier. On peut séparer chacun des deux opérandes en deux sections à peu près égales : $u = b^s w + x$ et $v = b^s y + z$, où $s = \lfloor n/2 \rfloor$, $0 \leq x < b^s$ et $0 \leq z < b^s$. Aucun des entiers w, x, y, z n'a plus de $\lceil n/2 \rceil$ chiffres. Le produit de u par v peut s'écrire :

$$uv = b^{2s} wy + b^s (wz + xy) + xz$$

et on peut utiliser cette méthode récursivement pour calculer les quatre produits de l'expression de droite. Cependant, une astuce permet de réduire de quatre à trois le nombre d'appels récursifs, et c'est ce qui permet de passer de $O(n^2)$ à $O(n^{1.59})$. Il suffit en fait de remarquer que $wz + xy = (w + x)(y + z) - wy - xz$, de sorte que le produit uv peut s'évaluer comme suit :

$$\begin{aligned} &p := wy; \quad q := xz; \quad r := (w + x)(y + z); \\ &uv := b^{2s} p + b^s (r - p - q) + q. \end{aligned}$$

```
MODULE nfact;

FROM MYINOUT  IMPORT WriteLn, WriteString, ReadCard, ReadLn;
FROM SUPINT   IMPORT SuperInteger, Create, ReadSup, WriteSup, Affect;
FROM SUPCONV  IMPORT IntToSup;
FROM SUPARITH IMPORT Negative, Equal, Greater, Subtract, Multiply;

(* Ce programme calcule  N factoriel. *)
(* Mode interactif avec l'usager. *)

VAR
   N, Un, Tampon : SuperInteger;
   Base : CARDINAL;

BEGIN
Create (Un, 2);   Create (N, 2);   Create (Tampon, 10);
IntToSup (1, Un);
WriteString (' Calcul de N factoriel. Valeur de N ?');   WriteLn;
ReadSup (N, 10); ReadLn;
IF Negative (N) THEN
   WriteString (' ERREUR : N est negatif');   WriteLn
ELSIF NOT Greater (N, Un) THEN
   Affect (Un, Tampon)
ELSE
   Affect (N, Tampon);   Subtract (N, Un, N);
   WHILE NOT Equal (Un, N) DO
      Multiply (N, Tampon, Tampon);
      Subtract (N, Un, N)
      END
   END;
WriteString (' Vous le voulez dans quelle base ? > ');
ReadCard (Base); ReadLn; WriteLn;
WriteSup (Tampon, Base, 70);   WriteLn
END nfact.
```

Figure 1: Calcul et impression de N factoriel.

```
DEFINITION MODULE gmult;

FROM SUPINT IMPORT SuperInteger;

PROCEDURE Mult (U, V, C : SuperInteger);
   (* Multiplie U par V et place le résultat dans C.  *)

END gmult.
```

Figure 2: Multiplication par “diviser-pour-régner” (définition).

```
IMPLEMENTATION MODULE gmult;

FROM SUPINT   IMPORT SuperInteger, Create, Delete, NumDigits;
FROM SUPARITH IMPORT Add, Subtract, Multiply, ShiftRight, ShiftLeft, CutLeft;

CONST
   T  = 8;  (* Les grands entiers créés occupent initialement 256 mots. *)
   n0 = 32; (* Seuil d'arrêt de la récursivité.                         *)

PROCEDURE Mult (U, V, C : SuperInteger);
   VAR
      s : INTEGER;
      W, X, Y, Z, P, Q, R : SuperInteger;
   BEGIN
   IF NumDigits (U) > NumDigits (V) THEN
      s := NumDigits (U) DIV 2
   ELSE
      s := NumDigits (V) DIV 2
      END;
   IF s <= n0 THEN
      Multiply (U, V, C)
   ELSE
      Create (W, T);  Create (X, T);  Create (Y, T);  Create (Z, T);
      Create (P, T);  Create (Q, T);  Create (R, T);
      ShiftRight (U, s, W);   ShiftRight (V, s, Y);   Mult (W, Y, P);
      CutLeft (U, s, X);      CutLeft (V, s, Z);      Mult (X, Z, Q);
      Add (W, X, X);    Add (Y, Z, Z);
      Delete (W);       Delete (Y);
      Mult (X, Z, R);   Subtract (R, P, R);   Subtract (R, Q, R);
      ShiftLeft (P, s + s, P);   ShiftLeft (R, s, R);
      Add (P, R, C);    Add (C, Q, C);
      Delete (X);   Delete (Z);   Delete (P);   Delete (Q);   Delete (R)
      END
   END Mult;

END gmult.
```

Figure 3: Multiplication par “diviser-pour-régner” (implantation).

Il suffit de trois multiplications pour calculer p, q et r, et les multiplications par b^s et b^{2s} se font en temps linéaire simplement par des décalages.

En pratique, cet algorithme ne devient intéressant que lorsque n est grand. Lorsque la taille des opérandes est suffisamment petite, il devient plus rapide d'appeler directement la procédure `Multiply` de `SUPARITH` et on le fait. Sinon, on décompose `U` et `V` tel que suggéré,

on calcule `P`, `Q` et `R` en utilisant trois appels récursifs, et on calcule le produit. A noter qu'à cause des appels récursifs, on doit à chaque appel créer les grands entiers servant de variables temporaires, puis les détruire afin que l'espace qu'ils occupent soit récupéré.

2.3 Quelques exemples de temps d'exécution

Nous avons effectué quelques expériences empiriques afin d'évaluer (grossièrement) le temps d'exécution de quelques procédures de SENTIERS, en fonction de la taille des opérandes. Pour chaque procédure et chaque taille, nous avons répété r fois l'expérience suivante: nous avons généré au hasard $k+1$ ensembles d'opérandes, selon une loi uniforme parmi les opérandes de la taille appropriée, et calculé le temps moyen par appel, en excluant le premier appel. Nous avons exclu ce premier appel parce qu'il est en général plus coûteux que les autres, puisqu'il sert à réserver de la mémoire et à créer des structures de données qui seront réutilisées au cours des appels subséquents. Le temps "d'overhead" induit par la génération des valeurs aléatoires, le contrôle des boucles, etc., fut estimé en calculant le temps d'exécution des mêmes programmes sans les appels de procédures, puis soustrait. Les résultats des r répétitions (indépendantes) de l'expérience ont servi à calculer un intervalle de confiance sur le temps moyen par appel, pour une taille d'opérandes donnée.

