Algorithmique et traduction pour calculatrices et autres langages

MapleV, Mathematica, Maxima, MuPAD, Pascal, C++, xcas

surnommé

"Le gros rouge d'Algorithmique"

Renée De Graeve

2 juillet 2001

Remerciements

Je remercie :
- Claire Helmstetter pour ses précieux conseils et ses remarques sur ce texte,
- Paul Perret pour avoir coanimé efficacement le premier stage IREM d'arithmétique, stage qui a donné naissance à ce document, et aussi pour avoir écrit la partie consacrée à `MapleV`,
- Lionel Darié pour avoir écrit la partie consacrée à `Mathematica`,
- Bernard Parisse pour avoir complété les parties consacrées à `Maxima`, `MuPAD`, `C++` et pour avoir écrit l'annexe indiquant comment on récupère des logiciels sur Internet,
- Jean Pierre Douris et Christiane Serret pour leur relecture,
- Marc Legrand pour l'intérêt porté à l'achèvement de ce document,
- Serge Cecconi pour son illustration,
- Ghislaine Richer pour la conception de la couverture,
- et tous mes collègues de l'IREM de Grenoble .

Je remercie aussi les représentants de Hewlett Packard, Sharp et Texas Instruments qui m'ont donné, lors du congrès national de l'APMEP (Association des Professeurs de Mathématiques de l'Enseignement Public) à Gérardmer, les calculatrices qui me manquaient pour compléter ce manuel (une HP49G, une TI83+ et une Sharp EL9600).

Introduction

Avec la démocratisation du calcul formel on pourrait croire qu'il devient inutile d'apprendre les algorithmes fondamentaux de l'arithmétique. Nous pensons pourtant que la compréhension de ces algorithmes aide à comprendre le concept mathématique sous-jacent et que le but de la formation n'est pas de transformer nos élèves en presse-boutons. Nous proposons donc d'implémenter ces algorithmes dans des langages largement accessibles aux élèves, ils pourront ensuite utiliser des logiciels de calcul formel lorsque les données dépassent les capacités des calculatrices ne faisant pas de calcul formel. Nous avons voulu dans ce qui suit écrire les algorithmes d'arithmétique du programme de spécialité maths : tout d'abord en langage algorithmique, puis dans les différents langages des différentes marques de calculatrices.
On remarquera qu'une fois que l'on s'est mis d'accord sur le langage algorithmique, la traduction en langage calculatrice (ou autre) se fait de façon (presque) systématique.
Nous espérons que ce fascicule facilitera la gestion des différents modèles de calculatrices dans les classes, et sera un premier pas vers l'introduction d'un langage algorithmique commun à tous.
Vous trouverez chapitre 4, les programmes regroupés selon les modèles de calculatrices, pour faire facilement les photocopies destinées aux élèves.
La nouvelle édition de cette brochure a été complétée par une introduction aux langages de calcul formel comme `Maple, Mathematica, Maxima` et `MuPAD`. En effet, un logiciel de calcul formel permet d'effectuer des manipulations algébriques sans avoir à effectuer d'approximation numérique, par exemple calculer la dérivée ou la primitive d'une fonction, résoudre certaines équations différentielles, etc.
`MuPAD` et `Maple` sont les deux logiciels de calculs formel autorisés à la 3eme épreuve orale de l'agrégation de mathématiques. `Mathematica` est un autre logiciel de calcul formel souvent utilisé mais malheureusement assez cher.
`Maxima` est aussi un logiciel de calcul formel : il est devenu récemment un logiciel libre donc gratuit. Il est maintenu par William F. Schelter (dont l'adresse est `wfs@math.utexas.edu`).
`Maple` est plus ancien que `MuPAD` et contient plus de commandes, mais vous devrez payer une licence d'utilisation si vous voulez l'utiliser chez vous (cf `www.maplesoft.com`). La version étudiante de `Maple` coûte environ 100 euros, les versions complètes de `Maple` et de `Mathematica` coûtent plus de 1000 euros.
Par contre, `Maxima` est libre et gratuit se reporter à l'adresse :
`http://www.ma.utexas.edu/users/wfs/maxima.html`

Quant à MuPAD, il est disponible gratuitement pour un usage éducatif se reporter à www.mupad.de et à www.sciface.com.
Les commandes qui existent dans les différents systèmes sont suffisament proches pour que le passage de l'un à l'autre se fasse en douceur.
Enfin, le dernier chapitre est consacré à une introduction à Pascal et à C++ qui sont des langages proches de ceux utilisés dans les calculatrices. À la fin de ce chapitre vous trouverez comment vous pouvez programmer en C++ en utilisant des types symboliques et ainsi faire du calcul formel... encore merci à Bernard Parisse!!!

Resumé
Ce document introduit les bases du langage algorithmique et permet de traduire un même programme en différents langages de programmation :
Pascal, C++, Maple, Mathematica, Maxima, MuPAD
et
en langages utilisés par différents types de calculatrices : CASIO, HP, SHARP, TI
Les exemples de programmation ont pour thème l'arithmétique.

Mots Clés

Algorithme
Algorithmique
Arithmétique
Bézout
Calculatrice
Calcul Formel
C++
Nombre premier
Maple
Mathematica
Maxima
MuPAD
Pascal
PGCD
Premier
xcas

Chapitre 1

Pour commencer

1.1 Comment éditer et sauver un programme

1.1.1 Traduction Casio

Appuyer sur la touche `MENU`, puis ouvrir le menu `PRGM` (en mettant en surbrillance `PRGM` avec les flèches, puis en appuyant sur la touche `EXE`).
Cela vous donne la liste des programmes (écran `Program List`) et un bandeau contenant : `EXE EDIT NEW DEL DEL.A SRC REN`
Avec `F3` vous sélectionnez `NEW` pour éditer un nouveau programme.
On vous demande `Program Name` : il suffit de taper le nom de votre programme (vous êtes en mode Alpha !).
Les touches `SHIFT VARS (PRGM)` font apparaître un bandeau contenant les différentes instructions et la touche `OPTN` fait apparaître un bandeau contenant les différentes fonctions mathématiques.
La touche `EXIT` permet de sortir d'une application.
Pour sauver votre programme, il suffit de taper `2nd EXIT (QUIT)` et on revient à l'écran `Program List`.

1.1.2 Traduction HP38G HP40G

Pour avoir accès au catalogue de programmes, on appuie sur les touches `shift 0 (PROGRAM)`.
Il apparait alors un écran contenant la liste des programmes disponibles et un bandeau (`EDIT NEW SEND RECV RUN`).
Pour taper un nouveau programme, on appuie sur `F2 (NEW)`.
On vous demande le nom du programme : attention ! vous n'êtes pas en mode `Alpha` appuyer sur `F4 (A..Z)` pour y être ! ! !.
Tapez son nom puis `F6 (OK)`.
Vous entrez votre programme et votre travail est automatiquement sauvegardé.

1.1.3 Traduction HP48 HP49G mode RPN

La `HP49G` peut travailler selon deux modes : le mode RPN qui est la notation polonaise inverse et où les différents résultats sont mis sur une pile, ou le mode algébrique qui est la notation habituelle (voir 1.1.4).

Un programme s'écrit dans la ligne de commande entre les délimiteurs ≪ ≫
Pour le sauver, il suffit de faire suivre le dernier ≫ par :

'NOMDUPROGRAMME' STO▷

1.1.4 Traduction HP49G mode Algébrique

Un programme s'écrit dans la ligne de commande entre les délimiteurs ≪ ≫
Pour le sauver, il suffit de faire suivre le dernier ≫ par :

STO ▷ NOMDUPROGRAMME

1.1.5 Traduction TI 80

On appuie sur la touche PRGM et on met en surbrillance NEW avec la flèche →, puis ENTER.
On obtient :
1 :CREATE NEW
Il suffit de taper ENTER.
On obtient :NAME=
Il suffit de taper le nom du programme (vous êtes en mode ALPHA !), puis ENTER.

Vous êtes dans l'éditeur et vous pouvez taper votre programme :
- les : sont mis automatiquement en début de ligne.
- les différentes instructions sont disponibles en appuyant sur PRGM : en effet, depuis l'éditeur le menu de PRGM n'est pas le même (CTL pour les différentes instructions, I/O pour les instructions d'entrées et de sorties, et enfin EXEC pour afficher la liste des noms des programmes ce qui permet d'écrire prgmNOMDUPROG dans une ligne de programme pour appeler un sous programme).
- les fonctions mathématiques sont accessibles en appuyant sur MATH et les fonctions booléennes en appuyant sur 2nd MATH (TEST).
Pour sauver votre programme, il suffit de taper 2nd MODE (QUIT) et vous revenez à l'écran HOME.

1.1.6 Traduction TI 83+

On appuie sur la touche PRGM et on met en surbrillance NEW avec la flèche →, puis ENTER.
On obtient :
PROGRAM Name=
Il suffit de taper le nom du programme (vous êtes en mode ALPHA !), puis ENTER.
Vous êtes dans l'éditeur et vous pouvez taper votre programme :
- les : sont mis automatiquement en début de ligne.
- les différentes instructions sont disponibles en appuyant sur PRGM : en effet, depuis l'éditeur le menu de PRGM n'est pas le même (CTL pour les différentes instructions, I/O pour les instructions d'entrées et de sorties, et enfin

EXEC pour afficher la liste des noms des programmes ce qui permet d'écrire prgmNOMDUPROG dans une ligne de programme pour appeler un sous programme).
- les fonctions mathématiques sont accessibles en appuyant sur MATH et les fonctions booléennes en appuyant sur 2nd MATH (TEST).
Pour sauver votre programme, il suffit de taper 2nd MODE (QUIT) et vous revenez à l'écran HOME.

1.1.7 Traduction TI89, 92

Pour avoir accès à l'éditeur, on appuie sur la touche APPS puis 7 (Program Editor) puis 3 (New) et ENTER.
On vous demande, dans Type, si on veut écrire une fonction (qui renvoie une valeur grâce à Return) ou un programme (appuyer -> puis faites votre choix 1 ou 2) puis ENTER pour valider votre choix.
On écrit ensuite le nom du programme dans la case Variable puis ENTER pour valider le nom, puis ENTER pour entrer dans l'éditeur.
Vous êtes prêt à taper votre programme.
Vous pouvez ajouter éventuellement les noms des paramètres dans la première ligne (en déplaçant le curseur avec les flèches).
Les touches F1 F2 F3 F4 vous facilitent l'édition.
En appuyant sur ♦ Q (HOME) vous sauvez votre programme et vous revenez à l'écran HOME.

1.1.8 Traduction SHARP EL9600

On appuie sur la touche 2nd MATRIX (PRGM) et on met en surbrillance NEW avec la flèche ↓ et ENTER (ou bien avec le stylet il suffit de toucher NEW deux fois).
On obtient :
TITLE ?
Il suffit de taper le nom du programme (vous êtes en mode ALPHA !), puis ENTER.
Vous êtes dans l'éditeur et vous pouvez taper votre programme :
- les différentes instructions sont disponibles en appuyant sur 2nd MATRIX (PRGM) : en effet, depuis l'éditeur le menu de 2nd MATRIX (PRGM) n'est pas le même (PRGM pour les instructions d'entrées et de sorties, BRNCH pour les différentes instructions de programmation etc...).
- les fonctions mathématiques sont accessibles en appuyant sur MATH (CALC et NUM donnent les fonctions de calculs et d'arrondis, INEQ donne les symboles d'inégalités et LOGIC donne les fonctions booléennes).
Pour sauver votre programme, il suffit de taper 2nd CL (QUIT) et on revient à l'écran HOME.

1.2 Comment corriger un programme

1.2.1 Traduction Casio

Si la syntaxe est mauvaise, la machine vous indique : `Go ERROR` si vous voulez aller là où se trouve l'erreur appuyer sur l'une des flèches ($\leftarrow, \rightarrow$).
Vous êtes alors dans l'éditeur prêt à corriger votre erreur.
Attention ! ! !
- vous n'êtes pas en mode insertion automatiquement (appuyer sur `2nd DEL (INS)` pour être en mode insertion jusqu'au prochain déplacement du curseur).

1.2.2 Traduction HP38G HP40G

Si la syntaxe est mauvaise, la machine vous dit :
`Invalid Syntax Edit program ?` Vous répondez `F6 (YES)`.
La machine vous met automatiquement le curseur là où le compilateur a détecté l'erreur. Il suffit donc de corriger ! ! !

1.2.3 Traduction HP48 HP49G mode RPN

Si la syntaxe est mauvaise, la machine vous met automatiquement le curseur là où le compilateur a détecté l'erreur. Il suffit donc de corriger ! ! !
Si l'erreur est détectée au cours de l'exécution du programme il faut taper : `'NOMDUPROGRAMME' VISIT` qui édite votre programme.
On corrige, puis `ENTER` sauve votre programme corrigé. Si vous ne voyez pas quelle peut être la source d'une erreur, vous pouvez utiliser le débuggueur :
- sur la `HP49G` taper `shift-bleu CAT (PRG)` et séléctionner le menu `14 RUN & DEBUG` puis `ENTER`.
- sur la `HP48G` taper `PRG` puis `NXT` puis ouvrir le répértoire `RUN` du bandeau.
Le bandeau `DEBUG SST SST` ↓ `NEXT HALT2 KILL` s'affiche.
`DEBUG` lance le programme et exécute la première instruction puis, `NEXT` affiche l'étape suivante sans l'exécuter et `SST` l'exécute (`NEXT SST` exécute le programme au pas à pas).
Si l'on veut faire l'exécution du programme en pas à pas à partir du milieu du programme il faut mettre un point d'arrêt à cet endroit en insérant la commande `HALT` dans le programme. On lance alors le programme qui s'arrête au niveau du `HALT`, on fait l'exécution pas à pas comme précédemment avec `NEXT SST` etc..
On utilise `KILL` pour arrêter le debugger quand on a compris où etait l'erreur et `Shift-rouge ON (CONT)` pour continuer l'exécution du programme.

1.2.4 Traduction HP49G mode Algébrique

Si la syntaxe est mauvaise, la machine vous met automatiquement le curseur là où le compilateur a détecté l'erreur. Il suffit donc de corriger ! ! !
Si l'erreur est détectée au cours de l'exécution du programme il faut taper : `VISIT('NOMDUPROGRAMME')` qui édite votre programme.
On corrige, puis `ENTER` sauve votre programme corrigé.

1.2.5 Traduction TI 80

Si la syntaxe est mauvaise, la machine vous demande si vous voulez aller là où se trouve l'erreur (`1 :GOTO`) ou si vous voulez arrêter (`2 :QUIT`).
Vous répondez : 1
Vous êtes alors dans l'éditeur prêt à corriger votre erreur.
Puis `2nd MODE (QUIT)` vous fait revenir à l'écran `HOME` en sauvant votre correction.

1.2.6 Traduction TI 83+

Si la syntaxe est mauvaise, la machine vous demande si vous voulez arrêter (`1 :QUIT`), ou si vous voulez aller là où se trouve l'erreur (`2 :GOTO`).
Vous répondez : 2
Vous êtes alors dans l'éditeur prêt à corriger votre erreur.
Puis `2nd MODE (QUIT)` vous fait revenir à l'écran `HOME` en sauvant votre correction.

1.2.7 Traduction TI89 92

Si la syntaxe est mauvaise, la machine vous demande si vous voulez aller là où se trouve l'erreur (`ENTER=GOTO`) ou si vous voulez arrêter (`ESC=CANCEL`).
Vous répondez : `ENTER`.
Vous êtes alors dans l'éditeur prêt à corriger votre erreur.
Puis ♦ `Q` (`HOME`) vous fait revenir à l'écran `HOME` en sauvant votre correction.
Si l'erreur est détectée au cours de l'exécution du programme il faut taper : `APPS` puis `7 (Program Editor)` puis `1 (Current)` et `ENTER` qui édite votre programme.

1.2.8 Traduction SHARP EL9600

Si la syntaxe est mauvaise, la machine vous demande si vous voulez aller là où se trouve l'erreur à l'aide des flèches (←, → `:Goto error`) ou si vous voulez arrêter (`CL :Quit`),.
Vous répondez en appuyant sur une flèche : vous êtes alors dans l'éditeur prêt à corriger votre erreur.
Attention ! ! !
- vous n'êtes pas en mode insertion automatiquement (appuyer sur `2nd DEL (INS)` pour être en mode insertion jusqu'au prochain `2nd DEL (INS)`).
- il faut valider la ligne corrigée en changeant de ligne en appuyant par exemple sur ↓.
Puis `2nd CL (QUIT)` vous fait revenir à l'écran `HOME`.

1.3 Comment exécuter un programme

1.3.1 Traduction Casio

Dans l'écran `Program List` (au départ de l'écran `HOME` touche `MENU PRGM EXE` ou après avoir tapé le programme `2nd EXIT (QUIT)`), il suffit de mettre

en surbrillance le nom du programme à exécuter et de taper sur `F1` pour `EXE` du bandeau (dernière ligne de l'écran où se trouve un menu).

1.3.2 Traduction HP38G HP40G

Pour exécuter un programme, on ouvre le catalogue de programmes en appuyant sur les touches `shift 0 (PROGRAM)`.
Il apparait alors un écran contenant la liste des programmes disponibles et un bandeau (`EDIT NEW SEND RECV RUN`).
On met le nom du programme à exécuter en surbrillance et on appuie sur `F6 (RUN)`.

1.3.3 Traduction HP48 HP49G mode RPN

Si le programme n'a pas de paramètres, il suffit de taper son nom dans la ligne de commande ou d'utiliser le menu `VAR`.
S'il y a des paramètres, on met les valeurs des paramètres sur la pile puis on tape le nom du programme.
Exemple :

```
45 75 PGCD
```

1.3.4 Traduction HP49G mode Algébrique

Si le programme n'a pas de paramètres, il suffit de taper son nom dans la ligne de commande ou d'utiliser le menu `VAR`.
S'il y a des paramètres, on fait suivre le nom du programme de parenthèses dans lesquelles on met les valeurs des paramètres séparées par une virgule.
Exemple :

```
PGCD(45,75)
```

1.3.5 Traduction TI 80

On appuie sur la touche `PRGM`, on laisse en surbrillance `EXEC`, et on tape le numéro du programme à exécuter.

1.3.6 Traduction TI 83+

On appuie sur la touche `PRGM`, on laisse en surbrillance `EXEC`, et on tape le numéro du programme à exécuter.

1.3.7 Traduction TI89 92

Si le programme n'a pas de paramètres, il suffit de taper son nom dans la ligne de commande.
S'il y a des paramètres, on fait suivre le nom du programme de parenthèses dans lesquelles on met les valeurs des paramètres séparées par une virgule.
Exemple :

```
PGCD(45,75)
```

1.3.8 Traduction SHARP EL9600

On appuie sur la touche `2nd MATRIX (PRGM)`, on met en surbrillance `EXEC`, et le programme à exécuter avec le stylet ou on utilise les flèches ou on tape le numéro (à deux chiffres) du programme à exécuter.

1.4 Comment améliorer puis sauver sous un autre nom un programme

1.4.1 Traduction Casio

Dans l'écran `Program List` (`MENU PRGM EXE`), il suffit de mettre en surbrillance le nom du programme à changer et de taper sur `F6` puis `F2` pour `REN` (REName) du bandeau.
Mais cela ne duplique pas le programme !!! on doit donc tout retaper pour améliorer...

1.4.2 Traduction HP38G HP40G

Pour modifier un programme (sans vouloir sauver l'ancien) on ouvre le catalogue de programmes, en appuyant sur les touches `shift 0 (PROGRAM)`.
Il apparait alors un écran contenant la liste des programmes disponibles et un bandeau (`EDIT NEW SEND RECV RUN`).
On met le nom du programme à modifier en surbrillance et on appuie sur `F1 (EDIT)`.
Si vous voulez avoir à la fois l'ancien et le nouveau programme il faut :
-ouvrir le catalogue de programmes (`shift 0 (PROGRAM)`)
-appuyer sur `F2 (NEW)` et taper le nom du programme modifié puis `F6 (OK)`.
L'éditeur s'ouvre, on appuie alors sur `VAR` puis `A..Z 5 (P)` pour mettre `Program` en surbrillance.
Avec les flèches mettre le nom du programme à modifier en surbrillance et appuyer sur `F4 (VALUE)` (pour cocher `VALUE` du bandeau) puis `F6 (OK)`.
Cela recopie le texte du programme dans l'éditeur.

1.4.3 Traduction HP48 HP49G mode RPN

On tape :
`'NOMDUPROGRAMME' RCL` puis `edit` du bandeau.
On fait les améliorations et on fait suivre le dernier ≫ par :

`'NOUVEAUNOM' STO▷`

1.4.4 Traduction HP49G mode Algébrique

On tape :
`RCL('NOMDUPROGRAMME')` puis `edit` du bandeau.
On fait les améliorations et on fait suivre le dernier ≫ par :

`STO▷ NOUVEAUNOM`

1.4.5 Traduction TI 80

Je n'ai pas trouvé comment faire...

1.4.6 Traduction TI 83+

Vous commencez comme si vous alliez écrire un nouveau programme (`PRGM NEW` puis le nom du nouveau programme).
Vous êtes alors dans l'éditeur, vous avez la possibilité d'éditer n'importe quel programme avec :
`2nd STO (RCL)` puis `PRGM`, puis, mettre en surbrillance `EXEC` puis, le numéro du programme à rappeler.
En bas de l'éditeur il s'écrit : `Rcl prgmPGCD` (par exemple).
`ENTER` ramène alors toutes les lignes du programme `PGCD` dans l'éditeur.

1.4.7 Traduction TI89 92

Vous ouvrez l'éditeur avec le programme que vous voulez améliorer puis sauver sous un autre nom :
`APPS` puis `7 (Program Editor)` puis `2 (Open)` puis mettre le nom du programme à modifier dans `Variable` (appuyer sur `->` pour avoir accès à tous les noms de vos programmes, sélectionner celui à modifier avec les flèches et `ENTER`), puis `ENTER`.
Ensuite, il suffit de taper sur ♦ `S` pour sauver cet éditeur sous un autre nom : `SAVE COPY AS` (on met le nouveau nom dans `Variable`).
Vous pouvez faire vos modifications puis ♦ `Q` (`HOME`) vous sauve votre programme modifié sous le nouveau nom, et vous revenez à l'écran `HOME`.

1.4.8 Traduction SHARP EL9600

On peut copier une ligne grâce au sous menu `COPY` du menu `2nd MATRIX (PRGM)`
On ne peut pas dupliquer un programme...on doit donc retaper pour améliorer...

Chapitre 2

Les différentes instructions

2.1 Les commentaires

Il faut prendre l'habitude de commenter les programmes. En algorithmique un commentaire commence par `//` et se termine par un passage à la ligne.

2.1.1 Traduction Casio

Il n'y en a pas!!!

2.1.2 Traduction HP38G HP40G HP48 HP49G

Le commentaire commence par :
`@` et se termine par un passage à la ligne.
Pour la `HP49G`, un commentaire commence par `@` et se termine par un passage à la ligne ou est entouré de deux `@`.
Le caractère `@` est obtenu en tapant `shift-rouge ENTER`
Attention!!! le compilateur efface les commentaires... donc pour garder vos commentaires, il faut écrire votre programme sous la forme d'un texte qu'il faut ensuite compiler avec `STR` → ce qui complique un peu...

2.1.3 Traduction TI80 83+

Il n'y en a pas!!!

2.1.4 Traduction TI89 92

Le commentaire commence par
© (F2 9) et se termine par un passage à la ligne.

2.1.5 Traduction SHARP EL9600

Les commentaires sont précédés de `Rem` (que l'on trouve dans le sous menu `PRGM` du menu `2nd MATRIX (PRGM)` lorsqu'on est en train d'éditer un programme).

2.2 Les variables

2.2.1 Leurs noms

Ce sont les endroits où l'on peut stocker des valeurs, des nombres, des expressions. Les noms des variables sont soit prédéfinis soit plus libres.
Par exemple chez `CASIO, HP38G, HP40G, TI80, TI83+ et SHARP-EL9600` on n'a droit qu'aux 26 lettres de l'alphabet pour stocker des nombres réels.
Mais avec la `TI 89`, la `TI92`, la `HP48` ou la `HP49G` on peut utiliser des noms parfois limités à 8 caractères.

2.2.2 Notion de variables locales

Cette notion n'existe pas pour les calculatrices `CASIO, HP38, HP40G, TI80, TI83+ et SHARP-EL9600`.
Pour la `TI92` il faut définir les variables locales en début de programme en écrivant :
`:local a,b`
La `HP48` ou `HP49G` peut utiliser des variables locales.
Les variables locales sont déclarées et initialisées (initialisation obligatoire !) grâce à → `(shift-rouge 0)`
Pour la `HP48` ou `HP49G mode RPN` il faut définir et initialiser les variables locales en écrivant :

$$\ll 1\ 2\ \rightarrow\ \texttt{A B} \ll \text{corps du programme} \gg\gg$$

pour initialiser `A` à 1 et `B` à 2.
En mode `RPN` on peut définir et initialiser plusieurs variables locales à la fois (cf exemple ci-dessus).
Pour la `HP49G mode Algébrique` chaque déclaration doit être suivie par un sous programme (délimiteurs ≪≫) en écrivant :

$$\ll 1\ \rightarrow\ \texttt{A} \ll 2\ \rightarrow\ \texttt{B} \ll \text{corps du programme} \gg\gg\gg$$

La flèche doit être entourée d'espaces, ces espaces sont mis automatiquement quand on n'est pas en mode `Alpha`.
Exemple :

$$\ll 3.14\ \rightarrow\ \texttt{PI} \ll 2 * \texttt{PI} * \texttt{R} \gg\gg \texttt{STO}\triangleright\ \texttt{PER}$$

Dans cet exemple, on a écrit le programme `PER`.
`PI` est une variable locale qui est déclarée et affectée par `3.14` → `PI`. Cette variable est locale pour le programme qui suit sa déclaration (ici ≪ `2 * PI * R` ≫).
Par contre, `R` est une variable globale (qui doit exister avant l'exécution du programme `PER`). Si, au cours d'un programme, on veut stocker une valeur dans une variable (locale ou globale) il faut bien sûr utiliser `STO▷`.

2.2.3 Notion de paramètres

Quand on écrit une fonction il est possible d'utiliser des paramètres.
Par exemple si `A` et `B` sont les paramètres de la fonction `PGCD` on écrit :
`PGCD(A,B)`

Ces paramètres se comportent comme des variables locales, la seule différence est qu'ils sont initialisés lors de l'appel de la fonction. L'exécution se fait en demandant par exemple : `PGCD(15,75)`
Il n'est pas possible d'écrire des fonctions sur toutes les calculatrices.
Pour les `TI 89 92` on met le nom des paramètres dans le nom de la fonction par exemple :
`:addition(a,b)`
Par exemple, pour la `HP48` ou `HP49G mode RPN` si on veut que `R` soit le paramètre de la fonction `PER` on écrit :

$$\ll \rightarrow \texttt{R} \ll \texttt{3.14} \rightarrow \texttt{PI} \ll \texttt{2 PI} * \texttt{R} * \gg\gg\gg' \texttt{PER}' \texttt{ STO}\triangleright$$

L'exécution se fait en demandant par exemple : `5 PER`.
Pour la `HP49G mode Algébrique` si on veut que `R` soit le paramètre de la fonction `PER` on écrit :

$$\ll \rightarrow \texttt{R} \ll \texttt{3.14} \rightarrow \texttt{PI} \ll \texttt{2} * \texttt{PI} * \texttt{R} \gg\gg\gg \texttt{STO} \triangleright \texttt{ PER}$$

L'exécution se fait en demandant par exemple : `PER(5)`.
La syntaxe des `HP48` et `HP49G` en mode `RPN` et en `Algébrique` est :

$$\ll \rightarrow \texttt{A B} \ll \ldots \gg\gg$$

pour écrire une fonction à deux paramètres. **Remarque** Le langage des `HP48/49` est fonctionnel puisque on peut passer des programmes ou des fonctions en paramètre.

2.3 Les Entrées

Pour que l'utilisateur puisse entrer une valeur dans la variable `A` au cours de l'exécution d'un programme, on écrira, en algorithmique :
`saisir A`
Et pour entrer des valeurs dans `A` et `B` on écrira :
`saisir A,B`

2.3.1 Traduction Casio

```
"A="?->A :
```

2.3.2 Traduction HP38G HP40G

```
INPUT A;"TITRE";"A=";;0 :
```
pour la `HP 40G` on a aussi :
```
PROMPT A :
```

2.3.3 Traduction HP48 HP49G mode RPN

```
"A" "" INPUT STR-> EVAL 'A' STO
```
ou pour la `HP49G mode RPN`
```
'A' PROMPTSTO
```

2.3.4 Traduction HP49G mode Algébrique

Pour entrer une variable `A`, on écrit :
`...PROMPTSTO('A')`

2.3.5 Traduction TI 80, 82, 83, 83+, 89, 92

`:Prompt A`
`:Prompt A,B`
ou encore :
`:Input "A=",A`

2.3.6 Traduction SHARP EL9600

`Input A`
ou
`Input A`
`Input B`
La machine met alors automatiquement `A=?` (ou `A=?` puis `B=?`).

2.4 Les Sorties

En algorithmique on écrit :
`Afficher "A=",A`

2.4.1 Traduction Casio

`"A=" :A` △
`ClrText` efface l'écran.
△ arrête l'affichage et sert aussi de fin d'instruction.

2.4.2 Traduction HP38G HP40G

`DISP 3;"A="A` : 3 représente le numéro de la ligne où `A` sera affiché
ou
`MSGBOX "A="A` :
`ERASE` : efface l'écran.
`FREEZE` : gèle l'affichage.

2.4.3 Traduction HP48 ou HP49G mode RPN

En général on affiche simplement les résultats sur la pile pour une réutilisation éventuelle, on écrira simplement :
`A B`
On peut aussi afficher le résultat tagué, on écrira alors :
`A "A=" ->TAG`
`HALT` arrête le programme et `Shift-rouge ON (CONT)` le continue.

2.4.4 Traduction HP49G mode Algébrique

Seul le dernier résultat s'inscrit dans l'historique.
Pour des résultats intermédiaires on tape :

$$\texttt{DISP}(\text{"A = "} + \texttt{A}, 3)$$

3 représente le numéro de la ligne
ou

$$\texttt{MSGBOX}(\text{"A = "} + \rightarrow \texttt{STR}(\texttt{A}))$$

(ici le + effectue la concaténation de deux chaines de caractères)
`CLLCD()` efface l'écran.
`FREEZE(7)` gèle l'affichage et permet de visualiser les 7 lignes de l'affichage.

2.4.5 Traduction TI 80 82 83 83+ 89 92

```
 :Disp "A=",A
```
`:ClrIO` efface l'écran.
`:ClrHome` efface l'écran pour la `TI 83+`.
`:Pause` arrête le programme (on appuie sur `ENTER` pour reprendre l'exécution).

2.4.6 Traduction SHARP EL9600

`Print A` affiche la valeur de `A`
ou `Print "A` affiche `A`, puis
`Print A` affiche la valeur de `A`

2.5 La séquence d'instructions ou action

Une action est une séquence d'une ou plusieurs instructions.
En langage algorithmique, on utilisera l'espace ou le passage à la ligne pour terminer une instruction.

2.5.1 Traduction Casio

A la fin d'une instruction, il faut mettre un séparateur qui peut être :
`le retour à la ligne` ou `:` ou $\triangle$

2.5.2 Traduction HP38G HP40G

: indique la fin d'une instruction.

2.5.3 Traduction HP48 ou HP49G mode RPN

Comme en algorithmique, le séparateur est soit `l'espace` soit `le retour à la ligne`.

2.5.4 Traduction HP49G mode Algébrique

Pour la `HP49G`, le ; est un séparateur d' instructions.
Le ; s'obtient en tapant en même temps sur `shift-rouge SPC`.

2.5.5 Traduction TI 83+ 89 92

: indique la fin d'une instruction. Il faut noter qu'à chaque passage à la ligne le : est mis automatiquement.

2.5.6 Traduction SHARP EL9600

C'est `ENTER` qui est utilisé comme séparateur d'instructions.
Chaque instruction doit être écrite sur une seule ligne.

2.6 L'instruction d'affectation

L'affectation est utilisée pour stocker une valeur ou une expression dans une variable.
En algorithmique on écrira par exemple : `2*A->B`
pour stocker `2*A` dans `B`
Les calculatrices `CASIO, HP38G, HP40G, TI80, TI83+, TI89, TI92 et SHARP-EL9600` utilisent cette notation.
Selon les cas, la flèche est obtenue à l'aide de la touche `STO` ou de la touche
→
Avec la `HP48` ou `HP49G mode RPN`, il faut utiliser la notation postfixée et la commande `STO` :
`2 A * 'B' STO`
Pour la `HP49G mode Algébrique`, on utilise la touche `STO` qui se traduit à l'écran de la calculatrice par : ▷ (que l'on notera : `STO`▷)
Pour la `SHARP EL9600`, on utilise la touche `STO` qui se traduit à l'écran de la calculatrice par : ⇒

2.7 Les instructions conditionnelles

`Si` *`condition`* `alors` *`action`* `fsi`
`Si` *`condition`* `alors` *`action1`* `sinon` *`action2`* `fsi`
Exemple :
`Si A = 10 ou A < B alors B-A->B sinon A-B->A fsi`

2.7.1 Traduction Casio

`If` *`condition`* `:Then` *`action`* `: IfEnd :`
`If` *`condition`* `:Then` *`action1`* `: Else :` *`action2`* `: IfEnd :`
Exemple :
`If A = 10 Or A < B :Then B-A->B : Else : A-B->A : IfEnd :`

2.7.2 Traduction HP38G HP40G

IF *condition* THEN *action* : END :
IF *condition* THEN *action1* : ELSE *action2* : END :
Exemple (Attention au == pour traduire la condition d'égalité) :

```
IF A == 10 OR A < B THEN B-A->B : ELSE A-B->A : END :
```

2.7.3 Traduction HP48 ou HP49G mode RPN

IF *condition* THEN *action* END
IF *condition* THEN *action1* ELSE *action2* END
Attention on utilise la notation postfixée et == pour traduire la condition d'égalité.
On écrit pour traduire l'exemple :

```
IF A 10 == A B < OR THEN B A - 'B' STO ELSE A B - 'A' STO END
```

on peut aussi écrire :

```
IF '(A==10) OR (A < B)' THEN ...
```

2.7.4 Traduction HP49G mode Algébrique

IF *condition* THEN *action* END
IF *condition* THEN *action1* ELSE *action2* END
Exemple (Attention au == pour traduire la condition d'égalité) :

```
IF A == 10 OR A < B THEN B-A STO▷ B ELSE A-B STO▷ A END
```

2.7.5 Traduction TI80

:If *condition* :Then : *action* : End
:If *condition* :Then : *action1* : Else : *action2* : End
Exemple :
Attention `or` n'existepas il faut donc écrire (cf p 23) : `:IF A = 10 :THEN : B-A->B :ELSE :IF A<B :THEN : B-A->B :ELSE : A-B->A :END :END`

2.7.6 Traduction TI82 83+

:If *condition* :Then : *action* : End
:If *condition* :Then : *action1* : Else : *action2* : End
Exemple :

```
:If A = 10 or A < B : Then : B-A->B : Else : A-B->A : End
```

2.7.7 Traduction TI89 92

:If *condition* Then : *action* : EndIf
:If *condition* Then : *action1* : Else : *action2* : EndIf
Exemple :

```
:If A = 10 or A < B Then : B-A->B : Else : A-B->A : EndIf
```

2.7.8 Traduction SHARP EL9600

If *non condition* Goto FSI
action

```
Label FSI

If non condition Goto SINON
action1
Goto FSI
Label SINON
action2
Label FSI
```

2.8 Les instructions "Pour"

```
Pour I de A à B faire action fpour
Pour I de A à B (pas P) faire action fpour
```

2.8.1 Traduction Casio

```
For A->I To B : action : Next
For A->I To B Step P : action : Next
```

2.8.2 Traduction HP38G HP40G

```
FOR I = A TO B STEP 1 ; action : END :
FOR I = A TO B STEP P ; action : END :
```

2.8.3 Traduction HP48 ou HP49G mode RPN

```
A B FOR I action NEXT
A B FOR I action P STEP
```

2.8.4 Traduction HP49G mode Algébrique

```
FOR (I, A, B) action NEXT
FOR (I, A, B ) action STEP P
```
L'instruction `FOR` déclare `I` comme variable locale et l'initialise automatiquement.

2.8.5 Traduction TI80 TI82 83+

```
 :For (I,A,B ) : action : End
:For (I,A,B,P ) : action : End
```

2.8.6 Traduction TI89 92

```
 :For I,A,B : action : EndFor
:For I,A,B,P : action : EndFor
```

2.8.7 Traduction SHARP EL9600

A ⇒ I
Label DPOUR
If I>B Goto FPOUR
action1
I + 1 ⇒ I
Goto DPOUR
Label FPOUR

2.9 L'instruction "Repeter"

Repeter *action* jusqua *condition*

2.9.1 Traduction Casio

Do : *action* : LpWhile *condition*

2.9.2 Traduction HP38 HP40G

DO *action* : UNTIL *condition* END :

2.9.3 Traduction HP48 ou HP49G mode RPN

DO *action* UNTIL *condition* END

2.9.4 Traduction HP49G mode Algébrique

DO *action* UNTIL *condition* END

2.9.5 Traduction TI80

:LBL 1
:*action* :IF *non condition* :THEN :GOTO 1
:END

2.9.6 Traduction TI82 83+

:Repeat *condition* : *action* : End

2.9.7 Traduction TI89 92

:Loop :*action* :If *condition* :Exit :EndLoop

2.9.8 Traduction SHARP EL9600

Label REPETER
action
If *non condition* Goto REPETER

2.10 L'instruction "Tant que"

Tant que *condition* faire *action* ftantque

2.10.1 Traduction Casio

While *condition* : *action* : WhileEnd :

2.10.2 Traduction HP38G HP40G

WHILE *condition* REPEAT *action* : END :

2.10.3 Traduction HP48 ou HP49G mode RPN

WHILE *condition* REPEAT *action* END

2.10.4 Traduction HP49G mode Algébrique

WHILE *condition* REPEAT *action* END

2.10.5 Traduction TI80

```
 :LBL 1
:IF condition :THEN :action
:GOTO 1
:END
```

2.10.6 Traduction TI82 83+

:While *condition* : *action* : End

2.10.7 Traduction TI89 92

:While *condition* : *action* : EndWhile

2.10.8 Traduction SHARP EL9600

```
Label DTANTQUE
If non condition Goto FTANTQUE
action
Goto DTANTQUE
Label FTANTQUE
```

2.11 Les conditions ou expressions booléennes

Une condition est une fonction qui a comme valeur un booléen, à savoir elle est soit `vraie` soit `fausse`.

2.11.1 Les opérateurs relationnels

Pour exprimer une condition simple on utilise les opérateurs :
`= > >` $\leq \geq \neq$
Attention pour les calculatrices HP l'égalité se traduit comme en langage `C` par :
`==`

2.11.2 Les opérateurs logiques

Pour traduire des conditions complexes, on utilise les opérateurs logiques :
`ou et non`
qui se traduisent sur les calculatrices par :
`or and not`
Attention ! ! ! Je ne les ai pas trouvés sur la `TI80` et il faut alors traduire :
`Si` *`condition1`* `et` *`condition2`* `alors` *`action1`* `sinon` *`action2`* `fsi`
par :
`:IF` *`condition1`* `:THEN :IF` *`condition2`* `:THEN`
`:`*`action1`* `:ELSE :`*`action2`* `:END`
`:ELSE :`*`action2`* `:END`
`Si` *`condition1`* `ou` *`condition2`* `alors` *`action1`* `sinon` *`action2`* `fsi`
par :
`:IF` *`condition1`* `:THEN`
`:`*`action1`* `:ELSE :IF` *`condition2`* `:THEN :`*`action1`*
`:ELSE :`*`action2`* `:END :END`
Chez `Sharp` il faut parenthéser et écrire (A) or (B) par exemple.

2.12 Les fonctions

Dans une fonction on ne fait pas de saisie de données : on utilise des paramètres qui seront initialisés lors de l'appel.
Dans une fonction on veut pouvoir réutiliser le résultat :
on n'utilise pas la commande `affichage` mais la commande `retourne`.
On écrit par exemple en algorithmique :

```
fonction addition(A,B)
retourne A+B
ffonction
```

Cela signifie que :
- Si on fait exécuter la fonction, ce qui se trouve juste après `retourne` sera affiché, mais les instructions qui suivent `retourne` seront ignorées.
- On peut utiliser la fonction dans une expression.

On ne peut pas écrire de fonctions avec les calculatrices `Casio SHARPEL9600 HP38G, HP40G TI83+`.

2.12.1 Traduction HP48 ou HP49G mode RPN

Pour la `HP48` ou la `HP49G mode RPN`, on suppose que les arguments de la fonction sont mis sur la pile avant l'appel de la fonction.

Dans l'écriture de la fonction les arguments sont des variables locales qui seront initialisées par les élements mis sur la pile. Le résultat de la fonction est alors mis sur la pile.
L'exemple se traduit par :
≪→ A B
≪ A B + ≫
≫
'ADDITION' STO
puis on met les valeurs de A et B au niveau 1 et 2 de la pile, aprés l'exécution d'ADDITION leur somme se trouvera au niveau 1 de la pile.
N.B. pour des fonctions simples, on peut éviter les variables locales par exemple ≪ + ≫ 'ADDITION' STO

2.12.2 Traduction HP49G mode Algébrique

Pour la HP49G :
L'exemple se traduit par :
≪→ A B
≪ A + B ≫
≫
STO▷ ADDITION
Puis, on tape :

ADDITION(4, 5)

2.12.3 Traduction TI89 92

```
:addition(a,b)
:Func
:Return a+b
:EndFunc
```

2.13 Les listes

On utilise les { } pour délimiter une liste.
Attention!!! En algorithmique on a choisit cette notation car c'est celle qui est employée par les calculatrices...à ne pas confondre avec la notion d'ensemble en mathématiques : dans un ensemble l'ordre des éléments n'a pas d'importance mais dans une liste l'ordre est important... Par exemple {} désigne la liste vide et {1, 2, 3} est une liste de 3 éléments.
Concat sera utilisé pour concaténer 2 listes ou une liste et un élément ou un élément et une liste :
{1, 2, 3}->TAB
Concat(TAB, 4) ->TAB (maintenant TAB désigne {1, 2, 3, 4}
TAB[2] désigne le deuxième élément de TAB ici 2.

2.13.1 Traduction Casio

Les variables listes ont pour noms : List 1, List 2, ... List 6.
Pour travailler avec des listes, il faut donner au départ le nombre d'éléments

de la liste avec la commande `Dim List` par exemple :
`10->Dim List 1` (`List 1` désigne alors une liste de 10 éléments nuls).
On peut utiliser les commandes suivantes :
`seq(i*i, i, 1, 10, 2)` désigne la liste des carrés des 5 premiers entiers impairs.
`List 1[i]` désigne le ième élément de la liste `List 1`.

2.13.2 Traduction HP38G HP40G

Ici les listes peuvent avoir des longueurs non définies à l'avance.
Les variables listes ont pour noms : `L0, L1, L2, ... L9`.
On utilise les { } pour délimiter une liste.
Par exemple {1, 2, 3} est une liste de 3 éléments.
Mais {} ne désigne pas la liste vide :
- avec la `HP38G` il faut utiliser la commande `CLEAR L1` pour initialiser la liste `L1` à vide.
- avec la `HP40G` il faut utiliser la commande `SYSEVAL 259588` pour initialiser la liste `L0` à vide (`SYSEVAL 259589` pour initialiser `L1` à vide etc...).
On peut aussi utiliser la commande `SUB` :
`SUB L0 ;L0 ;2 ;1` initialise la liste `L0` à vide.
Voici quelques commandes utiles :
`MAKELIST(I*I, I , 1, 10, 2)` désigne la liste des carrés des 5 premiers entiers impairs (2 indique le pas de `I`).
`L1(I)` désigne le `I`ème élément de la liste.
`CONCAT (L1, {5})` désigne une liste ayant l'élément 5 en plus des éléments de la liste `L1`.
`SUB L1 ;L0 ;2 ;4` met dans `L1` la suite extraite de `L0` composée des éléments d'indice allant de 2 à 4 (si `L0={1,2,3,4,5}` on aura `L1={2,3,4}`.

2.13.3 Traduction HP48 ou HP49G mode RPN

Ici les listes peuvent avoir des longueurs non définies à l'avance.
On utilise les {} pour délimiter une liste.
Par exemple {1 2 3} est une liste de 3 éléments et {} désigne la liste vide.
On obtient le `P`ième élément de `L` sur la pile avec :
`L P GET`
Si on veut modifier le `P`ième élément de `L` (par exemple le mettre à 0) on écrira :
`'L' P 0 PUT` ou `L P 0 PUT 'L' STO`
En effet `L P 0 PUT` renvoie sur la pile la liste modifiée alors que :
`'L' P 0 PUT` modifie la liste `L`.
Pour concaténer deux listes ou une liste et un élément on utilise le `+` ou on utilise la commande `AUGMENT`.
Remarque : c'est la commande `ADD`) qui permet d'ajouter deux listes de mêmes longueurs.

2.13.4 Traduction HP49G mode Algébrique

Pour la HP49G en mode algébrique, les listes peuvent avoir des longueurs non définies à l'avance.
On utilise les {} pour délimiter une liste.
Par exemple {1 2 3} est une liste de 3 éléments et {} désigne la liste vide.
On obtient le Pième élément de L sur la pile avec :
L[P] ou L(P) ou GET (L, P)
Si on veut modifier le Pième élément de L (par exemple le mettre à 0) on écrira :
PUT(L, P, 0) STO▷ L
ou
PUT('L', P, 0)
En effet PUT(L, P, 0) renvoie la liste modifiée (sans modifier L) alors que : PUT ('L', P, 0) modifie la liste L.
Pour concaténer deux listes ou une liste et un élément on utilise le + (et pour ajouter deux listes de mêmes longueurs on utilise la commande ADD).
La commade SEQ permet de constituer une liste, on tape :

$$\texttt{SEQ}(\texttt{'X}*\texttt{X'},\texttt{'X'},4,10,1)$$

on obtient :

$$\{16,25,36,49,64,81,100\}$$

2.13.5 Traduction TI80

Les variables listes ont comme noms L1, L2, L3, L4, L5, L6.
La commande DIM permet de créer une nouvelle liste (faite de zéros) ou de redimensionner une liste existante (numéro 3 du menu OPS de la touche 2nd STAT (LIST)), par exemple :
N− > DIM L1 : il y a deux possibilités,
soit L_1 n'existait pas alors L_1 est créée et contient une liste de zéros de longueur N,
soit L_1 existait, cette commande donne alors à L_1 la dimension N (soit en supprimant des termes, soit en rajoutant des zéros).
La commande SEQ permet de générer une liste (numéro 4 du menu OPS de la touche 2nd STAT (LIST)) :
$\texttt{SEQ}(\texttt{X}^2,\texttt{X},0,10,2)- > \texttt{L1}$ va par exemple créer la liste :
{0, 4, 16, 36, 64, 100} c'est à dire la liste des carrés de 0 à 10 avec un pas de 2 que l'on stocke dans L1.

2.13.6 Traduction TI83+

Les variables listes ont des noms prédéfinis $L_1, L_2, L_3, L_4, L_5, L_6$ ou bien peuvent avoir un nom de 1 à 5 caractères mais alors il faudra faire précéder ce nom du symbole ⌞ (que l'on trouve dans 2nd LIST OPS B) pour désigner cette variable.
Mais {} ne désigne pas la liste vide, il faut utiliser la commande :
$\texttt{ClrList}L_1$ (en position 4 du menu EDIT de la touche STAT) vide la liste L_1

(attention!!! dans un programme cela n'initialise pas la liste L_1 à vide).
La commande dim permet de créer une nouvelle liste (faite de zéros) ou de redimensionner une liste existante (en position 3 du menu OPS de la touche 2nd STAT (LIST)) :
$N->dim(L_1)$: il y a deux possibilités,
soit L_1 n'existait pas alors L_1 est créée et contient une liste de zéros de longueur N,
soit L_1 existait, cette commande donne alors à L_1 la dimension N (soit en supprimant des termes, soit en rajoutant des zéros).
La commande seq permet de générer une liste (en position 5 du menu OPS de la touche 2nd STAT (LIST)) :
$seq(X^2, X, 0, 10, 2)->TAB$ va par exemple créer la liste :
{0, 4, 16, 36, 64, 100} c'est à dire la liste des carrés de 0 à 10 avec un pas de 2 que l'on stocke dans TAB.
$\llcorner TAB(2)$ désigne le deuxième élément de TAB ici 4.
On peut aussi écrire :
$2->\llcorner TAB(2)$
La liste TAB est alors {0, 2, 16, 36, 64, 100}
ou si L_1 est de longueur N on peut rajouter un élément (par exemple 25) à L_1 en écrivant :
$25->L_1(N+1)$
augment permet de concaténer deux listes (en position 9 du menu OPS de la touche 2nd STAT (LIST)).

2.13.7 Traduction TI89-92

augment permet de concaténer deux listes.
{} désigne la liste vide. Pour travailler avec des listes, on peut initialiser une liste de n éléments avec la commande newlist, par exemple :
newlist(10)->L (L est alors une liste de 10 éléments nuls).
On peut utiliser les commandes suivantes :
seq(i*i, i, 1, 10) qui désigne la liste des carrés des 10 premiers entiers, ou seq(i*i, i, 0, 10, 2) qui désigne la liste des carrés des 5 premiers entiers pairs (le pas est ici égal à 2).
Exemple :
$seq(i*i, i, 0, 10, 2)->L$ va par exemple créer la liste :
{0, 4, 16, 36, 64, 100} c'est à dire la liste des carrés de 0 à 10 avec un pas de 2 que l'on stocke dans L.
L[i] qui désigne le ième élément de la liste L.
On peut aussi écrire :
2->L[2]
La liste L est alors {0, 2, 16, 36, 64, 100}
ou si L est de longueur n on peut rajouter un élément (par exemple 121) à L en écrivant :
121 -> L[n+1]
Dans l'exemple précédent n=6 on peut donc écrire :
121 -> L[7] (L est alors égale à {0, 2, 16, 36, 64, 100, 121}).
left (L, 5) désigne les 5 premiers éléments de la liste L.

2.13.8 Traduction SHARP EL9600

Ici les listes peuvent avoir des longueurs non définies à l'avance.
Les variables listes ont pour noms : `L1, L2, L3...L6`.
On utilise les { } pour délimiter une liste.
Par exemple {1, 2, 3} est une liste de 3 éléments.
Mais {} ne désigne pas la liste vide, il faut utiliser la commande :
`ClrList L1` (en position 4 dansle menu `OPE` de `STAT`) qui vide la liste `L1` (attention ! ! ! dans un programme cela n'initialise pas la liste `L1` à vide).
Voici quelques commandes utiles :
`seq` qui se trouve en position 5 dans le sous menu `OPE` de `2nd` × (`LIST`) :
`seq(X*X, 1, 10, 2)` qui désigne la liste des carrés des 5 premiers entiers impairs (2 indique le pas de la variable qui est toujours `X`).
`L1(I)` désigne le `I`ème élément de la liste.
`augment(L1, {5})` désigne une liste ayant l'élément 5 en plus des éléments de la liste `L1` (`augment` se trouve en position 8 dans le sous menu `OPE` de `2nd` × (`LIST`).
La commande `dim` permet de créer une nouvelle liste (faite de zéros) ou de redimensionner une liste existante :
`N ->dim(L1)` crée `L1` une liste de zéros de longueur `N`.
mais si `L1` existait, cette commande donne à `L1` la dimension `N` (soit en supprimant des termes, soit en rajoutant des zéros).
Si `L1` est de longueur `N` on peut rajouter un élément (par exemple 25) à `L1` en écrivant :

```
25 -> L1(N+1)
```

2.14 Un exemple : le crible d'Eratosthène

2.14.1 Description

Pour trouver les nombres premiers inférieurs ou égaux à N :

1. On écrit les nombres de 1 à N dans une liste.
2. On barre 1 et on met 2 dans la case P .
   Si $P \times P \leq N$ il faut traiter les éléments de P à N.
3. On barre tous les multiples de P à partir de $P.P$.
4. On augmente P de 1.
   Si $P \times P$ est inférieur ou égal à N, il reste à traiter les éléments non barrés de P à N.
5. On appelle P le plus petit élément non barré de la liste.
6. On refait les points 3 4 5 tant que $P \times P$ reste inférieur ou égal à N.