Ici, la taille d'un opérande est le nombre de chiffres *décimaux* requis pour le représenter. `Add` additionne deux entiers positifs de taille n pour obtenir un entier de taille n ou $n+1$, `Multiply` multiplie deux entiers de taille n pour obtenir un entier de taille $2n$ ou $2n-1$, `Mult` fait de même, mais en utilisant la procédure de la figure 3, `Divide` divise un entier de taille $2n$ par un entier de taille n, `GCD` calcule le plus grand commun diviseur de deux entiers de taille n, et `SupRandom` génère un entier selon la loi uniforme discrète entre 1 et n.

Les valeurs données au tableau 1 correspondent aux temps observés (moyenne par appel, en secondes, sous forme d'un intervalle de confiance à 90%) sur un VAX-11/780, en utilisant la version 3.1-9 du compilateur Modula-2 de Logitech [6], sous la version 4.6 de VMS. Les expériences ont été faites avec $r = 10$ répétitions, $k = 100$ pour $n = 100$, $k = 20$ pour $n = 500$ et $n = 2000$, et $k = 4$ pour $n = 10000$. Nous avons répété les mêmes expériences sur un Toshiba muni d'un processeur 80386 à 16MHz, sous MS-DOS et avec le compilateur TopSpeed Modula-2 (tableau 2), puis sur un Sparc-2 de SUN, avec le compilateur SUN–Modula-2 (tableau 3). Comme on peut le voir, les résultats furent similaires. Dans tous les cas, la base utilisée était de $b = 2^{15}$.

n	100	500	2000	10000
Add	.0006±.0001	.005±.001	.012±.001	.11±.01
Multiply	.0222±.0002	.505±.004	7.996±.021	206.20±.48
Mult	.0222±.0004	.422±.002	4.151±.012	53.11±.20
Divide	.0406±.0002	.804±.003	12.226±.007	307.50±.33
GCD	.1894±.0020	3.320±.020	48.207±.055	1249.43±2.40
SupRandom	.0052±.0003	.022±.001	.0816±.001	.43±.01

Tableau 1: Estimation du temps moyen par appel (sec.) sur VAX-11/780

n	100	500	2000
Add	.0004±.0002	.0035±.0007	.0040±.0009
Multiply	.0262±.0001	.6076±.0007	9.6289±.0008
Mult	.0262±.0002	.4738±.0008	4.3769±.0010
Divide	.0347±.0001	.6666±.0008	10.0746±.0008
GCD	.1580±.0004	2.8876±.0052	42.9148±.0478
SupRandom	.0039±2.E-5	.0191±.0004	.0663±.0007

Tableau 2: Estimation du temps moyen par appel (sec.) sous MS-DOS (80386 à 16Mhz)

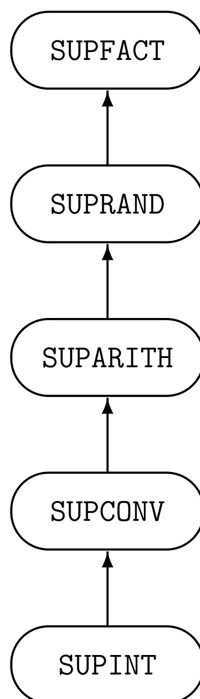
n	100	500	2000	10000
Add	.0002±2.1E-5	.0006±.0001	.0020±.0001	.02±.01
Multiply	.0051±2.7E-5	.1191±.0002	1.8973±.0010	89.65±.04
Mult	.0051±2.8E-5	.0979±.0001	.9742±.0009	13.39±.01
Divide	.0083±3.9E-5	.1647±.0003	2.5838±.0040	123.82±.02
GCD	.0278±0.0001	.5565±.0008	8.5292±.0096	216.00±.55
SupRandom	.0007±1.8E-5	.0031±.0001	.0131±.0002	.06±.01

Tableau 3: Estimation du temps moyen par appel (sec.) sur un Sparc-2 de SUN

3 Implantation

3.1 Organisation des modules

Nous allons maintenant décrire brièvement l'organisation des modules de SENTIERS et les principaux problèmes liés à leur implantation. Le diagramme ci-bas illustre la structure hiérarchique qui existe entre les différents modules. Lorsqu'on peut se rendre du module `A` au module `B` en suivant une ou plusieurs flèches, cela indique que le module `B` peut utiliser des objets du module `A`, directement ou indirectement, dans sa définition (".def") ou dans son implantation (".mod"). On remarque en particulier que tous les modules utilisent `SUPINT`. Par ailleurs, le module `SUPFACT` utilise le fichier `SUPFACT.PRI`, qui contient la liste des nombres premiers inférieurs à 2^{16}.



N.B.: Tous les modules utilisent aussi à divers degrés les outils fournis par l'ensemble "MYLIB".

3.2 Gestion de la mémoire

Lorsqu'on manipule des entiers dont la taille peut s'allonger sans limite prédéfinie, il faut prévoir un mécanisme de gestion dynamique de la mémoire. Nous n'avons pas retenu l'idée de stocker l'entier sous forme d'une liste de chiffres, avec des pointeurs, car cela risquait

de trop ralentir l'exécution des programmes. Pour des raisons d'efficacité, nous tenions à représenter les entiers dans des vecteurs. Par contre, nous ne voulions pas réserver un trop grand bloc de mémoire inutilement, afin d'éviter le gaspillage.

La solution adoptée est la suivante. Lors de la création d'un grand entier, on réserve un bloc de 2^k mots mémoire pour le vecteur de chiffres (la valeur de k est fournie par l'usager lors de la création). Chaque fois que la taille de ce bloc devient insuffisante, on alloue un nouveau bloc dont la taille est le double de la valeur précédente, on y copie le vecteur précédent, et on récupère l'ancien bloc dans une pile d'espace libre. Il y a une pile d'espace libre pour chaque taille de bloc (chaque puissance de 2). Lorsqu'on a besoin d'un nouveau bloc de mémoire pour doubler la taille d'un vecteur ou pour initialiser un nouveau grand entier, on va d'abord voir dans la pile appropriée avant de demander de la mémoire additionnelle au système.

Une variable de type `SuperInteger` est en fait un pointeur sur un `RECORD` contenant le signe du grand entier, le nombre de chiffres du grand entier, la longueur (en mots) du vecteur utilisé pour mémoriser ces chiffres, et un pointeur sur ce vecteur. Le type déclaré de ce dernier pointeur est celui d'un pointeur sur un vecteur de la plus grande taille possible que l'on peut adresser sur l'ordinateur utilisé. Les seules limites sur la taille des grands entiers sont donc celles imposées par la taille de la mémoire disponible et adressable. Pour éviter d'allouer tout cet espace en pratique, on doit tricher un peu pour déjouer le compilateur. L'astuce consiste à ne jamais allouer de mémoire en utilisant le type sur lequel pointe le pointeur, et à ne jamais déclarer de variable de ce type, mais à demander séparément un bloc de mémoire, puis à initialiser le pointeur à l'adresse de ce bloc. Ainsi, on fait croire au compilateur, et aux routines qui vérifient les indices hors limites à l'exécution, que l'on a affaire à un vecteur de taille pratiquement illimitée. Ce sont les modules eux mêmes qui s'assurent que l'on ne déborde pas du bloc de mémoire alloué.

3.3 Problèmes de visibilité

Tel que présenté à l'annexe A du présent guide, dans la définition de `SUPINT`, le type `SuperInteger` apparaît comme un type opaque, pour lequel on n'a pas accès à l'implantation. Les bons principes du génie logiciel veulent que l'on empêche l'usager d'aller modifier les champs du "`RECORD`" associé ou les chiffres du vecteur. Cependant, certains des autres modules de SENTIERS (comme par exemple `SUPARITH`) doivent justement pouvoir modifier le vecteur de chiffres et les champs du "`RECORD`". Idéalement, on voudrait que les modules de SENTIERS puissent y avoir accès, mais pas les autres modules. Malheureusement, Modula-2 ne permet pas une telle "hiérarchisation" de la visibilité, à moins de définir des modules à l'intérieur d'autres modules, ce qui amène d'autres problèmes. Ainsi, la définition complète du type `SuperInteger` apparaît dans `SUPINT.def`, et elle est en fait accessible à l'usager, mais pour éviter que l'usager en abuse, nous l'avons omise du guide de l'usager. Il en est de même de quelques (peu nombreuses) procédures, qui apparaissent dans le "`.def`" de certains modules, mais pas dans le guide. A noter que pour chaque module, le texte de l'annexe A du présent guide et le "`.def`" du module en question ont été produits à partir d'un seul et même fichier, l'un en utilisant TEX, et l'autre par un petit programme enlevant le texte superflu. Cette approche a certainement aidé à éviter les inconsistances.

3.4 Pourquoi pas une approche fonctionnelle ?

Tel que mentionné dans l'introduction, l'utilisation d'une approche fonctionnelle avec une notation infixe aurait été en principe plus intéressante pour l'usager. En particulier, cela aurait permis de composer les opérations comme on le fait couramment lorsqu'on utilise les opérateurs arithmétiques fournis par les langages habituels. Malheureusement, une telle approche posait des difficultés d'implantation considérables, et nous l'avons mise de côté. Nous résumons ici quelques-unes de ces difficultés.

D'abord, pour avoir une notation infixe, il faudrait pouvoir redéfinir les opérateurs, ce qui n'est pas possible en Modula-2. Nous nous limiterons donc à discuter l'approche fonctionnelle sans notation infixe. Supposons que `Mult (A, B)` et `Sub (A, B)` retournent des grands entiers qui sont respectivement le résultat du produit de `A` par `B`, et de la soustraction `A - B`, où `A` et `B` sont des grands entiers. En principe, pour calculer `C := AB - A`, et recopier ensuite le résultat dans `D`, on pourrait faire :

```
C := Sub (Mult (A, B), A);
D := C;
```

mais cela pose des problèmes importants. D'abord, comme `Sub` ne retourne qu'un pointeur (pointant sur une structure de donnée qu'il doit avoir lui-même créée), toute la structure sur laquelle pointait `C` est perdue sans que l'espace mémoire qu'elle occupait ne soit récupéré. De plus `D` pointe alors sur la même structure de données que `C`, et toute modification ultérieure à l'une de ces deux variables impliquera automatiquement une modification à l'autre (les deux grands entiers occupent le même espace mémoire). Une solution à ce problème consiste à définir un énoncé d'affectation (par une procédure appelée `Affect`) qui copie vraiment toute la structure de données. Les énoncés ci-haut peuvent alors s'écrire :

```
Affect (Sub (Mult (A, B), A), C);
Affect (C, D);
```

L'énoncé d'affectation ne doit évidemment pas détruire la structure de données qu'il copie : après avoir donné à `D` la valeur de `C`, on veut que `C` continue d'exister par lui-même. Cependant, dans le premier énoncé, le grand entier retourné par `Sub` ne doit pas être conservé après avoir été affecté à `C`. D'ailleurs, même s'il l'est, on ne pourra plus jamais y référer, car aucune variable ne lui correspond. Il y aura donc là de l'espace perdu que l'on ne pourra plus jamais récupérer. Dans le cas où un tel énoncé apparaît dans une boucle où dans une procédure récursive, cela peut être désastreux.