2.14.2 Écriture de l'algorithme

```
Fonction crible(N)
local TAB PREM I P
// TAB et PREM sont des listes
{} ->TAB
```

```
{} ->PREM
pour I de 2 \‘a N faire
  concat(TAB, I) -> TAB
fpour
concat(0, TAB) -> TAB
2 -> P
// On a fait les points 1 et 2
//barrer 1 a \’et\’e r\’ealis\’e en le rempla\c{c}ant par 0
//TAB est la liste 0 2 3 4 ...N
tant que P*P ≤ N faire
  pour I de P \‘a E(N/P) faire
//E(N/P) d\’esigne la partie enti\‘ere de N/P
    0 -> TAB[I*P]
  fpour
// On a barr\’e tous les multiples de P \‘a partir de P*P
  P+1 -> P
//On cherche le plus petit nombre <= N non barr\’e (non nul) entre P et N
  tant que (P*P ≤ N) et (TAB[P]=0) faire
    P+1 -> P
  ftantque
ftantque
//on \’ecrit le r\’esultat dans une liste PREM
pour I de 2 \‘a N faire
  si TAB[I] ≠ 0 alors
    concat(PREM, I) -> PREM
  fsi
fpour
r\’esultat: PREM
```

2.14.3 Traduction Casio

Voici le programme CRIBLE :

```
‘‘N’’?->N
N->Dim List 1
N->Dim List 2
Seq(I,I,1,N,1)->List 1
0->List 1[1]
2->P
While P*P≤N
For P->I To Int (N/P)
0->List 1[I*P]
Next
P+1->P
While P*P≤N And List 1[P]=0
```

```
P+1->P
WhileEnd
WhileEnd
0->P
For 2->I To N
If List 1[I]≠0
Then P+1->P
I->List 2[P]
IfEnd
Next
P△
List 2△
```

2.14.4 Traduction HP38G HP40G

Voici le programme CRIBLE :
L'utilisateur doit entrer la valeur de N.
A la fin la liste L2 contient les nombres premiers inférieurs ou égaux à N.

```
INPUT N;"CRIBLE";"N=";;10:
ERASE:
MAKELIST(I,I,1,N,1) -> L1:
0 -> L1(1):
2->P:
WHILE P*P ≤ N REPEAT
  FOR I = P TO INT(N/P) STEP 1;
    0->L1(I*P):
  END:
  DISP 3;""L1:
  P+1->P:
  WHILE P*P ≤ N AND L1(P) == 0 REPEAT
    P+1->P:
  END:
END:
{2}->L2:
@ on sait que 2 est premier
FOR I=3 TO N  STEP 1;
 IF L1(I) ≠ 0 THEN
  CONCAT(L2,{I}) ->L2:
  END:
END:
DISP 3 ;"PREM" L2:
FREEZE:
```

2.14.5 Traduction HP48 ou HP49G mode RPN

Voici le programme CRIBLE :
N est le paramètre qui doit être mis sur la pile.

Les variables locales sont :
P et I qui sont des entiers,
TA et PREM qui sont des listes.

```
≪ {} {} 2 1 → N TA PREM P I
  ≪ 0 'X' 'X' 2 N 1 SEQ + 'TA' STO
    WHILE P P * N ≤ REPEAT
       P N P / FLOOR FOR I
          TA I P * 0 PUT 'TA' STO
       NEXT
       1 'P' STO+
       WHILE P P * N ≤ TA P GET 0 == AND REPEAT
          1 'P' STO+
       END
    END
    2 N FOR I
       IF TA I GET 0 ≠ THEN
          I 'PREM' STO+
       END
    NEXT
    PREM
  ≫
≫
```

Puis on stocke ce programme dans `CRIBLE ('CRIBLE' STO)`.

2.14.6 Traduction HP49G mode Algébrique

Voici le programme CRIBLE écrit pour la `HP49G mode Algébrique` :
L'utilisateur doit taper par exemple :
`CRIBLE(100)`.

```
≪ → N
  ≪ 0+SEQ('I','I',1,N,1) → TA
    ≪ 2 → P
      ≪ WHILE P * P ≤ N REPEAT
        FOR (I , P , FLOOR(N/P))
          PUT('TA',I*P,0)
        NEXT ;
         P + 1 STO ▷ P;
        WHILE P*P ≤ N AND TA[P] == 0 REPEAT
           P + 1 STO ▷ P
        END
      END ;
      {2} → PREM
        ≪ FOR (I,3,N)
            IF TA[I] ≠ 0 THEN
              PREM + I STO ▷ PREM;
           END
        NEXT ;
        PREM
```

```
          ≫
        ≫
      ≫
    ≫
≫ STO▷ CRIBLE
```

2.14.7 Traduction TI80

Voici le programme CRIBLE : on entre N et ensuite la liste L_2 est égale à la liste des nombres premiers inférieurs ou égaux à N.

```
PROGRAM :CRIBLE
:INPUT N
:SEQ(I,I,1,N,1)->L1
:0->L1(1)
:2 -> P
:LBL 1
:IF P*P ≤ N
:THEN
:FOR (I,P,INT(N/P))
:0->L1(I*P)
:END
:P+1 -> P
:LBL 2
:IF (P*P)≤N
:THEN
:IF L1(P)=0
:THEN
:P+1 -> P
:GOTO 2
:END
:END
:GOTO 1
:END
:0 -> P
:FOR (I,2,N)
:IF L1(I)≠0
:THEN
:P+1 ->P
:I->L2(P)
:END
:END
:L2
```

Il n'y a pas ici de notion de variables locales ou de programmes avec des paramètres : toutes les variables sont globales.

Je n'ai pas trouvé la fonction booléenne `AND` : j'ai donc mis deux `IF` successifs.

La dernière instruction permet d'afficher la liste $L2$ des nombres premiers inférieurs ou égaux à N. Si à l'affichage la liste $L2$ n'est pas visible en entier, il faut appuyer sur ▷ pour faire défiler la liste.

2.14.8 Traduction TI83+

Voici le programme CRIBLE : on entre N et ensuite la liste L_2 est égale à la liste des nombres premiers inférieurs ou égaux à N.

```
PROGRAM :CRIBLE
:Input N
:seq(I,I,1,N)->L1
:0->L1(1)
:2 -> P
:While P*P ≤ N
:For (I,P,int(N/P))
:0->L1(I*P)
:End
:P+1 -> P
:While (P*P)≤N and L1(P)=0
:P+1 -> P
:End
:End
:0 -> P
:For (I,2,N)
:If L1(I)≠0
:Then
:P+1 ->P
:I->L2(P)
:End
:End
:P->dim(L2)
:Disp L2
```

Il n'y a pas ici de notions de variables locales ou de programmes avec des paramètres : toutes les variables sont globales.
La dernière instruction permet d'afficher la liste L_2 des nombres premiers inférieurs ou égaux à N. Si à l'affichage la liste L_2 n'est pas visible en entier, il faut appuyer sur ▷ pour faire défiler la liste ou encore il faut se servir de la touche `STAT` puis `EDIT` qui permet d'afficher les listes L_i

2.14.9 Traduction TI89 92

Voici la fonction crible :
- n est le paramètre de cette fonction.
- crible(n) est égal à la liste des nombres premiers inférieurs ou égaux à n.

```
:crible(n)
:Func
:local tab,prem,i,p
:newList(n)->tab
:newList(n)->prem
:seq(i,i,1,n) ->tab
:0 -> tab[1]
:2 -> p
```

```
:While p*p ≤ n
:For i,p,floor(n/p)
:0 -> tab[i*p]
:EndFor
:p+1 -> p
:While p*p ≤ n and tab[p]=0
:p+1 -> p
:EndWhile
:EndWhile
:0 -> p
:For i,2,n
:If tab[i] ≠ 0 Then
:p+1 ->p
:i ->prem[p]
:EndIf
:EndFor
:Return left(prem,p)
:EndFunc
```

2.14.10 Traduction SHARP EL9600

L'écriture du programme CRIBLE est ici assez fastidieux : on a choisi des noms de labels courts mais parlant....(DTQ1 veut dire Début du TantQue $n^o 1$)

```
CRIBLE
Input N
{2} ⇒ L2
seq(X, 1, N) ⇒ L1
0 ⇒ L1
2 ⇒ P
Label DTQ1
If P*P>N Goto FTQ1
P ⇒ I
Label DP1
If I>int (N/P) Goto FP1
0 ⇒ L1(I * P)
I + 1 ⇒ I
Goto DP1
Label FP1
P + 1 ⇒ P
Label DTQ2
If (P * P > N) or (L1(P) ≠ 0) Goto FTQ2
P + 1 ⇒ P
Goto DTQ2
Label FTQ2
Goto DTQ1
Label FTQ1
```

```
3 ⇒ I
1 ⇒ P
Label DP2
If I>N Goto FP2
If L1(I) = 0) Goto FSI
P + 1 ⇒ P
I ⇒ L2(P)
Label FSI
I + 1 ⇒ I
Label FP2
Print L2
End
```

Il n'y a pas ici de notions de variables locales ou de programmes avec des paramètres : toutes les variables sont globales.
La dernière instruction permet d'afficher la liste $L2$ des nombres premiers inférieurs ou égaux à N. Si à l'affichage la liste L2 n'est pas visible en entier, il faut retourner dans l'écran `HOME`, puis taper L2, puis appuyer sur ▷ pour faire défiler la liste ou encore il faut se servir de la touche `STAT` puis `EDIT` et `ENTER` qui permet d'afficher les listes Li

2.14.11 Exercice

Pour améliorer l'algorithme ci-dessus, on n'écrit dans le crible que les nombres impairs.
Refaire alors l'algorithme puis, écrire le programme correspondant pour votre calculatrice.

Chapitre 3

Les programmes d'arithmétique

3.1 Calcul du PGCD par l'algorithme d'Euclide

Soient A et B deux entiers positifs dont on cherche le $PGCD$.
L'algorithme d'Euclide est basé sur la définition récursive du $PGCD$:

$$\begin{aligned} PGCD(A,0) &= A \\ PGCD(A,B) &= PGCD(B, A \mod B) \; si \; B \neq 0 \end{aligned}$$

où $A \mod B$ désigne le reste de la division euclidienne de A par B.
Voici la description de cet algorithme :
on effectue des divisions euclidiennes successives :

$$\begin{aligned} A = &\quad B \times Q_1 + R_1 \quad 0 \leq R_1 < B \\ B = &\quad R_1 \times Q_2 + R_2 \quad 0 \leq R_2 < R_1 \\ R_1 = &\quad R_2 \times Q_3 + R_3 \quad 0 \leq R_3 < R_2 \\ &..... \end{aligned}$$

Après un nombre fini d'étapes, il existe un entier n tel que : $R_n = 0$.
on a alors :
$PGCD(A,B) = PGCD(B,R_1) = ...$
$PGCD(R_{n-1}, R_n) = PGCD(R_{n-1}, 0) = R_{n-1}$

3.1.1 Traduction algorithmique

-Version itérative
Si B $\neq$ 0 on calcule R=A mod B, puis avec B dans le rôle de A (en mettant B dans A) et R dans le rôle de B (en mettant R dans B) on recommence jusqu'à ce que B=0, le PGCD est alors A.

```
Fonction PGCD(A,B)
Local R

tant que B ≠ 0 faire
```

```
  A mod B->R
  B->A
  R->B
ftantque
r\'esultat A
ffonction
```

-Version récursive

On écrit simplement la définition récursive vue plus haut.

```
Fonction PGCD(A,B)
Si B ≠ 0 alors
  r\'esultat PGCD(B,A mod B)
  sinon
  r\'esultat A
fsi
ffonction
```

3.1.2 Traduction Casio

-Version itérative :

On définit le programme INAB qui permet de rentrer deux nombres A et B :

```
‘‘A’’?->A
‘‘B’’?->B
ClrText
Return
```

Puis on tape le programme PGCD :

```
Prog ‘‘INAB’’
While B ≠ 0
  A-B*Intg(A/B)->R
  B->A
  R->B
WhileEnd
A△
```

-Pas de version récursive.

3.1.3 Traduction HP38G HP40G

-Version itérative

On écrit tout d'abord le sous-programme IN qui permet d'entrer deux nombres A et B :

```
INPUT A;"A";;;1:
INPUT B;"B";;;1:
ERASE:
```

puis on écrit le programme PGCD :

```
RUN IN:
DISP 3;"PGCD "{A,B}:
```

```
WHILE B ≠ 0 REPEAT
A MOD B ->R:
B ->A:
R ->B:
END:
DISP 4;"PGCD "A:
FREEZE:
```

-il n'y a pas de version récursive...mais on peut écrire le programme PGCDR :

```
DISP 3;"PGCD "{A,B}:
FREEZE:
IF B ≠ 0 THEN
A MOD B ->R:
B ->A:
R ->B:
PGCDR:
ELSE
DISP 3;"PGCD "A:
FREEZE:
END:
```

Il faut tout d'abord appeler le programme IN. Le programme PGCDR affiche les PGCD intermédiaires qui sont calculés. L'appel récursif PGCDR renvoie au programme PGCDR qu'il faut faire exécuter en appuyant sur le RUN du bandeau.

3.1.4 Traduction HP48 HP49G mode RPN

-Version itérative

```
≪ 0 → A B R
  ≪ WHILE B 0 ≠ REPEAT
       A B MOD 'R' STO
       B 'A' STO
       R 'B' STO
    END
    A
  ≫
≫
```

Puis on stocke ce programme dans `PGCD ('PGCD' STO)`.

-Version récursive

```
≪ 0 → A B
  ≪ IF B 0 ≠ THEN
       B A B MOD PGCDR
    ELSE
       A
    END
  ≫
```

≫
Puis on stocke ce programme dans `PGCDR ('PGCDR' STO)`. **N.B.** il est possible de ne travailler qu'avec la pile sans variables locales en écrivant :

```
≪ IF DUP 0 ≠ THEN
   SWAP OVER MOD PGCDR
   ELSE DROP
   END ≫
```

3.1.5 Traduction HP49G mode Algébrique

-Version itérative

```
≪ → A,B
  ≪ 0 → R
    ≪ WHILE B ≠ 0 REPEAT
         A MOD B STO▷ R;
         B STO▷ A;
         R STO▷ B
      END;
      A
    ≫
  ≫
≫ STO▷ PGCD
```

Puis par exemple, `PGCD(45,75)` pour l'exécuter.

-Version récursive

```
≪ → A,B
  ≪ IF B ≠ 0 THEN
       PGCDR(B,A MOD B)
    ELSE
       A
    END
  ≫
≫ STO▷ PGCDR
```

Puis par exemple, `PGCDR(45,75)` pour l'exécuter.
Remarque :
Si on utilise la fonction du calcul symbolique `IREMAINDER` à la place de `MOD` dans les programmes précédents, `PGCD` (ou `PGCDR`) peut alors avoir comme paramètres des entiers de Gauss (c'est à dire les nombres $a + i \cdot b$ avec a et b entiers relatifs).

3.1.6 Traduction TI80

-Version itérative

```
PROGRAM :PGCD
:INPUT A
:INPUT B
:LBL 1
:IF B ≠ 0
```

```
:THEN
:A-INT(A/B)*B->R
:B->A
:R->B
:GOTO 1
:END
:DISP A
```

-Pas de version récursive.

3.1.7 Traduction TI83+

-Version itérative

```
PROGRAM :PGCD
:Prompt A,B
:While B ≠ 0
:A-int(A/B)*B->R
:B->A
:R->B
:End
:A
```

-Pas de version récursive.

3.1.8 Traduction TI89 92

-Version itérative

```
:pgcd(a,b)
:Func
:Local r
:While b ≠ 0
:mod(a,b)->r
:b->a
:r->b
:EndWhile
:Return a
:EndFunc
```

-Version récursive

```
:pgcd(a,b)
:Func
:If b ≠ 0 Then
:Return  pgcd(b, mod(a,b))
:Else
:Return a
:EndIf
:EndFunc
```

3.1.9 Traduction SHARP EL9600

-Version itérative :

```
PGCD
Input A
Input B
Label DTQ
If B=0 Goto FTQ
A− int (A/B) ∗ B ⇒ R
B ⇒ A
R ⇒ B
Goto DTQ
Label FTQ
Print ''PGCD
Print A
End
```

-Pas de version récursive.

3.2 Identité de Bézout par l'algorithme d'Euclide

Dans ce paragraphe la fonction Bezout(A,B) renvoie la liste $\{U, V, PGCD(A, B)\}$ où U et V vérifient :
$A \times U + B \times V = PGCD(A, B)$.

3.2.1 Version itérative sans les listes

L'algorithme d'Euclide permet de trouver un couple U et V vérifiant :
$A \times U + B \times V = PGCD(A, B)$
En effet, si on note A_0 etB_0 les valeurs de A et de B du début on a :

$$\begin{aligned} A &= A_0 \times U + B_0 \times V \quad avec\ U = 1\ et\ V = 0 \\ B &= A_0 \times W + B_0 \times X \quad avec\ W = 0\ et\ X = 1 \end{aligned}$$

Puis on fait évoluer A, B, U, V, W, X de façon que les deux relations ci-dessus soient toujours vérifiées.
Si :
$A = B \times Q + R \quad 0 \leq R < B \quad (R = A\ mod\ B\ et\ Q = E(A/B))$
On écrit alors :

$$R = A - B \times Q = A_0 \times (U - W \times Q) + B_0 \times (V - X \times Q) = A_0 \times S + B_0 \times T$$
$$avec\ S = U - W \times Q\ et\ T = V - X \times Q$$

Il reste alors à recommencer avec B dans le rôle de A (`B->A W->U X->V`) et R dans le rôle de B (`R->B S->W T->X`) d'où l'algorithme :

```
fonction Bezout (A,B)
local U,V,W,X,S,T,Q,R
1->U 0->V 0->W 1->X
```

```
tant que B ≠ 0 faire
A mod B->R
E(A/B)->Q
//R=A-B*Q
U-W*Q->S
V-X*Q->T
B->A W->U X->V
R->B S->W T->X
ftantque
r\'esultat {U, V, A}
ffonction
```

3.2.2 Version itérative avec les listes

On peut simplifier l'écriture de l'algorithme ci-dessus en utilisant moins de variables : pour cela on utilise des listes LA LB LR pour mémoriser les triplets {U, V, A} {W, X, B} et {S, T, R}. Ceci est très commode car les calculatrices savent ajouter des listes de même longueur (en ajoutant les éléments de même indice) et savent aussi multiplier une liste par un nombre (en multipliant chacun des éléments de la liste par ce nombre).

```
fonction Bezout (A,B)
local LA LB LR
{1, 0, A}->LA
{0, 1, B}->LB
tant que LB[3] ≠ 0 faire
LA-LB*E(LA[3]/LB[3])->LR
LB->LA
LR->LB
ftantque
r\'esultat LA
ffonction
```

3.2.3 Version récursive sans les listes

Si on utilise des variables globales pour `A B D U V T`, on peut voir la fonction `Bezout` comme calculant à partir de `A B`, des valeurs qu'elle met dans `U V D (AU+BV=D)` grâce à une variable locale `Q`.
On écrit donc une fonction sans paramètre : seule la variable `Q` doit être locale à la foncton alors que les autres variables `A, B ...` peuvent être globales.
`Bezout` fabrique `U, V, D` vérifiant `A*U+B*V=D` à partir de `A` et `B`. Avant l'appel récursif (on préserve `E(A/B)=Q` et on met `A` et `B` à jour (nouvelles valeurs), après l'appel les variables `U, V, D` vérifient `A*U+B*V=D` (avec `A` et `B` les nouvelles valeurs), il suffit alors de revenir aux premières valeurs de `A` et `B` en écrivant : `B*U+(A-B*Q)*V=A*V+B*(U-V*Q)` On écrit alors :

```
fonction Bezout
local Q
```

```
Si B ≠ 0 faire

E(A/B)->Q
A-B*Q->R
B->A
R->B
Bezout
U-V*Q->W
V->U
W->V
sinon
1->U
0->V
A->D
fsi
ffonction
```

3.2.4 Version récursive avec les listes

On peut définir récursivement la fonction Bezout par :
$Bezout(A, 0) = \{1,\ 0,\ A\}$
Si $B \neq 0$ il faut définir $Bezout(A, B)$ en fonction de $Bezout(B, R)$ lorsque $R = A - B \times Q$ *et* $Q = E(A/B)$.
On a :

$$Bezout(B, R) = LT = \{W, X, pgcd(B, R)\}$$
$$avec\ W \times B + X \times R = pgcd(B, R)$$

Donc :

$$\begin{aligned} W \times B + X \times (A - B \times Q) &= pgcd(B, R)\ ou\ encore \\ X \times A + (W - X \times Q) \times B &= pgcd(A, B). \end{aligned}$$

D'où si $B \neq 0$ et si $Bezout(B, R) = LT$ *on a* :
$Bezout(A, B) = \{LT[2],\ LT[1] - LT[2] \times Q,\ LT[3]\}$.

```
fonction Bezout (A,B)
local LT Q R

Si B ≠ 0 faire

E(A/B)->Q
A-B*Q->R
Bezout(B,R)->LT
R\'esultat {LT[2], LT[1]-LT[2]*Q, LT[3]}
sinon R\'esultat {1, 0, A}
fsi
ffonction
```

3.2.5 Traduction Casio

-Version itérative avec les listes

On définit le programme INAB qui permet de rentrer deux nombres A et B :

```
‘‘A’’?->A
‘‘B’’?->B
ClrText
Return
```

Puis on tape le programme BEZOUT :

```
Prog ‘‘INAB’’
{1, 0, A}->List 1
{0, 1, B}->List 2
While List 2[3] ≠ 0
List 1-List 2 * Int (List 1[3]/List 2[3])->List 3
List 2->List 1
List 3->List 2
WhileEnd
‘‘U V PGCD’’
List 1[1] △
List 1[2] △
List 1[3] △
```

-Pas de version récursive

3.2.6 Traduction HP38G HP40G

-Version itérative avec les listes

On utilise ici aussi le programme IN qui permet de rentrer deux entiers A et B :

```
INPUT A;"A";;;1:
INPUT B;"B";;;1:
ERASE:
```

Puis on tape le programme BEZOUT :

```
RUN IN:
DISP 3;"BEZOUT "{A,B}:
{1,0,A} ->L1:
{0,1,B} ->L2:
WHILE L2(3) ≠ 0 REPEAT
L1-L2*FLOOR(L1(3)/L2(3)) ->L3:
L2 ->L1:
L3 ->L2:
END:
DISP 4;"U V PGCD "L1:
FREEZE:
```

- Pour la HP40G, version récursive sans les listes.

On écrit le programme BEZOUR, grâce aux commandes :
PUSH (PUSH(A) pour mettre le contenu de A dans l'historique du CAS)
et POP (pour récupérer les valeurs mises dans l'historique du CAS)
PUSH et POP permettent de simuler la variable locale Q On a remplacé dans la traduction (cf 3.2.3) les variables R et W par la variable temporaire T.

```
PROGRAM BEZOUR
IF B ≠ 0 THEN
PUSH (FLOOR(A/B)):
A MOD B->T:
B->A:
T->B:
RUN BEZOUR:
U-V*POP->T:
V->U:
T->V:
ELSE
 1->U:
 0->V:
 A->D:
END:
```

PUSH (FLOOR(A/B)) a pour effet de mettre les différentes valeurs de FLOOR(A/B) sur une pile, et POP de les récuperer.
T est une variable auxillaire.
BEZOUR prend comme entrée les valeurs des variables globales A et B et remplit les variables globales U et V de façon que :
$A \cdot U + B \cdot V = PGCD(A, B)$.
On écrit ensuite le programme final BEZOURT permettant l'entrée de A et B et la sortie de {U, V, D}.

```
PROGRAM BEZOURT
PROMPT A:
PROMPT B:
RUN BEZOUR:
ERASE:
MSGBOX {U,V,D}:
```

Remarque :
Si on utilise la fonction de calcul symbolique IREMAINDER à la place de MOD et IQUOT(A,B) à la place de FLOOR(A/B) dans les programmes précédents, BEZOUT ou BEZOUR peut alors avoir comme paramètres des entiers de Gauss à condition de remplacer les noms des variables A, B, R... par Z1, Z2, Z3....
Remarque :
Si on utilise la fonction du calcul symbolique REMAINDER à la place de MOD dans les programmes précédents, BEZOUT (ou BEZOUR) peut alors avoir comme paramètres des polynômes à condition de remplacer les noms des variables A, B, R... par E1, E2, E3... et de changer le test d'arrêt.

3.2.7 Traduction HP48 ou HP49G mode RPN

-Version itérative avec les listes

Au début A et B contiennent les deux nombres pour lesquels on cherche l'identité de Bézout, puis A et B désignent les listes LA et LB de l'algorithme.

```
<< {} -> A B R
  << {1 0} 'A' STO+
{0 1} 'B' STO+
WHILE B 3 GET 0 ≠ REPEAT
 A B A 3 GET B 3 GET / FLOOR * - 'R' STO
 B 'A' STO
 R 'B' STO
END
A
   >>
 >>
```

Puis on stocke ce programme dans `BEZOUT ('BEZOUT' STO)`.