Pour résoudre ce problème, on pourrait penser à distinguer deux types de grands entiers : ceux créés explicitement par l'usager, et ceux créés par le système à l'intérieur des fonctions pour retourner leur résultat. On ajoutera à chaque grand entier un champ indiquant son type. Lorsqu'une fonction ou une procédure accepte en entrée un grand entier du second type (que l'on peut appeler "variable tampon"), elle l'utilise comme à l'habitude, mais lorsqu'elle en a terminé, elle le détruit. Ainsi, dans l'exemple précédent, l'espace occupé par le résultat de `Mult` est récupéré dès que `Sub` a terminé de l'utiliser. De même, le résultat de `Sub` est

détruit après avoir été affecté à `C`. Par contre, `C` n'est pas détruit après avoir été affecté à `D`. Malheureusement, cette approche n'est pas sans problèmes. Considérons l'exemple suivant :

```
PROCEDURE Toto (VAR A, B, C : SuperInteger);
   BEGIN
   Affect (Sub (A, B), C);
   Affect (Mult (A, C), C)
   END Toto;

     ⋮

Toto (Mult (X, Y), W, Z);
```

Ici, le paramètre formel `A` prend la valeur du résultat de `Mult (X, Y)`. Il s'agit donc d'une variable tampon, et lorsque `Sub` a terminé de l'utiliser, il le détruit. Plus loin, lorsque `Mult` veut réutiliser `A`, l'espace utilisé par ce dernier a déjà été récupéré.

Compte tenu de ces difficultés, nous avons adopté une approche procédurale, qui force l'utilisateur à déclarer et gérer lui-même les variables tampons utilisées dans les expressions compliquées.

ANNEXES

A. Définition fonctionnelle des modules de SENTIERS

SUPINT

`SUPINT` est le module de base de SENTIERS. Il fournit le type `SuperInteger`, de même que des procédures pour créer ou détruire des grands entiers, examiner leur longueur, effectuer des affectations, des échanges, des entrées/sorties, ou libérer la mémoire. Les grands entiers sont représentés dans des vecteurs de chiffres en base $b = 2^{(\beta-2)/2}$, où β est habituellement le nombre de bits dans un mot, sur l'ordinateur utilisé.

Chaque variable représentant un grand entier doit être déclarée de type `SuperInteger`, puis initialisée (avant sa première utilisation) en appelant la procédure `Create`. La taille du bloc de mémoire utilisé pour mémoriser le vecteur de chiffres associé est toujours une puissance de 2, et la taille initiale est donnée lors de la création. Lors des opérations sur les grands entiers, le système s'occupe d'agrandir automatiquement au besoin la taille des vecteurs de chiffres utilisés, de façon à ce qu'il n'y ait jamais de débordement (du moins tant qu'il reste de la mémoire disponible). Dans ce cas, l'ancien vecteur est placé dans une pile d'espace libre, et on alloue un nouveau vecteur, dont la taille est le double de celle du vecteur précédent. Il y a une pile d'espace libre pour chaque taille de vecteur (chaque puissance de 2). Lorsqu'on a besoin d'un nouveau bloc de mémoire contiguë, pour doubler le vecteur associé à un `SuperInteger` ou pour initialiser un nouveau `SuperInteger`, le module va d'abord voir dans la pile appropriée, avant de demander de l'espace additionnel au système. La procédure `Delete` enlève à une variable de type `SuperInteger` son caractère initialisé et récupère l'espace mémoire qu'elle occupait. Cet espace est conservé dans la pile d'espace libre appropriée. La procédure `ReleaseStack` permet de retourner au système le contenu de toutes les piles d'espace libre.

Comme une variable de type `SuperInteger` n'est qu'un pointeur sur un bloc d'information, on ne peut pas utiliser l'énoncé d'affectation "`:=`" pour donner au `SuperInteger` B la valeur du `SuperInteger` A. En fait, au terme de l'énoncé "`B := A`", on obtient deux pointeurs sur le même bloc d'information (c'est-à-dire que le même `SuperInteger` a maintenant deux noms différents), et le bloc d'information qui était associé à B est perdu. Les affectations entre deux variables de type `SuperInteger` doivent se faire par le biais de la procédure `Affect`. La procédure `Exchange` permet d'échanger le contenu de deux variables de type `SuperInteger`.

```
DEFINITION MODULE SUPINT;

TYPE
   SuperInteger;
```

Type de variable pouvant représenter un grand entier. Pour chaque variable de ce type, on doit d'abord appeler `Create`, pour l'initialiser, avant de l'utiliser.

```
PROCEDURE Create
   ( VAR A : SuperInteger;
     k : CARDINAL
   );
```

Initialise le bloc d'information correspondant au grand entier A. La valeur initiale du grand entier est de 0. La taille initiale du vecteur de chiffres sera de 2^k, ce qui permet de représenter

des nombres aussi grands que $b^{(2^k)} - 1$ (où b est la base choisie) sans que le module ait besoin d'agrandir ce vecteur. La valeur maximale de k dépend de l'implantation, et correspond habituellement à la plus grande valeur de k pour laquelle il est possible de déclarer un bloc de mémoire de taille 2^k.

```
PROCEDURE Delete
   ( VAR A : SuperInteger
   );
```

Cette procédure est l'inverse de la procédure `Create`. Elle permet de libérer l'espace occupé par le grand entier A.

```
PROCEDURE ReleaseStack;
```

Libère toute la pile d'espace libre et retourne cet espace-mémoire au système. Cette procédure peut servir dans le cas où on veut utiliser à d'autres fins l'espace occupé auparavant par des vecteurs de chiffres.

```
PROCEDURE NumDigits
   ( A : SuperInteger
   ) : CARDINAL;
```

Retourne le nombre de chiffres utilisés dans la représentation de A en base b (en excluant le signe).

```
PROCEDURE NumBits
   ( A : SuperInteger
   ) : CARDINAL;
```