-Version récursive avec les listes

```
<< {} -> A B T
  « IF B 0 ≠ THEN
B A B MOD BEZOUR 'T' STO
 T 2 GET DUP A B / FLOOR *
T 1 GET SWAP -
T 3 GET {} + + +
ELSE
 {1 0} A +
END
  >>
 >>
```

Puis on stocke ce programme dans `BEZOUR ('BEZOUR' STO)`.

3.2.8 Traduction HP49G mode Algébrique

-Version itérative avec les listes

Au début A et B contiennent les deux nombres pour lesquels on cherche l'identité de Bézout, puis A et B désignent les listes LA et LB de l'algorithme.

```
≪ → A,B
   ≪ {} → R
      ≪ {1,0,A} STO▷ A;
         {0,1,B} STO▷ B;
         WHILE B[3] ≠ 0 REPEAT
            A − B ∗ FLOOR(A[3]/B[3]) STO▷ R;
            B STO▷ A;
            R STO▷ B
         END
```

```
        A
      ≫
   ≫
≫ STO▷ BEZOUT
```

Puis par exemple `BEZOUT(45,75)` pour l'exécuter.

-Version récursive avec les listes

```
≪ → A,B
   ≪ {} → T
      ≪ IF B ≠ 0 THEN
            BEZOUR(B,A MOD B) STO▷ T;
            {T[2],T[1] − T[2] ∗ FLOOR(A/B),T[3]}
         ELSE
            {1,0,A}
         END
      ≫
   ≫
≫ STO▷ BEZOUR
```

Puis par exemple `BEZOUR(45,75)` pour l'exécuter.

3.2.9 Traduction TI80

-Version itérative avec les listes

Au début A et B contiennent les deux nombres pour lesquels on cherche l'identité de Bézout. LA, LB et LR de l'algorithme sont ici L_1, L_2 et L_3.

```
PROGRAM :BEZOUT
:INPUT A
:INPUT B
:{1,0,A}−>L1
:{0,1,B}−>L2
:LBL 1
:IF L2(3) ≠ 0
:THEN
:L1 − L2 ∗ INT(L1(3)/L2(3))−>L3
:L2−>L1
:L3−>L2
:GOTO 1
:END
:L1
```

-Pas de version récursive.

3.2.10 Traduction TI83+

-Version itérative avec les listes

Au début A et B contiennent les deux nombres pour lesquels on cherche l'identité de Bézout. LA, LB et LR de l'algorithme sont ici L_1, L_2 et L_3.

```
PROGRAM :BEZOUT
:INPUT A
```

```
:INPUT B
:{1,0,A}->L1
:{0,1,B}->L2
:While L2(3) ≠ 0
:L1-L2*int(L1(3)/L2(3))->L3
:L2->L1
:L3->L2
:End
:L1
```

-Pas de version récursive.

3.2.11 Traduction TI89 92

-Version itérative avec les listes

Au début a et b contiennent les deux nombres pour lesquels on cherche l'identité de Bézout, puis a et b désignent les listes LA et LB de l'algorithme.

```
:bezout (a,b)
:Func
:local r
:{1, 0, a}->a
:{0, 1, b}->b
:While b[3] ≠ 0
:a-b*floor(a[3]/b[3])->r
:b->a
:r->b
:EndWhile
:Return a
:EndFunc
```

-Version récursive avec les listes

```
:bezout (a,b)
:local L, q, r
:If b ≠ 0 Then
:floor(a/b)->q
:a-b*q->r
:bezout(b,r)->L
:Return {L[2], L[1]-L[2]*q, L[3]}
:Else
:Return {1, 0, a}
:EndIf
:EndFunc
```

3.2.12 Traduction SHARP EL9600

-Version itérative avec les listes

```
BEZOUT
Input A
```

```
Input B
{1,0,A} ⇒ L1
{0,1,B} ⇒ L2
Label DTQ
If L2(3)=0 Goto FTQ
L1 − int (L1(3)/L2(3)) ∗ L2(3) ⇒ L3
L2 ⇒ L1
L3 ⇒ L2
Goto DTQ
Label FTQ
Print ''BEZOUT
Print L1
End
```

-Pas de version récursive.

3.3 Décomposition en facteurs premiers d'un entier

3.3.1 Les algorithmes et leurs traductions algorithmiques

- Premier algorithme

Soit N un entier.
On teste, pour tous les nombres D de 2 à N, la divisibilité de N par D.
Si D divise N, on cherche alors les diviseurs de N/D etc...N/D joue le rôle de N et on s'arrête quand $N = 1$
On met les diviseurs trouvés dans la liste `FACT`.

```
fonction facprem(N)
local D FACT
2 -> D
{} -> FACT

tant que N ≠ 1 faire

  si N mod D = 0 alors
    FACT + D -> FACT
    N/D -> N
   sinon
    D+1 -> D
 fsi
ftantque
r\'esultat FACT
ffonction
```

- Première amélioration

On ne teste que les diviseurs D entre 2 et $E(\sqrt{N})$.
En effet si $N = D1 * D2$ alors on a :
soit $D1 \leq E(\sqrt{N})$, soit $D2 \leq E(\sqrt{N})$ car sinon on aurait :
$D1 * D2 \geq (E(\sqrt{N}) + 1)^2 > N$.

```
fonction facprem(N)
```

```
local D FACT
2 -> D
{} -> FACT
tant que D*D ≤ N faire
  si N mod D = 0 alors
    FACT + D -> FACT
    N/D-> N
   sinon
    D+1 -> D
 fsi
ftantque
FACT + N -> FACT
r\'esultat FACT
ffonction
```

- Deuxième amélioration On cherche si 2 divise N, puis on teste les diviseurs impairs D entre 3 et $E(\sqrt{N})$.

Dans la liste `FACT` on fait suivre chaque diviseur par son exposant : decomp(12)={2,2,3,1}.

```
fonction  facprem(N)
local K D FACT
{}->FACT
0 -> K
tant que N mod 2 = 0 faire
    K+1 -> K
    N/2 -> N
ftantque
si K ≠0 alors
    FACT + {2 K} -> FACT
fsi
3 ->D
tant que D*D ≤ N faire
  0 -> K
  tant que N mod D = 0 faire
    K+1 -> K
    N/D -> N
  ftantque
  si K ≠0 alors
    FACT + {D K} -> FACT
  fsi
  D+2 -> D
ftantque
si N ≠ 1 alors
FACT + {N 1} -> FACT
fsi
r\'esultat FACT
ffonction
```

3.3.2 Traduction Casio

Voici le programme FACTPREM :

```
‘‘N’’?->N
100->Dim List 1
0->K
0->I
While N-Int(N/2)*2 =0
K+1->K
N/2->N
WhileEnd
If K ≠0
Then 2->List 1[1]
K->List 1[2]
2->I
IfEnd
3->D
While D*D ≤N
0->K
While N-Int(N/D)*D=0
K+1->K
N/D->N
WhileEnd
If K ≠0
Then I+2->I
D->List 1[I-1]
K->List 1[I]
IfEnd
D+2->D
If N ≠1
Then I+2->I
N->List 1[I-1]
1->List 1[I]
IfEnd
I △
List 1 △
```

3.3.3 Traduction HP38G HP40G

On traduit le dernier algorithme.
La `HP38G` ne connait pas la liste {}, donc pour initialiser L1 avec la liste vide on écrit : `CLEAR L1`.
La `HP40G` ne connait pas la liste {}, donc pour initialiser L1 avec la liste vide on écrit :`SYSEVAL 259589`.
Voici le programme FACTPREM :

```
INPUT N;"N";;;1:
ERASE:
0 ->K:
CLEAR L1:  @ou SYSEVAL 259589:
WHILE N MOD 2 == 0 REPEAT
1+K -> K:
N/2 -> N:
END:
IF K ≠ 0 THEN
{2,K} ->L1:
END:
3 ->D:
WHILE D*D ≤ N REPEAT
0 -> K:
WHILE N MOD D == 0 REPEAT
K+1 -> K:
N/D -> N:
END:
IF K ≠ 0 THEN
CONCAT (L1,{D,K}) -> L1:
END:
2+D -> D:
END:
IF N ≠ 1 THEN
CONCAT (L1, {N,1}) -> L1:
END:
DISP 3; "FACT" L1:
FREEZE:
```

3.3.4 Traduction HP48 ou HP49G mode RPN

On traduit le dernier algorithme en utilisant une liste `FACT` contenant les facteurs premiers avec leur multiplicité.
Voici le programme FACTPREM :

```
<< 0 3 {} -> N K D FACT
  << WHILE N 2 MOD 0 ==
        REPEAT
        1 'K' STO+
        'N' 2 STO/
    END
    IF K 0 ≠ THEN
      {2 K} 'FACT' STO
   END
    WHILE N D D * ≥
```

```
      REPEAT
      0 'K' STO
      WHILE N D MOD 0 ==
        REPEAT
        1 'K' STO+
        'N' D STO/
      END
      IF K 0 ≠ THEN
       {D K} 'FACT' STO+
      END
      2 'D' STO+
    END
    IF N 1 ≠ THEN
      {N 1} 'FACT' STO+
    END
  >>
>>
```

Puis on stocke ce programme dans `FACTPREM ('FACTPREM' STO)`.

3.3.5 Traduction HP49G mode Algébrique

On traduit le dernier algorithme en utilisant une liste `FACT` contenant les facteurs premiers avec leur multiplicité.
Voici le programme FACTPREM :

```
≪ → N
  ≪ 0 → K
    ≪ WHILE N MOD 2 == 0 REPEAT
        K + 1 STO▷ K;
        N/2 STO▷ N
      END;
      {} → FACTO
        ≪ IF K ≠ 0 THEN
            FACTO + {2,K} STO▷ FACTO
          END;
          3 → D
            ≪ WHILE D * D ≤ N REPEAT
                0 STO▷ K;
                WHILE N MOD D == 0 REPEAT
                  K + 1 STO▷ K;
                  N/D STO▷ N;
                END;
                IF K ≠ 0 THEN
                  FACTO + {D,K} STO▷ FACTO
                END;
                D + 2 STO▷ D
              END;
              IF N ≠ 1 THEN
```

```
                    FACTO + {N, 1} STO▷ FACTO
                END;
                FACTO;
            ≫
        ≫
    ≫
  ≫
≫ STO▷ FACTPREM
```

Puis par exemple `FACTPREM(45)` pour l'exécuter.

3.3.6 Traduction TI80

-Version itérative avec les listes
La liste `L1` joue le rôle de la liste `FACT`.

```
PROGRAM :FACTPRE
:INPUT N
:CLRLIST L1
:0->K
:1->I
:LBL 1
:IF FPART(N/2)=0
:THEN
:K+1->K
:N/2->N
:GOTO 1
:END
: IF K ≠ 0
:THEN
:2->L1(1)
:K->L1(2)
:I+2->I
:END
:3->D
:LBL 2
: IF D * D ≤ N
:THEN
:0->K
:LBL 3
:IF FPART(N/D)=0
:K+1->K
:N/D->N
:GOTO 3
:END
: If K ≠ 0
:THEN
:D->L1(I)
:K->L1(I+1)
```

```
:I->I+2
:END
:D+2->D
:GOTO 2
:END
: IF N ≠ 1
:THEN
:N->L1(I)
:I+1->I
:1->L1(I)
:END
:DISP L1
```

3.3.7 Traduction TI83+

-Version itérative avec les listes
La liste L_1 joue le rôle de la liste FACT.
On initialise la liste L_1 à $\{0\}$ car la liste vide n'est pas admise...
Puis on rajoute au fur et à mesure les diviseurs avec leur multiplicité en début de liste de façon à laisser le zéro en fin de liste...puis en fin de programme, on supprime ce zéro génant en enlevant 1 à la dimension de L_1 : je n'ai pas trouvé d'autres moyens si on veut utiliser augment (car $ClrListL_1$ n'initialise pas L_1 à vide !)
sinon il faut écrire $100 -> dim(L1)$ puis gérer l'indice I qui compte les éléments pertinents de L1 (cf 3.3.8 le programme facprem1 de la TI89)

```
PROGRAM :FACTPREM
:Input N
:0->K
: 0− > L1
:While fPart(N/2)=0
:K+1->K
:N/2->N
:End
: If K ≠ 0
:Then
: augment({2, K}, L1)− > L1
:End
:3->D
: While D ∗ D ≤ N
:0->K
:While fPart(N/D)=0
:K+1->K
:N/D->N
:End
: If K ≠ 0
:Then
: augment({D, K}, L1)− > L1
:End
```

```
:D+2->D
:End
: If N ≠ 1
:Then
: augment({N, 1}, L1)− > L1
:End
: dim(L1) − 1− > dim(L1)
: DispL1
```

3.3.8 Traduction TI89 92

On traduit le dernier algorithme en utilisant une liste `fact` de 100 éléments : on ne peut donc factoriser que des nombres qui n'ont pas plus de 50 diviseurs premiers.(Exercice : Quel est le plus petit nombre ayant 50 diviseurs premiers ?)

On utilise un indice `i` pour compter les éléments pertinents de `fact`.

```
:factprem1(n)
:Func
:Local fact,k,d,i
:newList(100)->fact
:0->k
:0->i
:While mod(n,2)=0
:k+1->k
:n/2->n
:EndWhile
:If k ≠0 Then
:2->fact[1]
:k->fact[2]
:2->i
:EndIf
:3->d
:While d*d ≤n
:0->k
:While mod(n,d)=0
:k+1->k
:n/d->n
:EndWhile
:If k ≠0 Then
:i+2->i
:d->fact[i-1]
:k->fact[i]
:EndIf
:d+2->d
:EndWhile
:If n ≠1 Then
```

```
:i+2->i
:n->fact[i-1]
:1->fact[i]
:EndIf
:Return left(fact,i)
:EndFunc
```

ou bien
on construit la liste `fact` au fur et à mesure et on obtient le programme :

```
:factprem2(n)
:Func
:Local fact,k,d,i
:0->k
:{}->fact
:While mod(n,2)=0
:k+1->k
:n/2->n
:EndWhile
:If k ≠0 Then
:augment({2,k},fact)->fact
:EndIf
:3->d
:While d*d ≤n
:0->k
:While mod(n,d)=0
:k+1->k
:n/d->n
:EndWhile
:If k ≠0 Then
:augment({2,k},fact)->fact
:EndIf
:d+2->d
:EndWhile
:If n ≠1 Then
:augment({2,k},fact)->fact
:EndIf
:Return fact
:EndFunc
```

3.3.9 Traduction SHARP EL9600

-Version itérative avec les listes
On initialise la liste L1 à {0} car la liste vide n'est pas admise...
Puis on rajoute au fur et à mesure les diviseurs avec leur multiplicité en début de liste de façon à laisser le zéro en fin de liste...puis en fin de programme, on supprime ce zéro gênant en enlevant 1 à la dimension de L1 (je n'ai pas trouvé d'autres moyens car si on veut utiliser `augment` (car `ClrList`

L1 n'initialise pas L1 à vide ! ! !)
sinon il faut écrire 100 ⇒ dim(L1) puis gérer l'indice I qui compte les éléments pertinents de L1 (cf 3.3.8 le programme facprem1 de la TI89)

```
FACTPREM
Input N
{0} ⇒ L1
0 ⇒ K
Label DTQ1
If fpart(N/2) ≠ 0 Goto FTQ1
K + 1 ⇒ K
N/2 ⇒ N
Goto DTQ1
Label FTQ1
If K = 0 Goto FS1
augment({2, K}, L1) ⇒ L1
Label FS1
3 ⇒ D
Label DTQ2
If D*D>N Goto FTQ2
0 ⇒ K
Label DTQ3
If fpart(N/D) ≠ 0 Goto FTQ3
K + 1 ⇒ K
N/D ⇒ N
Goto DTQ3
Label FTQ3
If K = 0 Goto FS2
augment({D, K}, L1) ⇒ L1
Label FS2
D + 2 ⇒ D
Goto DTQ2
Label FTQ2
If N = 1 Goto FS3
augment({N, 1}, L1) ⇒ L1
Label FS3
dim(L1) − 1 ⇒ dim(L1)
Print L1
End
```

3.4 Calcul de A^P *mod* N

3.4.1 Traduction Algorithmique

-Premier algorithme
On utilise deux variables locales PUIS et I.
On fait un programme itératif de façon qu'à chaque étape PUIS représente A^I *mod* N

```
fonction puimod1 (A, P, N)
```

```
local PUIS, I
1->PUIS
pour I de 1 a P faire
  A*PUIS mod N ->PUIS
fpour
resultat PUIS
ffonction
```

-Deuxième algorithme
On utilise une seule variable locale PUI mais on fait varier P de façon qu'à chaque étape de l'itération on ait :
$resultat = PUI * A^P \ mod \ N$

```
fonction puimod2 (A, P, N)
local PUI
1->PUI
tant que  P>0  faire
  A*PUI mod N ->PUI
  P-1->P
ftantque
resultat PUI
ffonction
```

-Troisième algorithme
On peut aisément modifier ce programme en remarquant que :
$A^{2*P} = (A * A)^P$.
Donc quand P est pair on a la relation :
$PUI * A^P = PUI * (A * A)^{P/2} \ mod \ N$
et quand P est impair on a la relation :
$PUI * A^P = PUI * A * A^{P-1} \ mod \ N$.
On obtient alors un calcul rapide de $A^P \ mod \ N$.

```
fonction puimod3 (A, P, N)
local PUI
1->PUI
tant que  P>0  faire
  si P mod 2 =0 alors
    P/2->P
    A*A mod N->A
  sinon
    A*PUI mod N ->PUI
    P-1->P
  fsi
ftantque
resultat PUI
ffonction
```

On peut remarquer que si P est impair, P-1 est pair.
On peut donc écrire :

```
fonction puimod4 (A, P, N)
local PUI
1->PUI
```

```
tant que  P>0  faire
  si P mod 2 =1 alors
    A*PUI mod N ->PUI
    P-1->P
  fsi
P/2->P
A*A mod N->A
ftantque
resultat PUI
ffonction
```

-Programme récursif

On peut définir la puissance par les relations de récurrence : $A^0 = 1$
$A^{P+1}\ mod\ N = (A^P\ mod\ N) * A\ mod\ N$

```
fonction puimod5(A, P, N)
si P>0 alors
resultat puimod5(A, P-1, N)*A mod N
sinon
resultat 1
fsi
ffonction
```

-Programme récursif rapide

```
fonction puimod6(A, P, N)
si P>0 alors
  si P mod 2 =0 alors
    resultat puimod6((A*A mod N), P/2, N)
  sinon
    resultat puimod6(A, P-1, N)*A mod N
  fsi
sinon
resultat 1
fsi
ffonction
```

3.4.2 Traduction HP48 ou HP49G mode RPN

L'utilisateur doit mettre sur la pile : A, P, N pour obtenir $A^P mod\ N$.
Voici la traduction de l'algorithme rapide itératif :

```
« 1-> A P N PUI
« WHILE P 0 > REPEAT
    IF P 2 MOD 1 == THEN
    A PUI * N MOD 'PUI' STO
    'P' STO-
    END
  P 2 / 'P' STO
  A A * N MOD 'A' STO
  END
  PUI
```

```
»
»
```

Puis on stocke ce programme dans `PUIMOD ('PUIMOD' STO)`.

3.4.3 Traduction HP49G mode Algébrique

```
≪ → A P N
  ≪ 1 → PUI
    ≪ WHILE P > 0 REPEAT
        IF P MOD 2 == 1 THEN
          A * PUI MOD N STO▷ PUI
          P − 1 STO▷ P;
        END;
        P/2 STO▷ P;
        A * A MOD N STO▷ A;
      END;
      PUI
    ≫
  ≫
≫ STO▷ PUIMOD
```

Puis par exemple `PUIMOD(45,32,13)` pour l'exécuter.

3.4.4 Traduction TI 89-92

Voici la traduction de l'algorithme rapide itératif :

```
:puimod(a,p,n)
:Func
:Local pui
:1->pui
:While p>0
:If p mod 2 = 1 Then
:mod(a*pui,n)->pui
:p-1->p
:EndIf
:p/2->p
:mod(a*a,n)->a
:EndWhile
:Return pui
:EndFunc
```

3.4.5 Traduction Casio, HP38G, HP40G, TI80, TI83+ ou SHARP-EL9600

Les calculatrices `Casio, HP38G, HP40G, TI80, TI83+` ou `SHARP-EL9600` ne permettent pas d'écrire des fonctions et n'ont pas la notion de paramètres : on est donc obligé d'insérer le programme de ce calcul quand on en a besoin avec les variables pertinentes du programme principal.
Pour la traduction on se reportera page 70 pour la `Casio`, page 71 pour la

HP 38, page 73 pour la TI80, page 74 pour la TI83+ et à la page 75 pour la SHARP-EL9600 où ce programme est utilisé.

3.5 La fonction "estpremier"

3.5.1 Traduction Algorithmique

- Premier algorithme

On va écrire un fonction booléenne de paramètre N, qui sera égale à VRAI quand N est premier et à FAUX sinon.
Pour cela, on cherche si N posséde un diviseur $\neq 1$ et $\leq$ à $E(\sqrt{N})$ (partie entière de racine de N).
On traite le cas N=1 à part !
On utilise une variable booléenne PREM qui est au départ à VRAI, et qui passe à FAUX dès que l'on rencontre un diviseur de N.

```
Fonction estpremier(N)
local PREM, I, J
E(√N) ->J
Si N = 1 alors
  FAUX->PREM
  sinon
  VRAI->PREM
fsi
2->I
tant que PREM et I ≤J faire
  si N mod I = 0 alors
     FAUX->PREM
     sinon
     I+1->I
  fsi
ftantque
r\'esultat PREM
ffonction
```

-Première amélioration

On peut remarquer que l'on peut tester si N est pair, et sinon regarder si N posséde un diviseur impair.

```
Fonction estpremier(N)
local PREM, I, J
E(√N) ->J
Si (N = 1) ou (N mod 2 = 0) et N≠2 alors
  FAUX->PREM
  sinon
  VRAI->PREM
fsi
3->I
```

```
tant que PREM et I ≤J faire
  si N mod I = 0 alors
     FAUX->PREM
     sinon
     I+2->I
  fsi
ftantque
r\'esultat PREM
ffonction
```

- Deuxième amélioration
On regarde si N est divisible par 2 ou par 3, sinon on regarde si N posséde un diviseur de la forme $6 \times k - 1$ ou $6 \times k + 1$.

```
Fonction estpremier(N)
local PREM, I, J
E(√N) ->J
Si (N = 1) ou (N mod 2 = 0) ou ( N mod 3 = 0) alors
  FAUX->PREM
  sinon
  VRAI->PREM
fsi
si N=2 ou N=3 alors
VRAI->PREM
fsi
5->I
tant que PREM et I ≤J faire
  si (N mod I = 0) ou (N mod I+2 =0) alors
     FAUX->PREM
     sinon
     I+6->I
  fsi
ftantque
r\'esultat PREM
ffonction
```

3.5.2 Traduction Casio

Voici le programme ESTPREM :

```
‘‘N’’?->N
If Frac(N/2)*2=0 Or N=1 Or Frac(N/3)*3=0
Then 0->P
Else 1->P
IfEnd
If N=2 Or N=3
Then 1->P
IfEnd
5->I
```

```
Int(√N)->J
While I≤J And P
If Frac(N/I)*I=0 Or Frac(N/(I+2)*(I+2)=0
Then 0->P
Else I+6->I
IfEnd
WhileEnd
P △
```

3.5.3 Traduction HP38G HP40G

On traduit le dernier algorithme : le résultat est 0 (FAUX) ou 1 (VRAI). Voici le programme ESTPREM :

```
INPUT N;"N";;;1:
IF N MOD 2== 0 OR N MOD 3==0 OR N==1 THEN
0 ->P:
ELSE
1->P:
END:
IF N==2 OR N==3 THEN
1->P:
END:
5->I:
FLOOR(√N)->J :
WHILE I ≤ J AND P REPEAT
IF N MOD I==0 OR N MOD (I+2)==0 THEN
0 ->P:
ELSE
I+6 ->I:
END:
END:
ERASE:
DISP 5;P:
FREEZE:
```

3.5.4 Traduction HP48 ou HP49G mode RPN

On traduit le dernier algorithme : le résultat est 0 (FAUX) ou 1 (VRAI). Voici le programme ESTPREM :

```
« DUP √ FLOOR 0 5 -> N J PREM I
    « IF N 1 == N 2 MOD 0 == OR N 3 MOD 0 == OR THEN
   0 'PREM' STO
  ELSE
    1 'PREM' STO
  END
  IF N 2 == N 3 == OR THEN
```

```
    1 'PREM' STO
  END


WHILE PREM I J ≤ AND REPEAT

  IF N I MOD 0 == N I 2 + MOD 0 == OR THEN
     0 'PREM' STO
  ELSE
     I 6 + 'I' STO
  END
END
PREM

  »
»
```

Puis on stocke ce programme dans `ESTPREM ('ESTPREM' STO)`.

3.5.5 Traduction HP49G mode Algébrique

Voici le programme ESTPREM :

```
≪ → N
  ≪ 0 → P
    ≪ IF N MOD 2 == 0 OR N MOD 3 == 0 OR N == 1THEN
         0 STO▷ P
       ELSE;
         1 STO▷ P;
       END;
       IF N == 2 OR N == 3 THEN
         1 STO▷ P;
       END;
       FLOOR(√N) → J
         ≪ 5 → I
           ≪ WHILE I ≤ J AND P REPEAT
                IF N MOD I == 0 OR N MOD (I + 2) == 0 THEN
                  0 STO▷ P;
                ELSE;
                  I + 6 STO▷ I;
                END;
              END;
              P
           ≫
         ≫
    ≫
  ≫
≫ STO▷ ESTPREM
```

Puis par exemple `ESTPREM(45789)` pour l'exécuter.

3.5.6 Traduction TI80

On traduit le dernier algorithme : le résultat est 0 (FAUX) ou 1 (VRAI).

```
PROGRAM :ESTPREM
:INPUT N
:0->K
: INT(√N)-> J
:1->P
:IF FPART(N/2)=0
:THEN
: IF N ≠ 2
:THEN
:0->P
:END
:ELSE
:IF FPART(N/3)=0
:THEN
: IF N ≠ 3
:THEN
:0->P
:END
:ELSE
:IF N=1
:THEN
:0->P
:END
:END
:END
:5->I
:LBL 1
:IF P
:THEN
: IF I ≤ J
:THEN
:FOR (S,1,2)
: IF FPART(N/I) = 0
:THEN
:0->P
:ELSE
:I+2->I
:END
:END
:I+2->I
:GOTO 1
:END
:END
:DISP P
```

3.5.7 Traduction TI83+

On traduit le dernier algorithme : le résultat est 0 (FAUX) ou 1 (VRAI).

```
PROGRAM :ESTPREM
:Input N
:0->K
: int(√N)-> J
: If N = 1 or fPart(N/2) = 0 or fPart(N/3) = 0
:Then
:0->P
:Else
:1->P
:End
: If N = 2 or N = 3
:Then
:1->P
:End
:5->I
: While P and I ≤ J
: If fPart(N/(I + 2)) = 0 or fPart(N/I) = 0
:Then
:0->P
:Else
:I+6->I
:End
:End
:Disp P
```

3.5.8 Traduction TI89-92

On traduit le dernier algorithme : le résultat est false (FAUX) ou true (VRAI).

```
:estprem(n)
:Func
:Local i, j, prem
:If mod(n,2)=0 or mod(n,3)=0 or n=1 Then
:false->prem
:Else
:true->prem
:EndIf
:If n=2 or n=3 Then
:true->prem
:EndIf
:5->i
:Floor(√n)->j
:While i ≤j and prem
:If mod(n,i)=0 or mod(n,i+2)=0 Then
:false->prem
```

```
:Else
:i+6->i
:EndIf
:EndWhile
:Return prem
:EndFunc
```

3.5.9 Traduction SHARP EL9600

```
ESTPREM
Input N
int (√N) ⇒ J
If (N ≠ 1) and (fpart(N/2) ≠ 0) and (fpart(N/3) ≠ 0) Goto SINON1
0 ⇒ P
Goto FSI1
Label SINON1
1 ⇒ P
Label FSI1
If (N ≠ 2) and (N ≠ 3) Goto FSI2
1 ⇒ P
Label FSI2
5 ⇒ I
Label DTQ
If (P=0) or (I>J) Goto FTQ
If (fpart(N/I) ≠ 0) and (fpart(N/(I + 2)) ≠ 0) Goto SINON3
0 ⇒ P
Goto FSI3
Label SINON3
I + 6 ⇒ I
Label FSI3
Goto DTQ
Label FTQ
Print P
End
```

3.6 Méthode probabiliste de Mr Rabin

Si N est premier alors tous les nombres K strictement inférieurs à N sont premiers avec N, donc d'après le petit théorème de Fermat on a :
$K^{N-1} = 1 \ mod \ N$
Si N n'est pas premier, les entiers K vérifiant :
$K^{N-1} = 1 \ mod \ N$
sont très peu nombreux.
Plus précisement on peut montrer que si $N > 4$, la probabilité d'obtenir un tel nombre est inférieure à 0.25.
La méthode probabiliste de Rabin consiste à tirer au hasard un nombre K ($1 < K < N$) et de calculer :
$K^{N-1} \ mod \ N$

Si $K^{N-1} = 1 \ mod \ N$ on refait un autre tirage et, si $K^{N-1} \neq 1 \ mod \ N$ on est sûr que N n'est pas premier.
Si on obtient $K^{N-1} = 1 \ mod \ N$ pour 20 tirages de K on peut conclure que N est premier avec une probabilité d'erreur très faible inférieure à 0.25^{20} soit de l'ordre de 10^{-12} (on dit alors que N est pseudo-premier).
Bien sûr cette méthode est employée pour savoir si de grands nombres sont pseudo-premiers.

3.6.1 Traduction Algorithmique

On suppose que :
`Hasard(N)` donne un nombre entier au hasard entre 0 et $N-1$.
Le calcul de :
$K^{N-1} \ mod \ N$
se fait grâce à l'algorithme de la puissance rapide (cf page 59).
On notera :
`puismod(K, P, N)` la fonction qui calcule $K^P \ mod \ N$

```
Fonction estprem(N)
local K, I, P
1->I
1->P
Tant que P = 1 et I < 20 faire
hasard(N-2)+2->K
puismod(K, N-1, N)->P
I+1->I
ftantque
Si P =1 alors
resultat VRAI
sinon
resultat FAUX
fsi
ffonction
```

3.6.2 Traduction Casio

Voici le programme RABIN :

```
‘‘N’’?->N
1->P
1->I
While (I<20) And (P=1)
Int(Rand#*(N-2))+2->K
N-1->M
1->P
While 0<M
If M-Int(M/2)*2=1
Then K*P-Int(K*P/N)*N->P
M-1->M
IfEnd
```

```
M/2->M
K*K-Int(K*K/N)*N->K
WhileEnd
I+1->I
WhileEnd
If P=1
Then ''PREMIER'' :N △
Else ''NONPREMIER'' :N △
IfEnd
```

3.6.3 Traduction HP38G HP40G

Voici le programme RABIN :

```
INPUT N;"N";;;1:
RANDSEED TIME:
1->I:
1->P:
WHILE I < 20 AND P==1 REPEAT
  FLOOR( RANDOM * (N-2))+2->K:
  N-1->M:

  @ Calcul de K puissance M mod N dans P.
  1->P:
  WHILE  0 < M REPEAT
    IF M MOD 2 == 0 THEN
      M / 2 -> M :
      (K * K)  MOD N ->K :
    ELSE
      K*P MOD N -> P:
      M - 1 -> M:
    END:
  END:
  @ P contient K puissance M mod N et M=N-1.
  I+1 ->I:
END:
ERASE:
IF P==1 THEN
  DISP 3;"PREMIER " N:
ELSE
  DISP 3;"NON PREMIER " N:
END:
FREEZE:
```

3.6.4 Traduction HP48 ou HP49G mode RPN

On suppose que l'on a écrit la fonction `PUIMOD` qui prend sur la pile trois arguments `A, K, N` et qui renvoie $\texttt{A}^{\texttt{K}}$ `mod N`.

```
<< 1 0 1 ->  N I K P
  << 0 RDZ
WHILE P 1 == 20 I > AND REPEAT
 1 'I' STO+
RAND N 2 - * FLOOR 2 + 'K' STO
K N 1 - N PUIMOD  'P' STO
END
IF P 1 ==  THEN 1 ELSE 0
END
 >>
>>
```

Puis on stocke ce programme dans `RABIN ('RABIN' STO)`.

3.6.5 Traduction HP49G mode Algébrique

On suppose que l'on a écrit la fonction `PUIMOD` qui a trois arguments `A, K, N` et qui renvoie $\mathtt{A}^{\mathtt{K}} \bmod \mathtt{N}$.

```
≪ → N
  ≪ 1 → I
    ≪ 0 → K
      ≪ 1 → P
        ≪ RDZ(0);
          WHILE P == 0 AND I < 20 REPEAT
            1 + I STO▷ I;
            FLOOR((N − 2) ∗ RAND) + 2 STO▷ K;
            PUIMOD(K, N − 1, N) STO▷ P;
          END;
          IF P == 1 THEN
            1
          ELSE
            0
          END;
          P
        ≫
      ≫
    ≫
  ≫
≫ STO▷ RABIN
```

Puis par exemple `RABIN(45313)` pour l'exécuter.

Remarque :

On peut aussi utiliser la commande du calcul formel `POWMOD` et on écrit alors :

En mode `RPN` :

```
N MODSTO
K N 1 - POWMOD 'P' STO
```

à la place de :

```
K N 1 - N PUIMOD 'P' STO
```

En mode `Algèbrique` :

```
MODSTO(N) ;
```

POWMOD(K,N-1) STO▷ P
à la place de :
PUIMOD(K,N-1,N) STO▷ P

3.6.6 Traduction TI80

// Calcul de K^M mod N dans P.

```
PROGRAM :PUIMOD
:1->P
:LBL 1
:IF 0 < M
:THEN
:IF FPART(M/2)=0
:THEN
:M/2->M
:K*K-N*INT(K*K/N)->K
:ELSE
:K*P-N*INT(K*P/N)->P
:M-1->M
:END
GOTO 1
:END
:RETURN
```

// P contient K^P mod N et M=N-1.

```
PROGRAM :RABIN
:INPUT N
:1->I
:1->P
:LBL 1
:IF P=1
:THEN
:IF I < 20
:THEN
:RANDINT(2,N-1)->K
:N-1->M
:PRGM_PUIMOD
:I+1->I
:GOTO 1
:END
:END
:IF P=1
:THEN
:DISP ''PREMIER''
:ELSE
:DISP ''NONPREMIER''
:END
```

3.6.7 Traduction TI83+

@ Calcul de K^M mod N dans P.

```
PROGRAM :PUIMOD
:1->P
:While 0 < M
:If fPart(M/2)=0
:Then
:M/2->M
:K*K-N*int(K*K/N)->K
:Else
:K*P-N*int(K*P/N)->P
:M-1->M
:End
:End
:Return
```

@ P contient K^P mod N et M=N-1.

```
PROGRAM :RABIN
:Input N
:1->I
:1->P
:While P=1 and I < 20
:randInt(2,N-1)->K
:N-1->M
:prgmPUIMOD
:I+1->I
:End
:If P=1
:Then
:Disp ''PREMIER''
:Else
:Disp ''NONPREMIER''
:End
```

3.6.8 Traduction TI89 92

```
:rabin(n)
:Func
:Local k,i,p
:1->p
:1->i
:RandSeed 1147
:While p=1 and i<20
:rand(n-1)+1->k
:puimod(k,n-1,n)->p
:i+1->i
:EndWhile
:If p=1 Then
:Return true
```

```
:Else
:Return false
:EndIf
:EndFunc
```

3.6.9 Traduction SHARP EL9600

```
RABIN
Input N
1 ⇒ I
1 ⇒ P
Label DTQ1
If (I = 20) or (P ≠ 1) Goto FTQ1
int(random ∗ (N − 2)) + 2 ⇒ K
N − 1 ⇒ M
Gosub PUIMOD
I + 1 ⇒ I
Goto DTQ1
Label FTQ1
If P ≠ 1 Goto SINON2
Print ''premier
Goto FSI2
Label SINON2
Print ''nonpremier
Label FSI2
End
Label PUIMOD
Rem Calcul de K puissance M mod N dans P.
1 ⇒ P
Label DTQ2
If M=0 Goto FTQ2
If fpart(M/2) ≠ 0 Goto SINON1
M/2 ⇒ M
K ∗ K − int(K ∗ K/N) ∗ N ⇒ K
Goto FSI1
Label SINON1
K ∗ P − int(K ∗ P/N) ∗ N ⇒ P
M − 1 ⇒ M
Label FSI1
Goto DTQ2
Label FTQ2
Rem P contient K puissance M mod N et M=N-1.
Return
```

Chapitre 4

Les programmes selon les modèles

4.1 Les programmes selon la Casio

4.1.1 Les Entrées

`"A="?->A :`

4.1.2 Les Sorties

`"A=" :A` △

4.1.3 La séquence d'instructions ou action

A la fin d'une instruction, il faut mettre un séparateur qui peut être `le retour à la ligne` ou `:` ou △

4.1.4 L'instruction d'affectation

La flèche est obtenue à l'aide de la touche →.

4.1.5 Les instructions conditionnelles

`If` *condition* `:Then` *action* `: IfEnd :`
`If` *condition* `:Then` *action1* `: Else :` *action2* `: IfEnd :`

4.1.6 Les instructions "Pour"

`For A->I To B :` *action* `: Next`
`For A->I To B Step P :` *action* `: Next`

4.1.7 L'instruction "Tant que"

`While` *condition* `:` *action* `: WhileEnd :`

4.1.8 L'instruction "Repeter"

```
Do : action : LpWhile  condition
```

4.1.9 Un exemple : le crible d'Eratosthène

```
‘‘N’’?->N
N->Dim List 1
N->Dim List 2
Seq(I,I,1,N,1)->List 1
0->List 1[1]
2->P
While P*P≤N
For P->I To Int (N/P)
0->List 1[I*P]
Next
P+1->P
While P*P≤N And List 1[P]=0
P+1->P
WhileEnd
WhileEnd
0->P
For 2->I To N
If List 1[I]≠0
Then P+1->P
I->List 2[P]
IfEnd
Next
P△
List 2△
```

4.1.10 Calcul du PGCD par l'algorithme d'Euclide

On définit le programme INAB qui permet de rentrer deux nombres A et B :

```
‘‘A’’?->A
‘‘B’’?->B
ClrText
Return
```

Puis on tape le programme PGCD :

```
Prog ‘‘INAB’’
While B ≠ 0
  A-B*Intg(A/B)->R
  B->A
  R->B
WhileEnd
A△
```

4.1.11 Identité de Bézout par l'algorithme d'Euclide

```
Prog ‘‘INAB’’
{1, 0, A}->List 1
{0, 1, B}->List 2

While List 2[3] ≠ 0

List 1-List 2 * Int (List 1[3]/List 2[3])->List 3
List 2->List 1
List 3->List 2
WhileEnd
‘‘U V PGCD’’

List 1[1] △
List 1[2] △
List 1[3] △
```

4.1.12 Décomposition en facteurs premiers d'un entier N

```
‘‘N’’?->N
100->Dim List 1
0->K
0->I
While N-Int(N/2)*2 =0
K+1->K
N/2->N
WhileEnd

If K ≠0

Then 2->List 1[1]
K->List 1[2]
2->I
IfEnd
3->D

While D*D ≤N

0->K
While N-Int(N/D)*D=0
K+1->K
N/D->N
WhileEnd

If K ≠0

Then I+2->I
D->List 1[I-1]
K->List 1[I]
IfEnd
D+2->D

If N ≠1
```

```
Then I+2->I
N->List 1[I-1]
1->List 1[I]
IfEnd
I △
List 1 △
```

4.1.13 Calcul de $A^P \ mod \ N$

Voir le sous-programme du programme de la méthode probabiliste de Mr Rabin.

4.1.14 La fonction "estpremier"

```
‘‘N’’?->N
If Frac(N/2)*2=0 Or N=1 Or Frac(N/3)*3=0
Then 0->P
Else 1->P
IfEnd
If N=2 Or N=3
Then 1->P
IfEnd
5->I
Int(√N)->J
While I≤J And P
If Frac(N/I)*I=0 Or Frac(N/(I+2)*(I+2)=0
Then 0->P
Else I+6->I
IfEnd
WhileEnd
P △
```

4.1.15 Méthode probabiliste de Mr Rabin

```
‘‘N’’?->N
1->P
1->I
While (I<20) And (P=1)
Int(Rand#*(N-2))+2->K
N-1->M
1->P
While 0<M
If M-Int(M/2)*2=1
Then K*P-Int(K*P/N)*N->P
M-1->M
IfEnd
M/2->M
```

```
K*K-Int(K*K/N)*N->K
WhileEnd
I+1->I
WhileEnd
If P=1
Then ''PREMIER'' :N △
Else ''NONPREMIER'' :N △
IfEnd
```

4.2 Les programmes selon la HP38G HP40G

4.2.1 Les Entrées

`INPUT A;"TITRE";"A=";;0 :`
pour la `HP 40G` on a aussi :
`PROMPT A :`

4.2.2 Les Sorties

`DISP 3;"A="A :` 3 représente le numéro de la ligne où `A` sera affiché
ou
`MSGBOX "A="A :`

4.2.3 La séquence d'instructions ou action

: indique la fin d'une instruction.

4.2.4 L'instruction d'affectation

La flèche est obtenue à l'aide de la touche `STO` du bandeau.

4.2.5 Les instructions conditionnelles

`IF` *condition* `THEN` *action* `: END :`
`IF` *condition* `THEN` *action1* `: ELSE` *action2* `: END :`
(Attention au == pour traduire la condition d'égalité)

4.2.6 Les instructions "Pour"

`FOR I = A TO B STEP 1;` *action* `: END :`
`FOR I = A TO B STEP P;` *action* `: END :`

4.2.7 L'instruction "Tant que"

`WHILE` *condition* `REPEAT` *action* `: END :`

4.2.8 L'instruction “Repeter”

`DO` *action* `: UNTIL` *condition* `END :`

4.2.9 Un exemple : le crible d'Eratosthène

```
INPUT N;"CRIBLE";"N=";;10:
ERASE:
MAKELIST(I,I,1,N,1) -> L1:
0 -> L1(1):
2->P:
```

`WHILE P*P` $\leq$ `N REPEAT`

```
  FOR I = P TO INT(N/P) STEP 1;
    0->L1(I*P):
  END:
  DISP 3;""L1:
  P+1->P:
  WHILE P*P ≤ N AND L1(P) == 0 REPEAT
    P+1->P:
  END:
END:
{2}->L2:
FOR I=3 TO N  STEP 1;
 IF L1(I) ≠ 0 THEN
  CONCAT(L2,{I}) ->L2:
  END:
END:
DISP 3 ;"PREM" L2:
FREEZE:
```

4.2.10 Calcul du PGCD par l'algorithme d'Euclide

On écrit tout d'abord le sous-programme IN qui permet d'entrer deux nombres A et B :

```
INPUT A;"A";;;1:
INPUT B;"B";;;1:
ERASE:
```

puis on écrit le programme PGCD :

```
RUN IN:
DISP 3;"PGCD "{A,B}:
WHILE B ≠ 0 REPEAT
A MOD B ->R:
B ->A:
R ->B:
END:
DISP 4;"PGCD "A:
FREEZE:
```

4.2.11 Identité de Bézout par l'algorithme d'Euclide

```
RUN IN:
DISP 3;"BEZOUT "{A,B}:
{1,0,A} ->L1:
{0,1,B} ->L2:
WHILE L2(3) ≠ 0 REPEAT
```

```
L1-L2*FLOOR(L1(3)/L2(3)) ->L3:
L2 ->L1:
L3 ->L2:
END:
DISP 4;"U V PGCD "L1:
FREEZE:
```

4.2.12 Décomposition en facteurs premiers d'un entier N

Attention pour la `HP 40G` il faut remplacer dans ce qui suit `CLEAR L1` par `SYSEVAL 259589`.

```
INPUT N;"N";;;1:
ERASE:
0 ->K:
CLEAR L1: @ou  SYSEVAL 259589:
WHILE N MOD 2 == 0 REPEAT
1+K -> K:
N/2 -> N:
END:
IF K ≠ 0 THEN
{2,K} ->L1:
END:
3 ->D:
WHILE D*D ≤ N REPEAT
0 -> K:
WHILE N MOD D == 0 REPEAT
K+1 -> K:
N/D -> N:
END:
IF K ≠ 0 THEN
CONCAT (L1,{D,K}) -> L1:
END:
2+D -> D:
END:
IF N ≠ 1 THEN
CONCAT (L1, {N,1}) -> L1:
END:
DISP 3; "FACT" L1:
FREEZE:
```

4.2.13 Calcul de $A^P \ mod \ N$

Voir le sous-programme du programme de la méthode probabiliste de Mr Rabin.

4.2.14 La fonction "estpremier"

```
INPUT N;"N";;;1:
IF N MOD 2== 0 OR N MOD 3==0 OR N==1 THEN
0 ->P:
ELSE
1->P:
END:
IF N==2 OR N==3 THEN
1->P:
END:
5->I:
```

FLOOR($\sqrt{N}$)->J :
WHILE I $\leq$ J AND P REPEAT

```
IF N MOD I==0 OR N MOD (I+2)==0 THEN
0 ->P:
ELSE
I+6 ->I:
END:
END:
ERASE:
DISP 5;P:
FREEZE:
```

4.2.15 Méthode probabiliste de Mr Rabin

```
INPUT N;"N";;;1:
RANDSEED TIME:
1->I:
1->P:
WHILE I < 20 AND P==1 REPEAT
  FLOOR( RANDOM * (N-2))+2->K:
  N-1->M:
  @ Calcul de K puissance M mod N dans P.
  1->P:
  WHILE  0 < M REPEAT
    IF M MOD 2 == 0 THEN
      M / 2 -> M :
      (K * K)  MOD N ->K :
    ELSE
      K*P MOD N -> P:
      M - 1 -> M:
    END:
  END:
  @ P contient K puissance M mod N et M=N-1.
  I+1 ->I:
END:
ERASE:
```

```
IF P==1 THEN
  DISP 3;"PREMIER " N:
ELSE
  DISP 3;"NON PREMIER " N:
END:
FREEZE:
```

4.3 Les programmes selon la HP48 ou HP49G mode RPN

4.3.1 Les Entrées

```
'A' PROMPTSTO
```

4.3.2 Les Sorties

En général on affiche simplement les résultats sur la pile pour une réutilisation éventuelle, on écrira simplement :
`A B`
On peut aussi afficher le résultat tagué, on écrira alors :
`A "A=" ->TAG`

4.3.3 La séquence d'instructions ou action

Le séparateur est soit l'espace soit le retour à la ligne.

4.3.4 L'instruction d'affectation

Il faut utiliser la notation postfixée et la commande `STO` :
`2 A * 'B' STO`

4.3.5 Les instructions conditionnelles

`IF` *condition* `THEN` *action* `END`
`IF` *condition* `THEN` *action1* `ELSE` *action2* `END`
Attention on utilise la notation postfixée et == pour traduire la condition d'égalité.

4.3.6 Les instructions "Pour"

`A B FOR I` *action* `NEXT`
`A B FOR I` *action* `P STEP`

4.3.7 L'instruction "Tant que"

`WHILE` *condition* `REPEAT` *action* `END`

4.3.8 L'instruction “Repeter”

`DO` *action* `UNTIL` *condition* `END`

4.3.9 Un exemple : le crible d'Eratosthène

```
≪ {} {} 2 1 → N TA PREM P I
≪ 0 'X' 'X' 2 N 1 SEQ + 'TA' STO
  WHILE P P * N ≤ REPEAT
     P N P / FLOOR FOR I
        TA I P * 0 PUT 'TA' STO
```

```
      NEXT
      1 'P' STO+
      WHILE P P * N ≤ TA P GET 0 == AND REPEAT
        1 'P' STO+
      END
    END
    2 N FOR I
      IF TA I GET 0 ≠ THEN
        I 'PREM' STO+
      END
    NEXT
    PREM
  ≫
≫
```

Puis on stocke ce programme dans `CRIBLE ('CRIBLE' STO)`.

4.3.10 Calcul du PGCD par l'algorithme d'Euclide

-Version itérative

```
≪ 0 → A B R
  ≪ WHILE B 0 ≠ REPEAT
       A B MOD 'R' STO
       B 'A' STO
       R 'B' STO
    END
    A
  ≫
≫
```

Puis on stocke ce programme dans `PGCD ('PGCD' STO)`.

-Version récursive

```
≪ 0 → A B
  ≪ IF B 0 ≠ THEN
       B A B MOD PGCDR
    ELSE
       A
    END
  ≫
≫
```

Puis on stocke ce programme dans `PGCDR ('PGCDR' STO)`.

4.3.11 Identité de Bézout par l'algorithme d'Euclide

-Version itérative

```
<< {} -> A B R
  << {1 0} 'A' STO+
{0 1} 'B' STO+
WHILE B 3 GET 0 ≠ REPEAT
```

```
 A B A 3 GET B 3 GET / FLOOR * - 'R' STO
 B 'A' STO
 R 'B' STO
END
A
   >>
 >>
```

Puis on stocke ce programme dans `BEZOUT ('BEZOUT' STO)`.

-Version récursive avec les listes

```
<< {} -> A B T
  « IF B 0 ≠ THEN
B A B MOD BEZOUR 'T' STO
 T 2 GET DUP A B / FLOOR *
T 1 GET SWAP -
T 3 GET {} + + +
ELSE
 {1 0} A +
END
  >>
 >>
```

Puis on stocke ce programme dans `BEZOUR ('BEZOUR' STO)`.

4.3.12 Décomposition en facteurs premiers d'un entier N

```
<< 0 3 {} -> N K D FACT
  << WHILE N 2 MOD 0 ==
        REPEAT
        1 'K' STO+
        'N' 2 STO/
    END
    IF K 0 ≠ THEN
      {2 K} 'FACT' STO
   END
    WHILE N D D * ≥
      REPEAT
      0 'K' STO
      WHILE N D MOD 0 ==
        REPEAT
        1 'K' STO+
        'N' D STO/
      END
      IF K 0 ≠ THEN
       {D K} 'FACT' STO+
      END
      2 'D' STO+
    END
```

```
    IF N 1 ≠ THEN
      {N 1} 'FACT' STO+
    END
  >>
>>
```

Puis on stocke ce programme dans `FACTPREM ('FACTPREM' STO)`.

4.3.13 Calcul de $A^P \ mod \ N$

```
  « 1-> A P N PUI
  « WHILE P 0 > REPEAT
     IF P 2 MOD 1 == THEN
     A PUI * N MOD 'PUI' STO
     'P' STO-
     END
    P 2 / 'P' STO
    A A * N MOD 'A' STO
    END
    PUI
  »
  »
```

Puis on stocke ce programme dans `PUIMOD ('PUIMOD' STO)`.

4.3.14 La fonction "estpremier"

```
  « DUP √ FLOOR 0 5 -> N J PREM I
    « IF N 1 == N 2 MOD 0 == OR N 3 MOD 0 == OR THEN
    0 'PREM' STO
  ELSE
    1 'PREM' STO
  END
  IF N 2 == N 3 == OR THEN
    1 'PREM' STO
  END

WHILE PREM I J ≤ AND REPEAT
  IF N I MOD 0 == N I 2 + MOD 0 == OR THEN
     0 'PREM' STO
  ELSE
     I 6 + 'I' STO
  END
END
PREM
  »
»
```

Puis on stocke ce programme dans `ESTPREM ('ESTPREM' STO)`.