Retourne le nombre de bits dans la représentation binaire de A, soit $\lceil \log_2 A \rceil$.

```
PROCEDURE Affect
   ( A, B : SuperInteger
   );
```

Donne à B la valeur de A (sans modifier la valeur de A).

```
PROCEDURE Exchange
   ( VAR A, B : SuperInteger
   );
```

B prend la valeur de A et A la valeur de B.

```
PROCEDURE ReadSup
   ( A : SuperInteger;
     B : CARDINAL
   );
```

Permet de lire la valeur du grand entier `A`, dans la base `B`, dans le fichier d'entrée courant. La base doit être comprise entre 2 et 16 inclusivement. Cette procédure sautera les blancs et/ou les fins-de-ligne préalables, et acceptera une suite de chiffres dans la base choisie (au moins un chiffre), précédé éventuellement d'un signe "`+`" ou "`-`". Elle s'arrêtera au premier blanc ou fin-de-ligne rencontré (sauf pour le cas mentionné ci-après). Dans le cas où on veut qu'un nombre soit lu sur plusieurs lignes, il suffit de terminer la suite de chiffres de chaque ligne, sauf la dernière, par le signe "`~`". Si un caractère autre qu'un chiffre admissible dans la base choisie,

ou que l'un de ceux ci-haut mentionnés, est rencontré, un message d'erreur est affiché et le programme s'arrête.

```
PROCEDURE WriteSup
   ( A : SuperInteger;
     B : CARDINAL;
     D : CARDINAL
   );
```

Ecrit la valeur du grand entier `A` dans la base `B`, dans le fichier de sortie courant, avec un maximum de `D` chiffres par lignes. La base doit être comprise entre 2 et 16 inclusivement. Lorsque le nombre doit être écrit sur plusieurs lignes, le caractère ˜ est ajouté à la fin de chaque ligne, sauf la dernière.

```
PROCEDURE WriteSupSci
   ( A : SuperInteger;
     B : CARDINAL;
     n : CARDINAL
   );
```

Ecrit la valeur du grand entier `A` dans la base `B`, dans le fichier de sortie courant, sous le format scientifique suivant: . ¡n chiffres les plus significatifs¿ E ¡nombre total de chiffres¿ La base doit être comprise entre 2 et 16 inclusivement.

```
END SUPINT.
```

SUPCONV

Ce module sert à effectuer des conversions entre des variables de type `SuperInteger`, et des variables de type, `LONGCARD`, `LONGINT`, `REAL`, ou point flottant double précision.

```
DEFINITION MODULE SUPCONV;

FROM SUPINT IMPORT SuperInteger;

PROCEDURE IntToSup
   ( I : LONGINT;
     A : SuperInteger
   );
```

Donne à A la valeur de I.

```
PROCEDURE SupToInt
   ( A : SuperInteger;
     VAR I : LONGINT
   );
```

Donne à I la valeur de A. Si la valeur de A n'est pas représentable par une variable de type `LONGINT`, un message d'erreur est affiché et le programme s'arrête.

```
PROCEDURE CardToSup
   ( C : LONGCARD;
     A : SuperInteger
   );
```

Donne à A la valeur de C.

```
PROCEDURE SupToCard
   ( A : SuperInteger;
     VAR C : LONGCARD
   );
```

Donne à C la valeur de A. Si la valeur de A n'est pas représentable par une variable de type `LONGCARD`, un message d'erreur est affiché et le programme s'arrête.

```
PROCEDURE SupToReal
   ( A : SuperInteger;
     VAR R : REAL
   );
```

Donne à R la valeur de A. Si la valeur de A n'est pas représentable par une variable de type `REAL`, un message d'erreur est affiché et le programme s'arrête.

```
PROCEDURE RoundLongRealToSup
   ( R : LONGREAL;
     A : SuperInteger
   );
```

Donne à A la valeur (arrondie) de R.

```
PROCEDURE LongRealToSup
   ( R : LONGREAL;
     A : SuperInteger
   );
```

Donne à A la valeur de R. R doit être entier, et peut être supérieur au plus grand LONGINT possible. Il n'y a cependant pas de message d'erreur si R n'était pas représentable exactement en LONGREAL.

```
PROCEDURE SupToLongReal
   ( A : SuperInteger;
     VAR R : LONGREAL
   );
```

Donne à R la valeur de A. Si la valeur de A n'est pas représentable par une variable de type `LONGREAL`, un message d'erreur est affiché et le programme s'arrête.

```
END SUPCONV.
```

SUPARITH

`SUPARITH` est un module servant à effectuer des opérations arithmétiques de base et des comparaisons sur les grands entiers.

Des fonctions booléennes permettent de comparer deux `SuperInteger`, ou de savoir si un `SuperInteger` est négatif, positif ou nul. On peut changer le signe ou prendre la valeur absolue d'un `SuperInteger`, additionner, soustraire, multiplier ou diviser deux `SuperInteger`. Dans le cas de la division (entière), la procédure `Quotient` ne retourne que le quotient, `Modulo` ne retourne que le reste de la division (le modulo), et `Divide` retourne les deux. Les procédures effectuant des opérations arithmétiques ne modifient jamais les opérandes données en entrée, sauf dans le cas où le même `SuperInteger` est donné à la fois en entrée et en sortie. Par exemple, si `S1` et `S2` sont des `SuperInteger`, `Add (S1, S2, S1)` ajoute à `S1` la valeur de `S2`, tandis que `ChangeSign (S1, S1)` change le signe de `S1`. Lorsqu'on veut multiplier ou diviser par une puissance de la base, ou faire un calcul modulo une puissance de la base, on peut le faire de façon très efficace en utilisant `ShiftLeft`, `ShiftRight` ou `CutLeft`. Les procédures `MultiplyIntSup` et `DivideSupInt` permettent de multiplier ou de diviser un `SuperInteger` par un `LONGINT`, `PowerSupInt` permet d'élever un `SuperInteger` à une puissance entière positive, et `GCD` calcule le plus grand commun diviseur. Les procédures `MultMod` et `PowerMod` permettent de multiplier ou d'élever à une puissance dans l'arithmétique modulo un entier M.

```
DEFINITION MODULE SUPARITH;

FROM SUPINT IMPORT SuperInteger;

PROCEDURE ShiftLeft
   ( A : SuperInteger;
     k : CARDINAL;
     B : SuperInteger
   );
```