4.3.15 Méthode probabiliste de Mr Rabin

On suppose que l'on a écrit la fonction `PUIMOD` qui prend sur la pile trois arguments `A, K, N` et qui renvoie $\texttt{A}^{\texttt{K}}$ `mod N`.

```
<< 1 0 1 ->  N I K P
  << 0 RDZ
WHILE P 1 == 20 I > AND REPEAT
 1 'I' STO+
RAND N 2 - * FLOOR 2 + 'K' STO
K N 1 - N PUIMOD  'P' STO
END
IF P 1 ==  THEN 1 ELSE 0
END
 >>
>>
```

Puis on stocke ce programme dans `RABIN ('RABIN' STO)`.

4.4 Les programmes selon la HP49G mode Algébrique

4.4.1 Les Entrées

Pour entrer une variable locale `A`, on tape :
....0 → `A` ≪ `PROMPTSTO('A')`... ≫

4.4.2 Les Sorties

Seul le dernier résultat s'inscrit dans l'historique.
Pour des résultats intermédiaires on tape :

`DISP("A = " + A, 3)`

3 représente le numéro de la ligne
ou

`MSGBOX("A = "+ → STR(A))`

4.4.3 La séquence d'instructions ou action

Le ; est un séparateur d'instructions.
Le ; s'obtient en tapant en même temps sur `shift-rouge SPC`.

4.4.4 L'instruction d'affectation

On utilise la touche `STO` qui se traduit à l'écran de la calculatrice par : ▷
(que l'on notera : `STO▷`

4.4.5 Les instructions conditionnelles

noindent`IF` *condition* `THEN` *action* `END`
`IF` *condition* `THEN` *action1* `ELSE` *action2* `END`
(Attention au == pour traduire la condition d'égalité)

4.4.6 Les instructions "Pour"

`FOR (I, A, B)` *action* `NEXT`
`FOR (I, A, B )` *action* `STEP P`

4.4.7 L'instruction "Tant que"

`WHILE` *condition* `REPEAT` *action* `END`

4.4.8 L'instruction "Repeter"

`DO` *action* `UNTIL` *condition* `END`

4.4.9 Un exemple : le crible d'Eratosthène

```
≪ → N
  ≪ 0+SEQ('I','I',1,N,1) → TA
    ≪ 2 → P
      ≪ WHILE P * P ≤ N REPEAT
        FOR (I , P , FLOOR(N/P))
          PUT('TA',I*P,0)
        NEXT ;
         P + 1 STO▷ P;
        WHILE P*P ≤ N AND TA[P] == 0 REPEAT
          P + 1 STO▷ P
        END
      END ;
      {2} → PREM
        ≪ FOR (I,3,N)
            IF TA[I] ≠ 0 THEN
              PREM + I STO▷ PREM;
            END
        NEXT ;
        PREM
        ≫
      ≫
    ≫
  ≫
≫ STO▷ CRIBLE
```

4.4.10 Calcul du PGCD par l'algorithme d'Euclide

-Version itérative

```
≪ → A,B
  ≪ 0 → R
    ≪ WHILE B ≠ 0 REPEAT
        A MOD B STO▷ R;
        B STO▷ A;
        R STO▷ B
      END;
      A
    ≫
  ≫
≫ STO▷ PGCD
```

Puis par exemple, `PGCD(45,75)` pour l'exécuter.

-Version récursive

```
≪ → A,B
  ≪ IF B ≠ 0 THEN
      PGCDR(B,A MOD B)
    ELSE
      A
```

```
      END
   ≫
≫ STO▷ PGCDR
```

4.4.11 Identité de Bézout par l'algorithme d'Euclide

-Version itérative

```
≪ → A,B
   ≪ {} → R
      ≪ {1,0,A} STO▷ A;
         {0,1,B} STO▷ B;
         WHILE B[3] ≠ 0 REPEAT
            A − B ∗ FLOOR(A[3]/B[3]) STO▷ R;
            B STO▷ A;
            R STO▷ B
         END
         A
      ≫
   ≫
≫ STO▷ BEZOUT
```

Puis par exemple `BEZOUT(45,75)` pour l'exécuter.

-Version récursive avec les listes

```
≪ → A,B
   ≪ {} → T
      ≪ IF B ≠ 0 THEN
            BEZOUR(B, A MOD B) STO▷ T;
            {T[2], T[1] − T[2] ∗ FLOOR(A/B), T[3]}
         ELSE
            {1,0,A}
         END
      ≫
   ≫
≫ STO▷ BEZOUR
```

4.4.12 Décomposition en facteurs premiers d'un entier N

```
≪ → N
   ≪ 0 → K
      ≪ WHILE N MOD 2 == 0 REPEAT
            K + 1 STO▷ K;
            N/2 STO▷ N
         END;
         {} → FACTO
            ≪ IF K ≠ 0 THEN
                  FACTO + {2,K} STO▷ FACTO
               END;
               3 → D
                  ≪ WHILE D ∗ D ≤ N REPEAT
```

```
                    0 STO▷ K;
                    WHILE N MOD D == 0 REPEAT
                      K + 1 STO▷ K;
                      N/D STO▷ N;
                    END;
                    IF K ≠ 0 THEN
                      FACTO + {D, K} STO▷ FACTO
                    END;
                    D + 2 STO▷ D
                  END;
                  IF N ≠ 1 THEN
                    FACTO + {N, 1} STO▷ FACTO
                  END;
                  FACTO;
                ≫
            ≫
        ≫
    ≫
≫ STO▷ FACTPREM
```

4.4.13 Calcul de $A^P \ mod \ N$

```
≪ → A P N
  ≪ 1 → PUI
    ≪ WHILE P > 0 REPEAT
        IF P MOD 2 == 1 THEN
          A * PUI MOD N STO▷ PUI
          P − 1 STO▷ P;
        END;
        P/2 STO▷ P;
        A * A MOD N STO▷ A;
      END;
      PUI
    ≫
  ≫
≫ STO▷ PUIMOD
```

4.4.14 La fonction "estpremier"

```
≪ → N
  ≪ 0 → P
    ≪ IF N MOD 2 == 0 OR N MOD 3 == 0 OR N == 1THEN
        0 STO▷ P
      ELSE;
        1 STO▷ P;
      END;
      IF N == 2 OR N == 3 THEN
        1 STO▷ P;
      END;
```

```
        FLOOR(√N) → J
          ≪ 5 → I
            ≪ WHILE I ≤ J AND P REPEAT
                 IF N MOD I == 0 OR N MOD (I + 2) == 0 THEN
                   0 STO▷ P;
                 ELSE;
                   I + 6 STO▷ I;
                 END;
               END;
               P
            ≫
          ≫
      ≫
   ≫
≫ STO▷ ESTPREM
```

Puis par exemple `ESTPREM(45789)` pour l'exécuter.

4.4.15 Méthode probabiliste de Mr Rabin

On suppose que l'on a écrit la fonction `PUIMOD` qui a trois arguments `A, K, N` et qui renvoie $\texttt{A}^{\texttt{K}}\texttt{mod N}$.

```
≪ → N
  ≪ 1 → I
    ≪ 0 → K
      ≪ 1 → P
        ≪ RDZ(0);
          WHILE P == 0 AND I < 20 REPEAT
            1 + I STO▷ I;
            FLOOR((N − 2) ∗ RAND) + 2 STO▷ K;
            PUIMOD(K, N − 1, N) STO▷ P;
          END;
          IF P == 1 THEN
            1
          ELSE
            0
          END;
          P
        ≫
      ≫
    ≫
  ≫
≫ STO▷ RABIN
```

Puis par exemple `RABIN(45313)` pour l'exécuter.

Remarque :

On peut aussi utiliser la commande du calcul formel `POWMOD` et on écrit alors :

En mode `RPN` :

```
N MODSTO
K N 1 - POWMOD 'P' STO
```

à la place de :
```
K N 1 - N PUIMOD 'P' STO
```
En mode `Algèbrique` :
```
MODSTO(N);
POWMOD(K,N-1) STO▷ P
```
à la place de :
```
PUIMOD(K,N-1,N) STO▷ P
```

4.5 Les programmes selon la TI80

4.5.1 Les Entrées

```
 :Prompt A
:Prompt A,B
```
ou encore :
```
:Input "A=",A
```

4.5.2 Les Sorties

```
 :Disp "A=",A
```

4.5.3 La séquence d'instructions ou action

: indique la fin d'une instruction. Il faut noter qu'à chaque passage à la ligne le : est mis automatiquement.

4.5.4 L'instruction d'affectation

On utilise la touche `STO` qui se traduit à l'écran de la calculatrice par : →

4.5.5 Les instructions conditionnelles

```
 :If condition :Then : action : End
:IF condition :THEN : action1 : ELSE : action2 : END
```

4.5.6 Les instructions "Pour"

```
 :FOR (I,A,B ) : action : END
:FOR (I,A,B,P ) : action : END
```

4.5.7 L'instruction "Tant que"

```
 :LBL 1
:IF condition :THEN :action
:GOTO 1
:END
```

4.5.8 L'instruction "Repeter"

```
 :LBL 1
:action :IF non condition :THEN :GOTO 1
:END
```

4.5.9 Un exemple : le crible d'Eratosthène

Voici le programme CRIBLE : on entre N et ensuite la liste L_2 est égale à la liste des nombres premiers inférieurs ou égaux à N.

```
PROGRAM :CRIBLE
```

```
:INPUT N
: SEQ(I, I, 1, N, 1)− > L1
: 0− > L1(1)
:2 -> P
:LBL 1
:IF P*P ≤ N
:THEN
:FOR (I,P,INT(N/P))
: 0− > L1(I ∗ P)
:END
:P+1 -> P
:LBL 2
: IF (P ∗ P) ≤ N
:THEN
: IF L1(P) = 0
:THEN
:P+1 -> P
:GOTO 2
:END
:END
:GOTO 1
:END
:0 -> P
:FOR (I,2,N)
: IF L1(I) ≠ 0
:THEN
:P+1 ->P
: I− > L2(P)
:END
:END
:L2
```

Il n'y a pas ici de notions de variables locales ou de programmes avec des paramètres : toutes les variables sont globales.
Je n'ai pas trouvé la fonction booléenne `AND` : j'ai donc mis deux `IF` successifs.
La dernière instruction permet d'afficher la liste $L2$ des nombres premiers inférieurs ou égaux à N. Si à l'affichage la liste $L2$ n'est pas visible en entier, il faut appuyer sur ▷ pour faire défiler la liste.

4.5.10 Calcul du PGCD par l'algorithme d'Euclide

-Version itérative

```
PROGRAM :PGCD
:INPUT A
:INPUT B
:LBL 1
:IF B ≠ 0
:THEN
:A-INT(A/B)*B->R
```

```
:B->A
:R->B
:GOTO 1
:END
:DISP A
```

-Pas de version récursive.

4.5.11 Identité de Bézout par l'algorithme d'Euclide

-Version itérative avec les listes
Au début A et B contiennent les deux nombres pour lesquels on cherche l'identité de Bézout. LA, LB et LR de l'algorithme sont ici L_1, L_2 et L_3.

```
PROGRAM :BEZOUT
:INPUT A
:INPUT B
:{1,0,A}->L1
:{0,1,B}->L2
:LBL 1
:IF L2(3) ≠ 0
:THEN
:L1-L2*INT(L1(3)/L2(3))->L3
:L2->L1
:L3->L2
:GOTO 1
:END
:L1
```

-Pas de version récursive.

4.5.12 Décomposition en facteurs premiers d'un entier N

-Version itérative avec les listes
La liste `L1` joue le rôle de la liste `FACT`.

```
PROGRAM :FACTPRE
:INPUT N
:CLRLIST L1
:0->K
:1->I
:LBL 1
:IF FPART(N/2)=0
:THEN
:K+1->K
:N/2->N
:GOTO 1
:END
:IF K≠0
:THEN
:2->L1(1)
```

```
:K->L1(2)
:I+2->I
:END
:3->D
:LBL 2
: IF D * D ≤ N
:THEN
:0->K
:LBL 3
:IF FPART(N/D)=0
:K+1->K
:N/D->N
:GOTO 3
:END
: If K ≠ 0
:THEN
:D->L1(I)
:K->L1(I+1)
:I+2->I
:END
:D+2->D
:GOTO 2
:END
: IF N ≠ 1
:THEN
:N->L1(I)
:I+1->I
:1->L1(I)
:END
:DISP L1
```

4.5.13 Calcul de $A^P \ mod \ N$

Voir le sous-programme du programme de la méthode probabiliste de Mr Rabin.

4.5.14 La fonction "estpremier"

On traduit le dernier algorithme : le résultat est 0 (FAUX) ou 1 (VRAI).

```
PROGRAM :ESTPREM
:INPUT N
:0->K
: INT(√N)− > J
:1->P
:IF FPART(N/2)=0
:THEN
: IF N ≠ 2
:THEN
:0->P
```

```
:END
:ELSE
:IF FPART(N/3)=0
:THEN
: IF N ≠ 3
:THEN
:0->P
:END
:ELSE
:IF N=1
:THEN
:0->P
:END
:END
:END
:5->I
:LBL 1
:IF P
:THEN
: IF I ≤ J
:THEN
:FOR (S,1,2)
: IF FPART(N/I) = 0
:THEN
:0->P
:ELSE
:I+2->I
:END
:END
:I+2->I
:GOTO 1
:END
:END
:DISP P
```

4.5.15 Méthode probabiliste de Mr Rabin

// Calcul de K^M `mod N` dans `P`.

```
PROGRAM :PUIMOD
:1->P
:LBL 1
:IF 0 < M
:THEN
:IF FPART(M/2)=0
:THEN
:M/2->M
:K*K-N*INT(K*K/N)->K
:ELSE
```

```
:K*P-N*INT(K*P/N)->P
:M-1->M
:END
GOTO 1
:END
:RETURN
```
// P contient K^P mod N et M=N-1.
```
PROGRAM :RABIN
:INPUT N
:1->I
:1->P
:LBL 1
:IF P=1
:THEN
:IF I < 20
:THEN
:RANDINT(2,N-1)->K
:N-1->M
:PRGM_PUIMOD
:I+1->I
:GOTO 1
:END
:END
:IF P=1
:THEN
:DISP ''PREMIER''
:ELSE
:DISP ''NONPREMIER''
:END
```

4.6 Les programmes selon la TI83+

4.6.1 Les Entrées

```
 :Prompt A
:Prompt A,B
```
ou encore :
```
:Input "A=",A
```

4.6.2 Les Sorties

```
 :Disp "A=",A
```

4.6.3 La séquence d'instructions ou action

: indique la fin d'une instruction. Il faut noter qu'à chaque passage à la ligne le : est mis automatiquement.

4.6.4 L'instruction d'affectation

On utilise la touche `STO` qui se traduit à l'écran de la calculatrice par : →

4.6.5 Les instructions conditionnelles

```
 :If condition :Then : action : End
:If condition :Then : action1 : Else : action2 : End
```

4.6.6 Les instructions "Pour"

```
 :For (I,A,B ) : action : End
:For (I,A,B,P ) : action : End
```

4.6.7 L'instruction "Tant que"

```
 :While condition : action : End
```

4.6.8 L'instruction "Repeter"

```
 :Repeat condition : action : End
```

4.6.9 Un exemple : le crible d'Eratosthène

```
PROGRAM :CRIBLE
:Input N
: seq(I, I, 1, N)− > L1
: 0− > L1(1)
:2 -> P
:While P*P ≤ N
:For (I,P,int(N/P))
: 0− > L1(I ∗ P)
```

```
:End
:P+1 -> P
: While (P * P) ≤ N and L1(P) = 0
:P+1 -> P
:End
:End
:0 -> P
:For (I,2,N)
: If L1(I) ≠ 0
:Then
:P+1 ->P
: I− > L2(P)
:End
:End
: P− > dim(L2)
: Disp L2
```

4.6.10 Calcul du PGCD par l'algorithme d'Euclide

```
PROGRAM :PGCD
:Prompt A,B
:While B ≠ 0
:A-int(A/B)*B->R
:B->A
:R->B
:End
:A
```

4.6.11 Identité de Bézout par l'algorithme d'Euclide

```
PROGRAM :BEZOUT
:Prompt A,B
: {1, 0, A}− > L1
: {0, 1, B}− > L2
: While L2(3) ≠ 0
: L1 − L2 * int(L1(3)/L2(3))− > L3
: L2− > L1
: L3− > L2
:End
: L1
```

4.6.12 Décomposition en facteurs premiers d'un entier N

```
PROGRAM :FACTPREM
:Input N
:0->K
: 0− > L1
:While fPart(N/2)=0
:K+1->K
```

```
:N/2->N
:End
: If K ≠ 0
:Then
: augment({2,K},L₁)-> L₁
:End
:3->D
: While D * D ≤ N
:0->K
:While fPart(N/D)=0
:K+1->K
:N/D->N
:End
: If K ≠ 0
:Then
: augment({D,K},L₁)-> L₁
:End
:D+2->D
:End
: If N ≠ 1
:Then
: augment({N,1},L₁)-> L₁
:End
: dim(L₁) - 1-> dim(L₁)
: DispL₁
```

4.6.13 Calcul de $A^P \ mod \ N$

Voir le sous-programme du programme de la méthode probabiliste de Mr Rabin.

4.6.14 La fonction "estpremier"

```
PROGRAM :ESTPREM
:Input N
:0->K
: int(√N)-> J
: If N = 1 or fPart(N/2) = 0 or fPart(N/3) = 0
:Then
:0->P
:Else
:1->P
:End
: If N = 2 or N = 3
:Then
:1->P
:End
:5->I
: While P and I ≤ J
```

```
: If fPart(N/(I + 2)) = 0 or fPart(N/I) = 0
:Then
:0->P
:Else
:I+6->I
:End
:End
:Disp P
```

4.6.15 Méthode probabiliste de Mr Rabin

@ Calcul de K^M mod N dans P.

```
PROGRAM :PUIMOD
:1->P
:While 0 < M
:If fPart(M/2)=0
:Then
:M/2->M
:K*K-N*int(K*K/N)->K
:Else
:K*P-N*int(K*P/N)->P
:M-1->M
:End
:End
:Return
```

@ P contientK^P mod N et M=N-1.

```
PROGRAM :RABIN
:Input N
:1->I
:1->P
:While P=1 and I < 20
:randInt(2,N-1)->K
:N-1->M
:prgmPUIMOD
:I+1->I
:End
:If P=1
:Then
:Disp ''PREMIER''
:Else
:Disp ''NONPREMIER''
:End
```

4.7 Les programmes selon la TI89 92

4.7.1 Les Entrées

```
 :Prompt A
:Prompt A,B
```

ou encore :

```
:Input "A=",A
```

4.7.2 Les Sorties

```
 :Disp "A=",A
```

4.7.3 La séquence d'instructions ou action

: indique la fin d'une instruction. Il faut noter qu'à chaque passage à la ligne le : est mis automatiquement.

4.7.4 L'instruction d'affectation

on utilise la touche STO qui se traduit à l'écran de la calculatrice par : →

4.7.5 Les instructions conditionnelles

```
 :If condition Then : action : EndIf
:If condition Then : action1 : Else : action2 : EndIf
```

4.7.6 Les instructions "Pour"

```
 :For I,A,B : action : EndFor
:For I,A,B,P : action : EndFor
```

4.7.7 L'instruction "Tant que"

```
 :While condition : action : EndWhile
```

4.7.8 L'instruction "Repeter"

```
 :Loop :action :If condition :Exit :EndLoop
```

4.7.9 Un exemple : le crible d'Eratosthène

```
:crible(n)
:Func
:local tab,prem,i,p
:newList(n)->tab
:newList(n)->prem
:seq(i,i,1,n) ->tab
:0 -> tab[1]
:2 -> p
:While p*p ≤ n
```

```
:For i,p,floor(n/p)
:0 -> tab[i*p]
:EndFor
:p+1 -> p
:While p*p ≤ n and tab[p]=0
:p+1 -> p
:EndWhile
:EndWhile
:0 -> p
:For i,2,n
:If tab[i] ≠ 0 Then
:p+1 ->p
:i ->prem[p]
:EndIf
:EndFor
:Return left(prem,p)
:EndFunc
```

4.7.10 Calcul du PGCD par l'algorithme d'Euclide

-Version itérative

```
:pgcd(a,b)
:Func
:Local r
:While b ≠ 0
:mod(a,b)->r
:b->a
:r->b
:EndWhile
:Return a
:EndFunc
```

-Version récursive

```
:pgcd(a,b)
:Func
:If b ≠ 0 Then
:Return  pgcd(b, mod(a,b))
:Else
:Return a
:EndIf
:EndFunc
```

4.7.11 Identité de Bézout par l'algorithme d'Euclide

-Version itérative

```
:bezout (a,b)
:Func
:local r
:{1, 0, a}->a
:{0, 1, b}->b
:While b[3] ≠ 0
:a-b*floor(a[3]/b[3])->r
:b->a
:r->b
:EndWhile
:Return a
:EndFunc
```

-Version récursive avec les listes

```
:bezout (a,b)
:local L, q, r
:If b ≠ 0 Then
:floor(a/b)->q
:a-b*q->r
:bezout(b,r)->L
:Return {L[2], L[1]-L[2]*q, L[3]}
:Else
:Return {1, 0, a}
:EndIf
:EndFunc
```

4.7.12 Décomposition en facteurs premiers d'un entier N

```
:factprem1(n)
:Func
:Local fact,k,d,i
:newList(100)->fact
:0->k
:0->i
:While mod(n,2)=0
:k+1->k
:n/2->n
:EndWhile
:If k ≠0 Then
:2->fact[1]
:k->fact[2]
:2->i
:EndIf
:3->d
:While d*d ≤n
:0->k
:While mod(n,d)=0
```

```
:k+1->k
:n/d->n
:EndWhile
:If k ≠0 Then
:i+2->i
:d->fact[i-1]
:k->fact[i]
:EndIf
:d+2->d
:EndWhile
:If n ≠1 Then
:i+2->i
:n->fact[i-1]
:1->fact[i]
:EndIf
:Return left(fact,i)
:EndFunc
```

ou bien
on construit la liste `fact` au fur et à mesure et on obtient le programme :

```
:factprem2(n)
:Func
:Local fact,k,d,i
:0->k
:{}->fact
:While mod(n,2)=0
:k+1->k
:n/2->n
:EndWhile
:If k ≠0 Then
:augment({2,k},fact)->fact
:EndIf
:3->d
:While d*d ≤n
:0->k
:While mod(n,d)=0
:k+1->k
:n/d->n
:EndWhile
:If k ≠0 Then
:augment({2,k},fact)->fact
:EndIf
:d+2->d
:EndWhile
:If n ≠1 Then
```

```
:augment({2,k},fact)->fact
:EndIf
:Return fact
:EndFunc
```

4.7.13 Calcul de $A^P \ mod \ N$

```
:puismod(a,p,n)
:Func
:Local pui
:1->pui
:While p>0
:If p mod 2 = 1 Then
:mod(a*pui,n)->pui
:p-1->p
:EndIf
:p/2->p
:mod(a*a,n)->a
:EndWhile
:Return pui
:EndFunc
```

4.7.14 La fonction "estpremier"

On traduit le dernier algorithme : le résultat est false (FAUX) ou true (VRAI).

```
:estprem(n)
:Func
:Local i, j, prem
:If mod(n,2)=0 or mod(n,3)=0 or n=1 Then
:false->prem
:Else
:true->prem
:EndIf
:If n=2 or n=3 Then
:true->prem
:EndIf
:5->i
:Floor(√(n))->j
:While i ≤j and prem
:If mod(n,i)=0 or mod(n,i+2)=0 Then
:false->prem
:Else
:i+6->i
:EndIf
:EndWhile
:Return prem
:EndFunc
```

4.7.15 Méthode probabiliste de Mr Rabin

```
:rabin(n)
:Func
:Local k,i,p
:1->p
:1->i
:RandSeed 1147
:While p=1 and i<20
:rand(n-1)+1->k
:puimod(k,n-1,n)->p
:i+1->i
:EndWhile
:If p=1 Then
:Return true
:Else
:Return false
:EndIf
:EndFunc
```

4.8 Les programmes selon la SHARP EL9600

4.8.1 Les Entrées

`Input A`
ou
`Input A`
`Input B`

4.8.2 Les Sorties

`Print A` affiche la valeur de `A`
ou `Print "A` affiche `A`, puis
`Print A` affiche la valeur de `A`

4.8.3 La séquence d'instructions ou action

C'est `ENTER` qui est utilisé comme séparateur d'instructions.
Chaque instruction doit être écrite sur une seule ligne.

4.8.4 L'instruction d'affectation

On utilise la touche `STO` qui se traduit à l'écran de la calculatrice par :
⇒

4.8.5 Les instructions conditionnelles

`If` *non condition* `Goto FSI`
action
`Label FSI`

`If` *non condition* `Goto SINON`
action1
`Goto FSI`
`Label SINON`
action2
`Label FSI`

4.8.6 Les instructions "Pour"

`A` ⇒ `I`
`Label DPOUR`
`If I>B Goto FPOUR`
action1
`I` + `1` ⇒ `I`
`Goto DPOUR`
`Label FPOUR`

4.8.7 L'instruction "Tant que"

```
Label DTANTQUE
If non condition Goto FTANTQUE
action
Goto DTANTQUE
Label FTANTQUE
```

4.8.8 L'instruction "Repeter"

```
Label REPETER
action
If non condition Goto REPETER
```

4.8.9 Un exemple : le crible d'Eratosthène

```
CRIBLE
Input N
{2} ⇒ L2
seq(X, 1, N) ⇒ L1
0 ⇒ L1
2 ⇒ P
Label DTQ1
If P*P>N Goto FTQ1
P ⇒ I
Label DP1
If I>int (N/P) Goto FP1
0 ⇒ L1(I * P)
I + 1 ⇒ I
Goto DP1
Label FP1
P + 1 ⇒ P
Label DTQ2
If (P * P > N) or (L1(P) ≠ 0) Goto FTQ2
P + 1 ⇒ P
Goto DTQ2
Label FTQ2
Goto DTQ1
Label FTQ1
3 ⇒ I
1 ⇒ P
Label DP2
If I>N Goto FP2
If L1(I) = 0) Goto FSI
P + 1 ⇒ P
I ⇒ L2(P)
Label FSI
I + 1 ⇒ I
Label FP2
```

```
Print L2
End
```

4.8.10 Calcul du PGCD par l'algorithme d'Euclide

```
PGCD
Input A
Input B
Label DTQ
If B=0 Goto FTQ
A − int (A/B) ∗ B ⇒ R
B ⇒ A
R ⇒ B
Goto DTQ
Label FTQ
Print ''PGCD
Print A
End
```

4.8.11 Identité de Bézout par l'algorithme d'Euclide

```
BEZOUT
Input A
Input B
{1, 0, A} ⇒ L1
{0, 1, B} ⇒ L2
Label DTQ
If L2(3)=0 Goto FTQ
L1 − int (L1(3)/L2(3)) ∗ L2(3) ⇒ L3
L2 ⇒ L1
L3 ⇒ L2
Goto DTQ
Label FTQ
Print ''BEZOUT
Print L1
End
```

4.8.12 Décomposition en facteurs premiers d'un entier N

```
FACTPREM
Input N
{0} ⇒ L1
0 ⇒ K
Label DTQ1
If fpart(N/2) ≠ 0 Goto FTQ1
K + 1 ⇒ K
N/2 ⇒ N
Goto DTQ1
Label FTQ1
```

```
If K = 0 Goto FS1
augment({2,K},L1) ⇒ L1
Label FS1
3 ⇒ D
Label DTQ2
If D*D>N Goto FTQ2
0 ⇒ K
Label DTQ3
If fpart(N/D) ≠ 0 Goto FTQ3
K + 1 ⇒ K
N/D ⇒ N
Goto DTQ3
Label FTQ3
If K = 0 Goto FS2
augment({D,K},L1) ⇒ L1
Label FS2
D + 2 ⇒ D
Goto DTQ2
Label FTQ2
If N = 1 Goto FS3
augment({N,1},L1) ⇒ L1
Label FS3
dim(L1) − 1 ⇒ dim(L1)
Print L1
End
```

4.8.13 Calcul de $A^P \ mod \ N$

Voir le sous-programme du programme de la méthode probabiliste de Mr Rabin.

4.8.14 La fonction "estpremier"

```
ESTPREM
Input N
int (√N) ⇒ J
If (N ≠ 1) and (fpart(N/2) ≠ 0) and (fpart(N/3) ≠ 0) Goto SINON1
0 ⇒ P
Goto FSI1
Label SINON1
1 ⇒ P
Label FSI1
If (N ≠ 2) and (N ≠ 3) Goto FSI2
1 ⇒ P
Label FSI2
5 ⇒ I
Label DTQ
If (P=0) or (I>J) Goto FTQ
If (fpart(N/I) ≠ 0) and (fpart(N/(I + 2)) ≠ 0) Goto SINON3
```

```
0 ⇒ P
Goto FSI3
Label SINON3
I + 6 ⇒ I
Label FSI3
Goto DTQ
Label FTQ
Print P
End
```

4.8.15 Méthode probabiliste de Mr Rabin

```
RABIN
Input N
1 ⇒ I
1 ⇒ P
Label DTQ1
If (I = 20) or (P ≠ 1) Goto FTQ1
int(random ∗ (N − 2)) + 2 ⇒ K
N − 1 ⇒ M
Gosub PUIMOD
I + 1 ⇒ I
Goto DTQ1
Label FTQ1
If P ≠ 1 Goto SINON2
Print ''premier
Goto FSI2
Label SINON2
Print ''nonpremier
Label FSI2
End
Label PUIMOD
Rem Calcul de K puissance M mod N dans P.
1 ⇒ P
Label DTQ2
If M=0 Goto FTQ2
If fpart(M/2) ≠ 0 Goto SINON1
M/2 ⇒ M
K ∗ K − int(K ∗ K/N) ∗ N ⇒ K
Goto FSI1
Label SINON1
K ∗ P − int(K ∗ P/N) ∗ N ⇒ P
M − 1 ⇒ M
Label FSI1
Goto DTQ2
Label FTQ2
Rem P contient K puissance M mod N et M=N-1.
Return
```

Chapitre 5

Algorithmique et Maple

5.1 Pour commencer

Attention
`Maple` est sensible à la différence entre majuscules et minuscules.

5.1.1 Comment éditer et sauver un programme

Traduction Maple sous Linux

On lance `Maple` depuis une fenêtre `xterm` en tapant :
`xmaple &`.
Nous allons écrire une fonction appelée `ma_procedure` qui calculera la moyenne de deux nombres. Votre texte doit ressembler à :

```
ma_procedure:=proc (x1,x2)
RETURN((x1+x2)/2);
end: #End of ma_procedure
```

Traduction Maple sous Windows

Sur un PC lancer `Maple` par un double clic sur l'icône approprié.
Pour éditer un programme, on est alors dans un éditeur plein écran, avec des `zones d'entrées (ou de saisie d'ordres)`, des `zones de textes` et des `zones de sorties`.
Les zones d'entrées débutent par les symboles `[>`.
Sur une ligne d'entrée, on écrit des ordres, ou commandes `Maple`, qui peuvent être immédiatement exécutés.
Après chaque ordre, terminé par `;` et envoyé par la touche `Entrée`, la sortie correspondante est affichée dans une zone de sortie, gérée par `Maple` et une nouvelle zone d'entrée est automatiquement créée.
Un ordre, terminé par `:` et envoyé par la touche `Entrée` sera exécuté sans entraîner de sortie.
On peut créer, des lignes d'entrée, au-dessous de la position du curseur, en faisant un clic, dans la barre des icônes, sur `[>`.
Pour écrire un texte, on peut transformer une ligne d'entrée en ligne de texte, grâce à l'icône `T`.
On dispose alors de plusieurs styles, disponibles dans un menu déroulant.

On peut aussi écrire un commentaire dans une ligne d'entrée, en utilisant le caractère `#`. (la fin de la ligne est alors considérée comme un texte.

```
...
[> ma_procedure := proc(x1, x2);
RETURN((x1 + x2)/2);
end; # fin de ma_procedure
```

À la fin de chaque ligne, taper Majuscule-Entrée (2 touches enfoncées ensemble)
Après la dernière ligne, on tapera simplement sur `Entrée`.
Le ; après `end` entraînera l'affichage du texte de la procedure, si la syntaxe est correct.

5.1.2 Comment corriger un programme

Pour modifier un programme existant, ou pour corriger une erreur de syntaxe, on revient dans le texte même, à l'endroit voulu (avec un clic de la souris). On réalise les modifications du texte, et on termine en tapant `Entrée` (sans forcément revenir en fin de texte).
La procédure est de nouveau analysée et affichée (si elle est correcte).

5.1.3 Comment exécuter un programme

Les procédures de `Maple` sont essentiellement des fonctions.
`ma_procedure(5,13)` est un nombre simplement affiché ou utilisé dans d'autres procédures...

5.1.4 Comment améliorer puis sauver sous un autre nom un programme

En général on sauvegarde toute une feuille de travail (worksheet), contenant des procédures éventuellement, et peut-être aussi des ordres éxécutés en mode immédiat.
L'environnement habituel, menu déroulant Fichier (ou File), options Enregistrer (Save) ou Enregistrer sous (Save As...), permet de créer des fichiers au format mws (fichiers sources `Maple`)

5.2 Les différentes instructions

5.2.1 Les commentaires

Il faut prendre l'habitude de commenter les programmes. En algorithmique un commentaire commence par
`//` et se termine par un passage à la ligne.

Traduction MapleV

Un commentaire commence par `#` et se termine par un passage à la ligne.

5.2.2 Les variables

Leurs noms

Ce sont les endroits où l'on peut stocker des valeurs, des nombres, des expressions.
Un nom doit commencer par une lettre, ne pas contenir d'espace, de symbole opératoire (+, -, ...) et ne pas être un mot réservé (D, I, ... pour `MapleV`)
Avec `MapleV` on peut utiliser des noms ayant plus de 8 caractères.

Notion de variables locales

Pour `MapleV`, il faut définir les variables locales en début de programme en écrivant :
`local a,b`

Notion de paramètres

Quand on écrit une fonction il est possible d'utiliser des paramètres.
Par exemple si `A` et `B` sont les paramètres de la fonction `PGCD` on écrit :
`fonction PGCD(A,B)....ffonction`

Traduction MapleV

```
PGCD :=proc(A,B)
....
....
end :
```

Ces paramètres NE se comportent PAS comme des variables locales : ils sont initialisés lors de l'appel de la fonction mais ne peuvent pas être changés au cours de la procédure. L'exécution se fait en demandant par exemple :
`PGCD(15,75)`

5.2.3 Les Entrées

Pour que l'utilisateur puisse entrer une valeur dans la variable `A` au cours de l'exécution d'un programme, on écrira, en algorithmique :
`saisir A`
Et pour entrer des valeurs dans `A` et `B` on écrira :
`saisir A,B`

Traduction MapleV

On peut utiliser :
`A := readline() ;` ou
`A :=readstat('A= ?') ;` mais la plupart des entrées se font par passage de paramètres.

5.2.4 Les Sorties

En algorithmique on écrit :
`Afficher "A=",A`

Traduction MapleV

`print('A='.A) :`
On préférera ‘ (accent grave) à " (guillemet), comme délimiteur de chaîne, car le premier n'est pas affiché, alors que le deuxième l'est.

5.2.5 La séquence d'instructions ou action

Une action est une séquence d'une ou plusieurs instructions.
En langage algorithmique, on utilisera l'espace ou le passage à la ligne pour terminer une instruction.

Traduction MapleV

`:` ou `;` indique la fin d'une instruction.
Une instruction terminée par (`;`) génére une sortie et celle terminée par (`:`) n'en génére pas.

5.2.6 L'instruction d'affectation

L'affectation est utilisée pour stocker une valeur ou une expression dans une variable.
En algorithmique on écrira par exemple : `2*A->B`
pour stocker `2*A` dans `B`
ATTENTION
Avec `Maple` on écrit :
`b :=2*a` pour stocker `2*a` dans `b`.

5.2.7 Les instructions conditionnelles

`Si <`*condition*`> alors <`*action*`> fsi`
`Si <`*condition*`> alors <`*action1*`> sinon <`*action2*`> fsi`
Exemple :
`Si A = 10 ou A < B alors B-A->B sinon A-B->A fsi`

Traduction MapleV

Si ... alors
`if <`*condition*`> then <`*action*`> fi ;`
Si ... alors ... sinon ...
`if <`*condition*`> then <`*action1*`> else <`*action2*`> fi ;`
Le schéma général à n cas :
`if <`*condition_1*`> then <`*action_1*`>`
`elif <`*condition_2*`> then <`*action_2*`>`
`... elif <`*condition_n-1*`> then <`*action_n-1*`> else <`*action_n*`> fi ;`

5.2.8 Les instructions "Pour"

`Pour I de A à B faire <action> fpour`
`Pour I de A à B (pas P) faire <action> fpour`

Traduction MapleV

Voici la syntaxe exacte :
`for <nom> from <expr> by <expr> to <expr>`
`do <action> od ;`
Par exemple
`for i from 1 to n do <action :> od`
`for i from 1 by p to n do <action :> od`

5.2.9 L'instruction "Repeter"

`Repeter <action> jusqua < condition>`

Traduction MapleV

Pas de schéma répéter en `MapleV`

5.2.10 L'instruction "Tant que"

`Tant que <condition> faire <action> ftantque`

Traduction MapleV

`while <condition> do <action> od` :

5.2.11 Les conditions ou expressions booléennes

Une condition est une fonction qui a comme valeur un booléen, à savoir elle est soit `vraie` soit `fausse`.

Les opérateurs relationnels

Pour exprimer une condition simple on utilise les opérateurs :
`=` `<` `>` $\leq$ $\geq$ $\neq$

Les opérateurs logiques

Pour traduire des conditions complexes, on utilise les opérateurs logiques :
`ou et non`
qui se traduisent sur les calculatrices et en `MapleV` par :
`or and not`

5.2.12 Les fonctions

Dans une fonction on ne fait pas de saisie de données : on utilise des paramètres qui seront initialisés lors de l'appel.
Dans une fonction on veut pouvoir réutiliser le résultat :
on n'utilise pas la commande `affichage` mais la commande `retourne`.
On écrit par exemple en algorithmique :

```
fonction addition(A,B)
retourne A+B
ffonction
```

Cela signifie que :
-Si on fait exécuter la fonction, ce qui se trouve juste après `retourne` sera affiché. Les instructions qui suivent `retourne` seront ignorées.

Traduction MapleV

```
addition:= proc(a,b)
RETURN(a+b);
end:
```

5.2.13 Les listes

On utilise les { } pour délimiter une liste.
Attention!!! En algorithmique on a choisit cette notation car c'est celle qui est employée par les calculatrices...à ne pas confondre avec la notion d'ensemble en mathématiques : dans un ensemble l'ordre des éléments n'a pas d'importance mais dans une liste l'ordre est important...
Par exemple {} désigne la liste vide et {1, 2, 3} est une liste de 3 éléments.
`append` sera utilisé pour concaténer 2 listes ou une liste et un élément ou un élément et une liste :
`{1, 2, 3}->TAB`
`append(TAB, 4) ->TAB` (maintenant `TAB` désigne `{1, 2, 3, 4}`
`TAB[2]` désigne le deuxième élément de `TAB` ici `2`.

Traduction MapleV

En `Maple` on utilise `{ }`, comme en mathématiques, pour représenter un **ensemble**.
Exemple : `De := {1, 2, 3, 4, 5, 6}`
Dans un ensemble, l'ordre n'a pas d'importance. La répétition est interdite.

Pour délimiter une **liste**, on utilise `[ ]`.
L'ordre est pris en compte, la répétition est possible.
Exemple : `[Pile, Face, Pile]`

Une **séquence** est une suite d'objets, séparés par une virgule.

Si `C` est un ensemble ou une liste, `op(C)` est la séquence des objets de `C`.

`nops(C)` est le nombre d'éléments de `C`.
Ainsi, si `L` est une liste, `{op(L)}` est l'ensemble des objets (non répétés) de `L`.
Exemples
On écrit :
`S :=NULL :` (pour la séquence vide)
`S :=S,A :` (pour ajouter un élément à `S`)
Une liste est une séquence entourée de crochets :
`L :=[S] :`
`L[i]` est le iéme élément de la liste `L`.
On peut aussi revenir à la séquence : `S :=op(L) :`

5.3 Exemple : le PGCD par l'algorithme d'Euclide

Soient A et B deux entiers positifs dont on cherche le $PGCD$.
L'algorithme d'Euclide est basé sur la définition récursive du $PGCD$:

$$\begin{aligned} PGCD(A,0) &= A \\ PGCD(A,B) &= PGCD(B, A \mod B) \; si \; B \neq 0 \end{aligned}$$

où $A \mod B$ désigne le reste de la division euclidienne de A par B.
Voici la déscription de cet algorithme :
on effectue des divisions euclidiennes successives :

$$\begin{aligned} A = &\quad B \times Q_1 + R_1 \quad 0 \leq R_1 < B \\ B = &\quad R_1 \times Q_2 + R_2 \quad 0 \leq R_2 < R_1 \\ R_1 = &\quad R_2 \times Q_3 + R_3 \quad 0 \leq R_3 < R_2 \\ &..... \end{aligned}$$

Après un nombre fini d'étapes, il existe un entier n tel que : $R_n = 0$.
on a alors : $PGCD(A,B) = PGCD(B,R_1) = ... = PGCD(R_{n-1},R_n) = PGCD(R_{n-1},0) = R_{n-1}$

5.3.1 Traduction algorithmique

-Version itérative
Si B $\neq$ 0 on calcule R=A mod B, puis avec B dans le rôle de A (en mettant B dans A) et R dans le rôle de B (en mettant R dans B) on recommence jusqu'à ce que B=0, le PGCD est alors A.

```
Fonction PGCD(A,B)
Local R
tant que B ≠ 0 faire
  A mod B->R
  B->A
  R->B
ftantque
r\'esultat A
ffonction
```

-Version récursive

On écrit simplement la définition récursive vue plus haut.

```
Fonction PGCD(A,B)
Si B ≠ 0 alors
  r\'esultat PGCD(B,A mod B)
  sinon
  r\'esultat A
fsi
ffonction
```

Traduction MapleV

-Version itérative :

```
pgcd:=proc(x,y)
local a,b,r:
a:=x:
b:=y:
While (b>0) do
r:=irem(a,b):
a:=b:
b:=r:
od:
RETURN(a):
end:
```

-Version récursive :

```
pgcd:=proc(a,b)
if (b=0) then
RETURN(a)
 else
RETURN(pgcd(b,irem(a,b))):
fi:
end:
```

Chapitre 6

Algorithmique et Mathematica

6.1 Pour commencer

6.1.1 Comment éditer et sauver un programme

Sur un PC sous Windows ou sur un Mac, double-cliquer sur l'icône `Mathematica`. Un nouveau notebook (fichier `Mathematica`) est automatiquement ouvert. Taper les commandes du programme. Elles s'inscrivent dans une cellule `Input` : un crochet de type `Input`, à droite, enserre les lignes de commandes. La touche `retour chariot` permet d'aller à la ligne . La touche `Enter` du pavé numérique ou `Maj-Retour chariot` valide le programme.

Nous n'utiliserons pas l'éditeur d'équations de `Mathematica` qui permet d'entrer les commandes directement en notation mathématique.
Exemple :

```
maProcedure[x1_,x2_] :=(x1+x2)/2
```

Sauvegarder le fichier comme n'importe quel fichier Windows ou Mac (Menu File puis Save). Le suffixe .nb (notebook) est automatiquement ajouté aux fichiers `Mathematica`.

6.1.2 Comment corriger un programme

`Mathematica` dispose d'un procédé qui suit les évaluations pas à pas : `On[]` déclenche le mécanisme d'évaluation pas à pas. `Off[]` l'arrête. On peut aussi utiliser `Trace[expr]` qui fournit une liste d'expressions intermédiaires générées par l'évaluation d'expressions.
Exemple :

```
On[] ; maProcedure[5, 7]
Off[]
```

La réponse est :

```
On : :trace : On[ ] - -> Null.
maProcedure : :trace : maProcedure[5, 7] - -> (5+7)/2.
Plus : :trace : 5+7 - ->12.
Power : :trace : 1/2 - -> 1/2.
Times : :trace : (5+7)/2 - -> 12/2.
```

Times : :trace : $\frac{12}{2}$ - -> 6.
ConpoundExpression :trace : On[ ] ; maProcedure[5, 7] - -> 6.
6

Trace[maProcedure[5, 7]]
La réponse est :
{maProcedure[5, 7], $\frac{5+7}{2}$, {5+7,12}, { $\frac{1}{2}$, $\frac{1}{2}$}, $\frac{12}{2}$, 6}.

6.1.3 Comment éxécuter un programme

Mathematica est un système interactif : pour insérer une commande, on clique dans le notebook, un trait horizontal s'affiche dans la largeur de la page, et on tape la commande.
Par exemple pour obtenir la moyenne de 5 et 13 on tape :
maProcedure[5, 13]
la réponse est :
9

6.1.4 Comment améliorer un programme

Il suffit de modifier le code puis de valider le programme.

6.2 Les différentes instructions

6.2.1 Les commentaires

Un commentaire commence par (* et se termine par *).
On peut aller à la ligne à l'intérieur d'un commentaire.
Un commentaire comporte autant de lignes que l'on veut.
Exemple :

```
(* Ceci est un commentaire
qui peut faire plusieurs lignes *)
```

6.2.2 Les variables

Leurs noms

Les variables Mathematica sont des symboles représentés par toute chaîne alphanumérique, sans limitation de longueur ne commençant pas par un chiffre. Le symbole ne doit bien sûr pas être utilisé par le système, auquel cas un message d'erreur prévient l'utilisateur.

Notion de variables locales

Deux niveaux de localisation des variables sont possibles :
- La localisation dynamique (ou par valeurs)
  Elle se fait à l'aide de la commande Block. Le premier argument de Block est écrit à l'intérieur de crochets : il désigne la liste des variables locales et ces variables peuvent être eventuellement initialisées.

Exemple :
`f[x1_, x2_,..] :=Block[{a,b,...}, sequence d'instructions]`
Ici `a` et `b` sont des variables locales. À l'appel de `f`, si `a` (ou `b`) était déjà affecté `Block` met sa valeur de coté et la restitue après l'execution de la procédure `f`. Mais il reste des possibilités de conflit entre les symboles désignant ces variables. La localistion dynamique sera donc restreinte au cas où la procédure n'opère que sur les valeurs des variables.
Les fonctions :
`Do, Table Sum, NSum, Product,NProduct, NIntegrate` (mais pas `Integrate`) bénéficient d'une localisation dynamique des indices d'itération.

– La localisation statique (ou lexicale)
On utilise `Module`, dont la syntaxe est similaire à celle de `Block`. Mais `Module` renomme les variables locales, réduisant quasiment à néant les risques de conflit avec une variable non locale qui aurait la même désignation. Cependant, `Module` est plus lent que `Block`.
Exemple :
`f[x1_, x2_,..] :=Module[{a,b,...}, sequence d'instructions]`

Notion de paramètres

Une fonction `Mathematica` comporte la plupart du temps un ou plusieurs paramètres.
Exemple :
`PGCD[a_,b_] :=(...;...;...)`
Ces paramètres sont initialisés lors de l'appel de la fonction.
Attention : si `a` ou `b` sont affectés à l'intérieur de la procédure, les arguments de la fonction (à son appel) seront nécessairement des variables symboliques.

6.2.3 Les Entrées

On peut utiliser :
`A=Input[]`
ou encore
`A=Input["A= ?"]`
Si on utilise l'interface utilisateur normale, une boîte de dialogue s'ouvre.

6.2.4 Les Sorties

`Print["A= ",A]`

6.2.5 La séquence d'instructions ou action

On enchaîne les instructions sur une même ligne (ou pas) en les séparant par un point virgule (;). Mais les instructions suivies d'un ; ne provoquent pas de sortie. Pour obtenir une sortie pour chaque instruction, il faut les enchaîner par un passage à la ligne (touche "retour chariot").

6.2.6 L'instruction d'affectation

Deux affectations sont possibles : l'affectation immédiate (=) et l'affectation différée (:=).
Si on utilise = `Mathematica` évalue le membre de droite et affecte le résultat au membre de gauche.
Si on utilise := l'affectation est différée, `Mathematica` n'évalue le membre de droite qu'à l'appel du membre de gauche et produit l'affectation.
On retiendra que la définition d'une fonction se fait plutôt avec une affectation différée.

6.2.7 Les instructions conditionnelles

Toutes les instructions sont des fonctions à plusieurs variables.
La commande `If` autorise une logique à 3 états.
`If[condition,action si vrai, action si faux,(optionnel) action si indéterminé]`
Exemple :

```
signe[n_] := If[n>=0,Print[''positif''],Print[''negatif''], Print[''je ne sais pas'']
```

Ainsi, la réponse de `signe[''maths'']` est `je ne sais pas` .

6.2.8 Les instructions "Pour"

`For[initialisation,test,increment,instructions]`
À l'appel de `For`, `Mathematica` évalue l'initialisation de la variable, puis évalue les instructions et incrémente la variable (selon les spécifications du programmeur) jusqu'à ce que le test ne soit plus vrai.
Exemple :

```
For[i=0,i<3,i++,Print[i]]
```

La réponse est :
0
1
2

6.2.9 L'instruction "Repeter"

Il n'y a pas vraiement de schéma "répéter jusqu'à" mais il existe une commande "répéter".
`Do[instructions,{n}]` : répéte `n` fois les instructions.
`Do[instructions,{n, nmax}]` : répéte les instructions pour `n` variant de 1 à `nmax`, par `pas` de 1.
`Do[instructions,{n, nmin, nmax}]` : répéte les instructions pour `n` variant de `nmin` à `nmax`, par `pas` de 1.
`Do[instructions,{n, nmin, nmax, p}]` : répéte `n` fois les instructions pour `n` variant de `nmin` à `nmax`,par `pas` de `p`.
La boucle n'est plus effectuée quand `n` dépasse `nmax`.
Exemple :

```
Do[Print[i],{i, 1.2, 3.3, 0.5}]
```

La réponse est :
```
1.2
1.7
2.2
2.7
3.2
```

6.2.10 L'instruction "Tant que"

`While[test,instructions]`
`test` est évalué, si le `test` est vrai, on effectue les instructions, puis on recommence (on évalue `test` ...) jusqu'à ce que ce `test` ne soit plus vrai.
Exemple :
`i=0 ;While[i<3,Print[i] ;i++]`
La réponse est :
```
0
1
2
```

6.2.11 Les conditions ou expressions booléennes

Les opérateurs relationnels sont :
== < > <= >= ≠
Les opérateurs logiques sont :
`Or`(∨), `And`(∧), `Not`(¬), `Xor`, `Implies`(⇒)
Ce sont des opérateurs préfixés.
Exemple :
`And[2<5, 3<8]`
La réponse est :
`True`

6.2.12 Les fonctions

En `Mathematica`, il n'est pas necessaire d'employer `Return` pour afficher un résultat, mais `Return` permet de définir la valeur d'une fonction. Ainsi, `Return[<résultat>]` force l'évaluation de ce résultat et fait sortir immédiatement de la fonction.
Exemple :
`addition[a_, b_] := Return[a+b]` puis on tape :
`addition[2,3]`
la réponse est : 5
On remarquera l'utilisation des crochets pour mettre les paramètres.

6.2.13 Les listes

Une liste est une suite d'objets séparés par des virgules, à l'intérieur de deux accolades.
`{}` est la liste vide.
De nombreuses fonctions opèrent sur les listes.