Procédure très efficace qui multiplie A par b^k et place le résultat dans B. Effectue un décalage à gauche de k positions.

```
PROCEDURE ShiftRight
   ( A : SuperInteger;
     k : CARDINAL;
     B : SuperInteger
   );
```

Procédure très efficace qui divise A par b^k et place le résultat dans B. Effectue un décalage à droite de k positions.

```
PROCEDURE CutLeft
   ( A : SuperInteger;
     k : CARDINAL;
     B : SuperInteger
   );
```

Procédure très efficace qui calcule le reste de la division de A par b^k et place le résultat dans B. Conserve les k chiffres de droite et élimine tous les autres.

```
PROCEDURE Even
   ( A : SuperInteger
   ) : BOOLEAN;
```

Retourne `TRUE` si A est pair, et `FALSE` sinon.

```
PROCEDURE Positive
   ( A : SuperInteger
   ) : BOOLEAN;
```

Retourne `TRUE` si A est un entier positif (> 0), et `FALSE` sinon.

```
PROCEDURE Negative
   ( A : SuperInteger
   ) : BOOLEAN;
```

Retourne `TRUE` si A est un entier négatif (< 0), et `FALSE` sinon.

```
PROCEDURE Zero
   ( A : SuperInteger
   ) : BOOLEAN;
```

Retourne `TRUE` si A est égal à zéro, et `FALSE` sinon.

```
PROCEDURE Smaller
   ( A, B : SuperInteger
   ) : BOOLEAN;
```

Retourne `TRUE` si la valeur de A est inférieure à celle de B ($A < B$), et `FALSE` sinon.

```
PROCEDURE Greater
   ( A, B : SuperInteger
   ) : BOOLEAN;
```

Retourne `TRUE` si la valeur de A est supérieure à celle de B ($A > B$), et `FALSE` sinon.

```
PROCEDURE Equal
   ( A, B : SuperInteger
   ) : BOOLEAN;
```

Retourne `TRUE` si la valeur de A est égale à celle de B, et `FALSE` sinon.

```
PROCEDURE AbsSizeOrd
   ( A, B : SuperInteger
   ) : INTEGER;
```

Compare la taille de deux `SuperInteger` sans egard au signe. Retourne 1 si la taille de A est inférieure à celle de B ($ABS(A) < ABS(B)$). Retourne -1 si la taille de A est supérieure à celle de B ($ABS(A) > ABS(B)$). Retourne 0 si A et B sont de taille égales ($ABS(A) = ABS(B)$).

```
PROCEDURE ChangeSign
   ( A, B : SuperInteger
   );
```

Place la valeur de $-A$ dans B.

```
PROCEDURE AbsVal
   ( A, B : SuperInteger
   );
```

Place la valeur absolue de A dans B.

```
PROCEDURE Add
   ( A, B, C : SuperInteger
   );
```

Additionne A et B et place le résultat dans C.

```
PROCEDURE Subtract
   ( A, B, C : SuperInteger
   );
```

Place la valeur de A moins la valeur de B dans C.

```
PROCEDURE Multiply
   ( A, B, C : SuperInteger
   );
```

Multiplie A et B et place le résultat dans C.

```
PROCEDURE MultiplyIntSup
   ( I : LONGINT;
     B, C : SuperInteger
   );
```

Multiplie I par B et place le résultat dans C.

```
PROCEDURE Divide
   ( A, B, Q, R : SuperInteger
   );
```

Divise A par B, place la valeur du quotient dans Q, et le reste dans R. On a $QB + R = A$ et $|R| < |B|$. Si $B = 0$, imprime un message d'erreur et arrête le programme.

```
PROCEDURE DivideSupInt
   ( A : SuperInteger;
     B : LONGINT;
     Q : SuperInteger;
     VAR R : LONGINT
   );
```

Divise A par B, place la valeur du quotient dans Q, et le reste dans R. On a $QB + R = A$ et $|R| < |B|$. Si $B = 0$, imprime un message d'erreur et arrête le programme.

```
PROCEDURE DivideLR (S, T : SuperInteger; VAR equal : BOOLEAN) : LONGREAL;
```

Retourne le plus grand réel tel que ce réel est inferieur où égal à S / T. S'il est égal, `equal` est mis à `TRUE`, sinon à `FALSE`.

```
PROCEDURE DivideRacLR (S, T : SuperInteger) : LONGREAL;
```

Retourne le plus grand réel tel que le carré de ce réel est inferieur où égal à S / T.

```
PROCEDURE Quotient
   ( A, B, Q : SuperInteger
   );
```

Calcule le quotient tronqué $\lfloor A/B \rfloor$ et place sa valeur dans Q.

```
PROCEDURE Modulo
   ( A, B, R : SuperInteger
   );
```

Calcule $A \bmod B$ et place le résultat dans R. On a $R = A - B\lfloor A/B \rfloor$. Si $B \leq 0$, imprime un message d'erreur et arrête le programme. A noter que le modulo pourra être différent du reste de la division de A par B et qu'il sera toujours positif.

```
PROCEDURE PowerSupInt
   ( A : SuperInteger;
     I : LONGCARD;
     C : SuperInteger
   );
```

Calcule A^I et place le résultat dans C.

```
PROCEDURE GCD
   ( A, B, C : SuperInteger
   );
```

Calcule le plus grand commun diviseur de A et B et place le résultat dans C.

```
PROCEDURE MultMod
   ( A, B, M, C : SuperInteger
   );
```

Calcule le produit $AB \bmod M$ et place le résultat dans C.

```
PROCEDURE PowerMod
   ( A, B, M, C : SuperInteger
   );
```

Calcule $A^B \bmod M$ et place le résultat dans C.

```
END SUPARITH.
```

SUPRAND

Permet de générer des grands entiers au hasard. Chaque chiffre du grand entier est généré séparément, en utilisant le générateur 32-bits proposé dans [10]. En fait, on génère une valeur entre zéro et `D = B - A`, à laquelle on ajoute ensuite `A`. On génère d'abord le chiffre le plus significatif. Dans le cas (habituellement rare) où le chiffre ainsi généré est égal au chiffre le plus significatif de `D`, on doit vérifier si le nombre généré dépasse `D`, auquel cas on recommence le tout.

```
DEFINITION MODULE SUPRAND;

FROM SUPINT IMPORT SuperInteger;

PROCEDURE SupRandom
   ( A, B, C : SuperInteger
   );
```

Génère un grand entier au hasard, selon la loi uniforme discrète entre `A` et `B` (inclusivement), et place sa valeur dans `C`.

```
PROCEDURE SetSeed
   ( S1, S2 : LONGINT
   );
```

Initialise le germe initial du générateur au couple (`S1`, `S2`). Ces valeurs doivent satisfaire : $1 \leq$ `S1` ≤ 2147483562 et $1 \leq$ `S2` ≤ 2147483398.

```
PROCEDURE GetState
   ( VAR S1, S2 : LONGINT
   );
```

Permet d'examiner l'état du générateur. Cet état est retourné dans `S1` et `S2`. Peut servir si on veut mémoriser l'état du générateur afin, par exemple, de l'y réinitialiser ultérieurement.

```
END SUPRAND.
```

SUPFACT

Ce module offre des outils pour décomposer un grand entier en facteurs premiers, ou tester sa primalité. Il utilise le fichier `SUPFACT.PRI`, qui contient la liste de tous les nombres premiers inférieurs à 2^{16} écrits en base 2 sur un mot de l'ordinateur. Pour tester la primalité des grands entiers supérieurs à 2^{32} et n'ayant pas de facteur inférieur à 2^{16}, on utilise le test de Rabin [1, 2]. Il s'agit d'un test probabiliste de type Monte-Carlo, que l'on répète k fois. Si le nombre est premier, l'algorithme ne se trompe jamais. Par contre, si le nombre est composé, il est possible que l'algorithme dise qu'il est premier (se trompe), mais seulement avec une probabilité inférieure à 4^{-k}. L'algorithme prend un temps linéaire en fonction de k, et il suffit de choisir une valeur de k assez grande pour que la probabilité d'erreur soit négligeable (comme par exemple $k = 100$). A noter cependant qu'étant donné un nombre quelconque, la probabilité que ce nombre soit effectivement premier, étant donné que l'algorithme dit qu'il l'est, dépend de la probabilité *a priori* que le nombre soit premier, et n'est pas vraiment facile à calculer (voir [1] pour plus de détails à ce sujet). Les procédures `IsPrime` et `IsProbablyPrime` utilisent cet algorithme (`IsPrime` vérifie d'abord si le grand entier est divisible par un nombre premier inférieur à 2^{16}). Si l'une ou l'autre retourne `FALSE` pour un grand entier A, alors A est composé. Si `IsProbablyPrime` retourne `TRUE`, A est probablement premier. Si `IsPrime` retourne `TRUE`, alors A est certainement premier s'il est inférieur à 2^{32}, et il est probablement premier s'il est supérieur à 2^{32}.

Pour la décomposition en facteurs, on utilise l'algorithme de Pollard tel qu'amélioré et présenté par Montgomery [12], après avoir enlevé tous les facteurs premiers inférieurs à 2^{16}. La primalité des gros facteurs qui restent (supérieurs à 2^{32}) est testée en utilisant le test de Rabin avec $k = 100$.

```
DEFINITION MODULE SUPFACT;

FROM SUPINT IMPORT SuperInteger;

TYPE
   Factors = POINTER TO InfoFactors;    (* Une liste de facteurs.       *)
   InfoFactors = RECORD
      Factor       : SuperInteger;      (* Un facteur.                  *)
      Multiplicity : LONGCARD;          (* Sa multiplicité.             *)
      Prime        : BOOLEAN;           (* FALSE ssi on sait qu'il est  *)
                                        (* composé.                     *)
      Next         : Factors            (* Liste des facteurs suivants, *)
                                        (* = NIL s'il n'y en a plus.    *)
      END;

PROCEDURE IsPrime
   ( A : SuperInteger;
     k : CARDINAL
   ) : BOOLEAN;
```

Si `A` est un nombre premier, retourne toujours `TRUE`, sinon retourne presque toujours `FALSE` (voir ci-haut). Pour les entiers inférieurs à 2^{32}, l'algorithme ne se trompe jamais. Pour les autres, il utilise le test de Rabin avec k tours de boucle s'ils n'ont aucun facteur premier inférieur à 2^{16}.

```
PROCEDURE IsProbablyPrime
   ( A : SuperInteger;
     k : CARDINAL
   ) : BOOLEAN;
```

Applique le test de Rabin avec k tours de boucle, mais ne vérifie pas si A est divisible par un nombre premier inférieur à 2^{16}.

```
PROCEDURE Factorize
   ( A     : SuperInteger;
     T     : LONGREAL;
     VAR L : Factors
   );
```

Tente de décomposer `A` en facteurs premiers. Cette procédure s'arrête dès qu'elle a consommé plus de `T` minutes de CPU, avec une marge d'erreur d'environ une minute. La liste des facteurs est retournée dans `L`, et pour chacun des facteurs, on indique sa multiplicité et s'il est premier ou pas. Par exemple, si `A` = 32, la liste ne contiendra qu'un seul facteur premier, soit 2, de multiplicité 5. Si `A` est (très probablement) premier, la liste contiendra la valeur de `A`, avec la multiplicité 1. Dans le cas où la procédure est interrompue pour avoir dépassé le temps de CPU alloué, le cofacteur restant est placé dans la liste et pourrait éventuellement être décomposé par un appel subséquent à `Factorize`. La procédure ne s'interrompt jamais pendant qu'elle élimine les facteurs premiers inférieurs à 2^{16}, ni pendant qu'elle teste la primalité d'un nombre, ce qui explique en majeure partie la marge d'erreur sur le temps de CPU effectivement consommé. Notez qu'il est possible que l'algorithme ne réussisse pas à décomposer certains nombres, surtout si leurs facteurs sont de grande taille.

```
END SUPFACT.
```

B. Guide d'utilisation sur VAX/VMS.

SENTIERS sous VAX/VMS a été implanté en utilisant le compilateur Modula-2 de Logitech [6] (version 3.1–9). La base utilisée est de 2^{15}, et un grand entier peut avoir au maximum 2^{29} chiffres.

Tous les modules décrits dans ce guide sont précompilés, selon les spécifications de [6]. Chaque fichier du progiciel est placé dans un répertoire accessible aux usagers éventuels. Pour simplifier l'utilisation, les usagers peuvent redéfinir les noms complets des modules (incluant le répertoire) par des noms logiques plus courts. Ils pourront utiliser ces noms logiques dans leurs énoncés `IMPORT` et lors de l'édition de liens. Par exemple, si `SUPINT` est placé dans le répertoire "`[DIR]`", on peut définir son nom logique en exécutant la commande:

```
$ DEFINE SUPINT.*   [DIR]SUPINT.*
```

Si l'usager veut utiliser des modules qui se trouvent disons dans les répertoires "`[DIR]`" et "`[DIR1]`", disons, il peut désigner ces répertoires comme contenant ses librairies principales en effectuant les commandes

```
$ ASSIGN [DIR]  MOD$LIBRARY
$ ASSIGN [DIR1] MOD$LIBRARY_1
```

A l'Université Laval, les modules de SENTIERS sont installés sur l'ordinateur `saphir`, dans le répertoire "`INFORMATIQUE:[1170130.SENTIERS]`".

Le fichier `SENTIERS.COM` contient un exemple d'un ensemble de commandes permettant de définir un environnement facilitant le travail avec SENTIERS. Le fichier `SLINK.COM` définit une commande pour faire l'édition de liens d'un programme avec *tous* les modules de SENTIERS. En pratique, pour réduire la taille du code exécutable, il est cependant préférable de ne faire l'édition de liens qu'avec les modules effectivement utilisés (directement ou indirectement). A titre d'exemple, l'usager peut consulter le fichier `testv.mod`, qui contient le programme ayant servi à faire les expériences de la section 2.3. Le fichier `testvlink.com` a servi à en faire l'édition de liens, et `testvsoumet.com` a servi à lancer quelques exécutions.

C. Guide d'utilisation sous MS-DOS avec TopSpeed Modula-2.

SENTIERS sous MS-DOS est implanté en utilisant le compilateur *TopSpeed Modula-2* [15]. Les programmes SIMOD doivent donc être traités par ce compilateur.

D. Guide d'utilisation sur SUN avec SUN/Modula-2.

SENTIERS sous SUN/OS est implanté en utilisant le compilateur Modula-2 de SUN. Les programmes SENTIERS doivent donc être traités par ce compilateur.

Références

[1] Beauchemin, P., Brassard, G., Crépeau, C., Goutier, C. et Pomerance, C. (1988). The Generation of Random Numbers that are Probably Prime.

[2] Brassard, G. et Bratley, P. (1987). *Algorithmique, conception et analyse*, Masson et cie., Paris, et Les Presses de l'Université de Montréal.

[3] Brent, R. P. (1978). A FORTRAN Multiple Precision Arithmetic Package. *ACM Trans. on Math. Software*, **4**, 1, 57–81.

[4] Cherry, L. et Morris, R. (1978). BC – An Arbitrary Precision Desk Calculator Language. Dans le *UNIX Programmer's Manual*, Bell Laboratories.

[5] Collins, G. E. (1966). PM, A System for Polynomial Manipulation. *Communications of the ACM*, **9**, 8, 578–589.

[6] Eckhardt, H., Koch, J., Mall, M. et Putfarken, P. (1985). *Logitech Modula-2 User's Manual*, Logitech Inc., Redwood City, California.

[7] Ford, G. A. et Wiener, R. S. (1985). *Modula-2 : A Software Development Approach*. Wiley, New-York.

[8] Gleaves, R. (1984). *Modula-2 for Pascal Programmers*. Springer-Verlag, New-York.

[9] Knuth, D. E. (1981). *The Art of Computer Programming, vol. 2 : Seminumerical Algorithms*, second edition, Addison-Wesley.

[10] L'Ecuyer, P. (1987). Efficient and Portable Combined Random Number Generators. A paraître dans *Communications of the ACM*.

[11] L'Ecuyer, P. (1988). SIMOD: Définition fonctionnelle et guide d'utilisation, version 2.0. Rapport technique DIUL-RR-8804, Département d'informatique, Université Laval.

[12] Montgomery, P. L. (1987). Speeding the Pollard and Elliptic Curve Methods of Factorization. *Math. of Computation*, **48**, 177, 243–264.

[13] Morris, R. et Cherry, L. (1978). DC – An Interactive Desk Calculator. Dans le *UNIX Programmer's Manual*, Bell Laboratories.

[14] Macintosh Programmer's Workshop; v. 2.0 (1987). Apple Computers inc.

[15] *TopSpeed Modula-2 User's Manual*, Jensen and Partners International (1988).

[16] Wirth, N. (1985). *Programming in Modula-2*. Third ed., Springer-Verlag, New-York.