Exemple :
`l = {1, 2, 2, 3}`
`Length[l]` est le nombre d'éléments de `l`, dans l'exemple la réponse est : 4
`l[[4]]` désigne le quatrième élément ici 3.
`Append[l,4]` désigne la liste formée par `l` et `4` ici c'est la liste :
`{1, 2, 2, 3, 4}` (la liste `l` n'est pas modifiée).
`AppendTo[l,4]` modifie la liste `l` en mettant l'élément `4` à la fin de la liste : dans l'exemple `l` est alors égale à :
`{1, 2, 2, 3, 4}`
`Prepend[l,0]` et `PrependTo[l,0]` met de la même façon l'élément `0` au début de la liste `l`.
`Join[{1, 2, 3},{2, 3, 4, 5}]` concaténe les deux listes en :
`{1, 2, 3, 2, 3, 4, 5}`
`Union[{1, 2, 3},{2, 3, 4, 5}]` concaténe les deux listes en :
`{1, 2, 3, 4, 5}` en éliminant les doublons.
`Union[ {1, 5, 3, 2, 3, 4, 5}]` appliqué à une seule liste élimine les doublons et trie la liste, dans la mesure du possible (dans l'exemple la réponse est `{1, 3, 4, 5}`).
`Intersection[{1, 2, 2},{5, 4, 2, 2}]` donne l'intersection de deux listes, ces listes sont interprétées comme des ensembles et les doublons sont éliminés (dans l'exemple la réponse est `{2}`).

On peut aussi faire d'autres opérations sur les listes :

- le produit scalaire est obtenu avec `.` entre les deux listes :
  `{1, 2, 3}.{4, 2, 1}` donne `11`
- le produit terme à terme est obtenu avec `*` entre les deux listes :
  `{1, 2, 3}*{4, 2, 1}` donne `{4, 4, 3}`
- la somme terme à terme est obtenue avec `+` entre les deux listes :
  `{1, 2, 3}+{4, 2, 1}` donne `{5, 4, 4}`
- le produit terme à terme d'une liste avec un réel est obtenu avec `*` entre le réel et la liste :
  `2*{1, 2, 3}` donne `{2, 4, 6}`
- la somme terme à terme d'une liste avec un réel est obtenue avec `+` entre le réel et la liste :
  `5+{1, 2, 3}` donne `{6, 7, 8}`
- la puissance d'une liste avec un réel est obtenue avec `^` entre la liste et le réel :
  `{1, 2, 3}^2` donne `{1, 4, 9}`
- la puissance d'un réel avec une liste est obtenue avec `^` entre le réel et la liste :
  `5^{1, 2, 3}` donne `{5, 25, 125}`

6.2.14 Calcul du PGCD par l'algorithme d'Euclide

-Version itérative procédurale

```
pgcd[x_,y_] :=Module[{a=x,b=y,r},While[b>0,r=Mod[a,b] ;a=b ;b=r] ;a]
```

-Version récursive procédurale

```
pgcd2[a_,b_] :=If[b==0,a,pgcd2[b,Mod[a,b]]]
```

-Version récursive déclarative

```
pgcd3[a_,0] :=a ;
pgcd3[a_,b_] :=pgcd3[b,Mod[a,b]]
```

Chapitre 7

Algorithmique et Maxima

7.1 Installation

Site home : `http://www.ma.utexas.edu/users/wfs/maxima.html`

Installation Linux : binaires au format RPM (distributions Red Hat, Mandrake, ...)
`http://rpmfind.net/linux/RPM/sourceforge/symaxx/maxima-5.6-1.i386.html`
Au format DEB (distribution Debian)
`ftp://ftp.lip6.fr/pub/linux/distributions/debian/pool/main/m/maxima/maxima_5.6-5_i386.deb`
Compilation depuis le source
Il faut d'abord installer le compilateur Lisp GCL, disponible sur :
`ftp://ftp.ma.utexas.edu/pub/gcl`
puis maxima 5.6 disponible sur :
`ftp://ftp.ma.utexas.edu/pub/maxima/`

Installation Windows, récuperez et exécutez le programme d'installation :
`http://www.ma.utexas.edu/users/wfs/maxima-download/maxima55l-setup.exe`

7.2 Présentation

`Maxima` est un logiciel de calcul formel généraliste : il permet de manipuler de manière exacte des expressions symboliques : par exemple développer, factoriser une expression, dériver, trouver une primitive, calculer une limite, résoudre une équation de manière exacte si c'est possible ou de manière approchée sinon. Il sait tracer des graphes de fonctions d'une variable, mais aussi de fonctions de 2 variables. Il peut être utilisé dans le supérieur aussi bien que dans le secondaire.

Ce logiciel est développé depuis les années 1970, il est devenu récemment un logiciel libre et est maintenu par William F. Schelter (dont l'adresse est `wfs@math.utexas.edu`).

7.2.1 Lancement

Tapez `xmaxima` dans une fenêtre de commandes. La fenêtre `xmaxima` se décompose en 4 parties de haut en bas :

- une barre de menu (`file`, `edit`, `help`)
- la feuille de calcul qui contient les commandes et les résultats de `Maxima`

- une barre de boutons de navigation de l'aide de `Maxima`
- l'aide de `Maxima`

7.2.2 Exemples

Voici quelques exemples de commandes de `Maxima` :
- développer une expression : `ratsimp((1+x)^10);`
- factoriser une expression sur les entiers : `factor(x^100-1);`
- dériver une fonction : `diff(sin(x^2),x);`
- trouver une primitive : `integrate(1/(x^4+1),x);`
- calculer la limite d'une fonction en un point :
  `limit(sin(x)/x,x,0);`
- résoudre une équation polynomiale : `solve(x^5+x+1=0);`
- résoudre un système d'équations linéaires :
  `solve([x+y+z=5,3*x-5*y=10,y+2*z=3],[x,y,z]);`
- tracer le graphe d'une fonction :
  `plot2d(sin(x),[x,0,2*%Pi]);`
- tracer le graphe d'une fonction de 2 variables :
  `plot3d(x^2-y^2,[x,-2,2],[y,-2,2]);`

7.2.3 Forces et faiblesses

Les points forts :
- l'aide en ligne se trouve dans la même fenêtre,
- les graphes de fonctions sont intégrés à la feuille de calcul
- les commandes et les réponses de `Maxima` sont numérotées automatiquement, il est donc très facile de réutiliser un résultat précédent (inutile de le stocker explicitement dans une variable)
- le navigateur de l'aide permet de lancer `netmath` qui devrait pouvoir lancer d'autres logiciels libres ou gratuits plus spécialisés dans d'autres domaines, par exemple `PARI/GP` pour les théoriciens des nombres.

La principale faiblesse de `Maxima` est qu'il ne sait pas bien gérer les extensions algébriques, mais ceci ne devrait pas être gênant avant la maitrise de mathématiques.

7.3 Pour commencer

7.3.1 Comment lancer Maxima ?

Sur un PC, sous Linux, lancer `Maxima` en tapant :
`xmaxima`

7.3.2 Comment éditer et sauver un programme

Sur une ligne d'entrée, on écrit des ordres (ou commandes) `Maxima` qui peuvent être immédiatement exécutées.
Après chaque ordre, terminé par ; et envoyé par la touche `Entrée`, la sortie correspondante est affichée dans une zone de sortie, gérée par `Maxima` et la

réponse numerotée par un label ; une nouvelle zone d'entrée (elle aussi numerotée) est automatiquement créée.
Un ordre, terminé par $ et envoyé par la touche `Entrée` sera éxécuté sans entraîner de sortie.
Pour `Maxima` un programme est une fonction.
On écrira par exemple :

```
ma_procedure(x1,x2) :=(x1+x2)/2 ;
```

ou encore

```
f(x) :=cos(x)-x ;
newton(f,x0) := block([dfx0,numer],
local (df),
numer :true ; define(df(x),diff(f(x),x)),
do (dfx0 :df(x0),
if dfx0=0.0 then diplay("division par zero"),
x0 :x0-f(x0)/dfx0,
if abs(f(x0))<1.1E-6 then return(x0))) ;
```

Après la dernière ligne, on tapera simplement sur `Entrée`.
Le ; entraînera l'affichage du texte de la procedure, qui aura alors été analysé et considéré comme correct, syntaxiquement.

On peut aussi taper son programme dans son éditeur préféré (par exemple emacs) puis on le sauve par exemple sous le nom `essai.mc`
Puis dans `Maxima` il faut taper :
`batch("essai.mc") ;` pour valider le programme.
Si l'on veut traduire un programme écrit en `Maxima` en `Lisp` il faut taper : `translate_file("essai.mc")` et cela crée un fichier `essai.LISP`.Ce fichier peut alors être utiliser dans `Maxima` en tapant : `loadfile("essai.LISP")`

Si le programme pgcdr a été traduit en `Lisp` la commande :
`dispfun(pgcdr)` reécrit ce programme en `Maxima`.

7.3.3 Comment corriger un programme

Pour modifier un programme existant, ou pour corriger une erreur de syntaxe, on revient dans le texte même, à l'endroit voulu (avec un clic de la souris).
On réalise les modifications du texte, et on termine en tapant `Entrée` (sans forcément revenir en fin de texte).
La procédure est de nouveau analysée et affichée (si elle est correcte).
On peut aussi faire les corrections et les améliorations dans l'éditeur, puis refaire la commande `batch(" ") ;`

7.3.4 Comment éxécuter un programme

Les procédures de `Maxima` sont essentiellement des fonctions.
Comme en `MuPAD`, `ma_procedure(5, 13)` est un nombre qui peut être simplement affiché ou utilisé dans d'autres fonctions...

7.3.5 Comment sauver une session de travail

Pour sauver la session au complet il faut taper :
`save("travail.LISP",all)`
pour sauver la valeur d'une variable ou une fonction il faut taper les noms de variables ou de fonctions séparés par des virgules à la place de `all`

7.4 Les différentes instructions

7.4.1 Les commentaires

Les commentaires sont parenthésés par `/*` et `*/`

7.4.2 Les variables

Leurs noms

Ce sont les endroits où l'on peut stocker des valeurs, des nombres, des expressions.
Un nom doit commencer par une lettre, ne pas contenir d'espace, de symbole opératoire (+, -, ...) et ne pas être un mot réservé.
Avec `Maxima` on peut utiliser des noms ayant plus de 8 caractères.

Notion de variables locales

Le `block` permet de définir et aussi d'initialiser des variables locales qui seront locales pour la suite d'instructions située dams le `block`.
La syntaxe est :
`block([v1 :val1,v2 :val2,v3,v4 :val4],inst1,inst2,... instn)`
La valeur du `block` est alors la dernière instruction.
Exemple 1 :

```
block([w:0], for i:1 thru 10 do (w:w+i^2));
```

La réponse est : `DONE`
Si l'on veut obtenir comme réponse la valeur de `w` il faut écrire `w` comme dernière instruction :

```
block([w:0], for i:1 thru 10 do (w:w+i^2),w);
```

Cette fois la réponse est : `385`
Attention à la place des parenthèses car pour :

```
block([w:0], for i:1 thru 10 do (w:w+i^2,w));
```

La réponse est : `DONE`

Exemple 2 :

```
g(x):=block([u:x+3,w], u:u^2, w:(y+2)^2, u+w);
```

La réponse est :

```
g(x) := block([u : x + 3, w], u : u^2 , w : (y + 2)^2 , u + w)
```

Alors :
`g(2)` donne `(y + 2)^2 + 25`

Exemple 3 :

```
h(x):=block([u:x+3,w], u:u^2,
```

```
              if (u<3) then  w:(y+2)^2 else w:(y+2)^2+1,
              u+w);
```

La réponse est :

```
 h(x) :=block([u : x + 3, w], u : u^2 ,
   if (u<3) then w : (y + 2)^2  else  w : (y + 2)^2  + 1, u + w)
```

Alors :

`[h(-2),h(2)]` donne `[(y+2)^2+1,(y+2)^2+26]`

Notion de paramètres

Quand on écrit une fonction il est possible d'utiliser des paramètres. Par exemple si `a` et `b` sont les paramètres de la fonction `pgcd` on écrit :
`pgcd(a,b)`
Ces paramètres se comportent comme des variables locales, la seule différence est qu'ils sont obligatoirement initialisés lors de l'appel de la fonction. L'exécution se fait en demandant par exemple : `pgcd(15,75)`

7.4.3 Les Entrées

`read(string1, ...)` écrit ses arguments puis lit et évalue ce que l'utilisteur a entré.
Par exemple :
`a :read("entrez le nbre d'iterations")`
`readonly (string1,...)` écrit ses arguments puis lit ce que l'utilisateur a entré sans l'évaluer.

7.4.4 Les Sorties

`disp(expr1,expr2, ...)` affiche les valeurs de ses arguments.
`display(expr1, expr2, ...)` affiche les valeurs de ses arguments sous la forme `expr1=valeur(expr1)`....
`ldisplay(expr1, expr2, ...)` affiche comme `display` mais donne aux résultats intermédiaires un label (`E11...`).

7.4.5 La séquence d'instructions ou action

Une action est une séquence d'une ou plusieurs instructions.
Dans un `block` ou dans la définition d'une fonction les différentes instructions sont séparées par des virgules (`,`)
Lorsque l'on veut exécuter une commande il faut terminer cette commande par un point-virgule (`;`) ou par un dollar (`$`) (par un point-virgule (`;`) si on veut avoir en écho la réponse ou par un dollar (`$`) pour ne pas avoir d'écho).

7.4.6 L'instruction d'affectation

L'affectation se traduit par le signe `:` (deux points).
Quand on utilise l'affectation (par exemple `a :2 ;`) `Maxima` évalue le membre de droite et affecte le résultat au membre de gauche.

Exemple :
Pour stocker 2 dans la variable `a` on écrit `a :2 ;`.
Pour stocker la valeur contenue dans `b` dans la variable `a` on écrit `a :b ;`
et pour stocker `b+2` dans `a` on écrit `a :b+2 ;`

Attention!!!
`:=` sert à définir une fonction.
Exemples :

```
f(x) :=2*x+1
```

alors `f(2)` donne 5

```
factorielle(n) :=block([f :1],for i :1 thru n do (f :f*i),f) ;
```

alors
`factorielle(4)` donne 24.

7.4.7 Les instructions conditionnelles

`if` *condition* `then` *expression1*
`if` *condition* `then` *expression1* `else` *expression2*
Exemple :

```
if (a = 10) or (a < b) then b :b-a else a :a-b
```

7.4.8 Les instructions "do"

Il y a trois types d'instructions `do` :
Le premier type est semblable à l'instruction "pour" (cf 7.4.9).
Le deuxième type est semblable à l'instruction "répéter" (cf 7.4.10).
Le troisième type est semblable à l'instruction "tant que" (cf 7.4.11).

7.4.9 Les instructions "Pour"

La syntaxe est :

```
for i :1 step 2 thru 10 do (i1,i2,i3)
```

Exemples :

```
block([w :0], for i :1 thru 10 do (w :w+i*i))
```

réponse : `DONE`

```
block([w :0], for i :1 thru 10 do (w :w+i*i),w)
```

réponse : `385`

```
(x :1,for i from 1 thru 10 do (x :x*i),x) ;
```

réponse `3628800` On remarquera que `step 1` peut être omis.

```
(C1)for a:1 thru 6 step 2 do ldisplay (i);
(E1)          a = 1
(E2]          a = 3
(E2)          a = 5
```

La fonction `ldisplay` donne des labels (ici `E1, E2, E3`) aux réponses intermédiaires.
On peut aussi écrire :

```
for f in [log, atan] do ldisp(f(1)) ;
```

On obtient :

$$\frac{0}{4}\ \%\mathrm{PI}$$

7.4.10 L'instruction "Repeter"

La syntaxe est :

```
for i :1 step 2 unless condition do (i1,i2,i3)
```

Exemples :

```
for p :1 unless p>7 do
(t :diff(t,x)/p,
 s:s+subst(x=0,t)*x^p)
```

On remarquera que `step 1` peut être omis.

```
f(x) :=cos(x)-x ;
newton(f,x0) := block([dfxn,numer],
local (df,xn),
numer :true ; define(df(x),diff(f(x),x)),
for xn :x0 unless abs(f(xn))<1.1E-6 do
(dfx0 :df(xn),
if dfx0=0.0 then diplay("division par zero"),
xn :xn-f(xn)/dfxn)) ;
```

On peut aussi omettre "for...unless condition " en utilisant la fonction `return`

```
f(x) :=cos(x)-x ;
newton(f,x0) := block([dfx0,numer],
local (df),
numer :true ; define(df(x),diff(f(x),x)),
do (dfx0 :df(x0),
if dfx0=0.0 then diplay("division par zero"),
x0 :x0-f(x0)/dfx0,
if abs(f(x0))<1.1E-6 then return(x0))) ;
```

7.4.11 L'instruction "Tant que"

La syntaxe est :

```
for i :1 step 2 while condition do (i1,i2,i3)
```

Exemples :

```
(s :1, for i :1 while i<10 do (s :s+i))
```

On remarquera que `step 1` peut être omis.

```
s:0$
for i:1 while i<=10 do s:s+i;
          DONE
s;
          55
```

Autre forme permise (lorsque l'indice n'apparait pas dans les instructions) :

```
x=20$
```

```
(while x>10 do (x :x-1),x) ;
```
la réponse est : 10
ou encore
```
x=20$
(thru 10 while x>-20 do (x :x-1),x) ;
```
la réponse est : 10 (la boucle while est effectuée au plus 10 fois)

7.4.12 Les conditions ou expressions booléennes

`ou` se traduit par `or`
`et` se traduit par `and`
`non` se traduit par `not`(opérateur préfixé)
`égalité` se traduit par `=`
`différent` se traduit par `#`
`inférieur` se traduit par `<`
`supérieur` se traduit par `>`
`inférieur ou égal` se traduit par `<=`
`supérieur ou égal` se traduit par `>=`
Exemple :
```
x:20;$ a:(if (x >17) then 2 else 4);
```
réponse 4
```
a:(if (x >17) then x:2 else a:4,a+x);
```
réponse 24
```
a:(x:1,for i from 1 thru 10 do (x:x*i));
```
réponse `DONE`

7.4.13 Les fonctions

`:=` sert à définir les fonctions.
Exemples :
`f(x) :=x+2 ; f(2)` donne 4
```
g(a, b) := (if a= 10 or a<b then b :b - a else a :a - b, [a, b]) ;
```
`g(2,5)` donne `[2,3]`
La fonction `return` permet de sortir d'un block en renvoyant son argument.
Exemple :
```
heron(n) :=block([r :2],
for i :1 thru 10 do (r :0.5*(r+n/r),
if abs(r*r-n)<0.000001 then return(r)),
[r,10]) ;
```
`heron(n)` renvoie la valeur approchée de $\sqrt{(n)}$ à 10^{-6} prés si on a fait moins de 10 itérations et sinon renvoie $[U_{10}, 10]$ où $U_0 = 2, U_i = 0.5 * (U_{i-1} + n/U_{i-1})$
.

7.4.14 Les listes

On utilise les [] pour délimiter une liste.
Par exemple [1, 2, 7, x+y] est une liste de 4 éléments.
Voici quelques commandes utiles :

`append(liste1, liste2, ...)` renvoie la concaténation des listes placées en argument (faire `example(append)` ; pour avoir un exemple.
`cons(expr, liste1)` renvoie la liste où on a mis en premier `expr` dans `liste1`.
`endcons(expr, liste1)` renvoie la liste où on a mis en dernier `expr` dans `liste1`.
`first(expr)` renvoie le premier élément de `expr`.
`last(expr)` renvoie le dernier élément de `expr`.
`delete(expr1, expr2)` enlève dans `expr2` toutes les occurrences de `expr1`.
`delete(expr1, expr2, n)` enlève dans `expr2` toutes les `n` premières occurrences de `expr1`.
`length(expr)` renvoie la longueur de l'expression `expr`.
`listp(expr)` renvoie `true` si `expr` est une liste et `false` sinon.
`makelist(expr,var,i1,i2)` renvoie une liste constituée par les différentes valeurs prises par `expr` lorsque `var` varie entre `i1` et `i2`.
Exemples :
`makelist(i*i,i,2,8)` ; renvoie
`[4, 9, 16, 25, 36, 49, 64]`
`makelist(concat(X,i),i,1,6)` renvoie
`[X1,X2,X3,X4,X5,X6]`
`makelist(X=Y,Y,[A,B,C])` renvoie
`[X=A,X=B,X=C]`

`menber(expr, liste)` renvoie `true` ou `false` selon que `expr` est dans `liste` ou pas.
`rest(expr, n)` renvoie `expr` sans ses `n` premiers termes si `n` est positif ou sans ses `n` derniers termes si `n` est negatif.
`reverse(liste)` renvoie la liste dans l'ordre inverse.
On peut aussi définir des matrices par exemple :

```
A :matrix([1,2],[3,4]) ;
```

7.4.15 Calcul du PGCD par l'algorithme d'Euclide

Version itérative

```
pgcd(a,b) :=(local(r),do (if b=0 then return(abs(a)),
r :mod(a,b), a :b,b :r)) ;
pgcd1(a,b) :=block([r],do (if b=0 then return(abs(a)),
r :mod(a,b), a :b,b :r)) ;
pgcd2(a,b) :=(local(r),while b#0 do (r :mod(a,b), a :b,b :r),
abs(a)) ;
pgcd3(a,b) :=block([r],while b#0 do (r :mod(a,b), a :b,b :r),
abs(a)) ;
```

Version récursive

```
pgcd4(a,b) :=block([],if b=0 then return(abs(a)),return(pgcd4(b,mod(a,b)))) ;
```

ou encore

```
pgcd5(a,b) :=if b=0 then abs(a) else pgcd5(b,mod(a,b)) ;
```

Chapitre 8

Algorithmique et MuPAD

8.1 Pour commencer

Attention
`MuPAD` est sensible à la différence entre majuscules et minuscules.

8.1.1 Comment éditer et sauver un programme

Il est commode d'utiliser `emacs` comme interface, et d'avoir deux *Buffers* ouverts : l'un deux sera le texte du programme, l'autre sera une session `*MuPAD*` qui permettra de tester que le programme est correct (il faudra parfois ouvrir un troisième Buffer pour une session de mise au point lorsqu'il se produit une erreur un peu récalcitrante).
Un exemple : Lancez `emacs` puis tapez sur la touche `Echap` puis, tapez `x mupad` < **Return**>
puis `Start MuPAD` du menu `MuPAD` ce qui lance une session `*MuPAD*` comme d'habitude. Dans le menu `Files` sélectionnez `Open File` et donnez, par exemple, comme nom de fichier `prog1.mu`. Dans le menu `Buffers` vous devez avoir les buffers suivants `*MuPAD*` et `prog1.mu`, il suffit de cliquer sur le nom de buffer pour passer de la session `*MuPAD*` à l'édition du programme.

Nous allons écrire une fonction appelée `ma_procedure` qui calculera la moyenne de deux nombres.

Sélectionnez le buffer `prog1.mu`, puis dans le menu `MuPAD` sélectionnez `Shapes` puis `Procedure`. On vous demande dans l'ordre d'entrer le nom de la procédure, par exemple `ma_procedure`, puis les arguments (ou paramètres) qu'il faudra lui passer, par exemple `x1,x2`, puis les options (tapez sur <**Return**> pour ne pas donner d'options).
Votre texte doit ressembler à :

```
ma_procedure:=
proc (x1,x2)
begin

end_proc: /* End of ma_procedure */
```

Il ne reste plus qu'à entrer entre `begin` et `end_proc` :

```
return (x1+x2)/2;
```

et à sauvegarder le fichier (`Save Buffer` du menu `Files`).

Sélectionnez maintenant le buffer `*MuPAD*` et tapez :
`read("prog1.mu");`
ce qui charge le fichier texte qui contient la définition de `ma_procedure`. Vous pouvez maintenant calculer la moyenne des deux entiers 5 et 13 en tapant : `ma_procedure(5,13);` ce qui doit vous renvoyer `9`.

Exemple de traduction LaTeX

```
 tex:=proc (e)
begin
print(Unquoted,generate::TeX(e);
end_proc: /* End of ma_procedure */
```

Sauvez ce programme sous le nom : `tex.mu` et dans votre session `*MuPAD*` tapez :
`read(''tex.mu'') ;`
Ainsi la commande `tex` vous transformera l'écriture de votre résultat en LaTeX.

8.1.2 Comment corriger un programme

Si la syntaxe est mauvaise, la machine vous indique un message d'erreurs. Faites apparaître le texte de votre programme : ouvrir le menu `Buffers` et sélectionner le nom de votre programme (`prog1.mu`).
Vous pouvez corriger votre erreur, puis sauvegarder votre correction dans le fichier `prog1.mu` (`Save Buffer` du menu `Files`).
Sélectionnez maintenant le buffer `*MuPAD*` et tapez :
`read("prog1.mu");`
ce qui charge le fichier texte qui contient la définition de `ma_procedure`.
Si vous ne voyez pas quelle peut être la source d'une erreur, vous pouvez utiliser le débuggueur `mdx`. Le programme `mdx` permet de stopper l'exécution d'un programme `MuPAD` à une ligne donnée et continuer l'exécution ligne après ligne avec la possibilité d'examiner le contenu des variables.

Pour lancer `mdx`, choisissez `Debug` du menu `MuPAD` ou tapez dans la fenêtre `emacs` la touche `Echap` puis `x mdx` puis < **Return**>.
Vous devez voir apparaitre :
`Run mdx (like this): mupad prog1.mu`
Tapez `Entree` (après avoir changé le nom du fichier si nécessaire).

Vous devez voir apparaitre un buffer qui ressemble fortement au buffer `*MuPAD*`. Vous pouvez maintenant exécuter en mode pas à pas n'importe quelle procédure de votre fichier en tapant par exemple :
`debug(ma_procedure(5,13));`
La fenêtre `emacs` doit maintenant être divisée en deux parties, avec le texte source d'une part et le buffer `*gud-prog1.mu*` d'autre part. Les commandes de mise au point doivent être écrites dans le buffer `*gud-prog1.mu*`.
Les principales sont :

- `p` affiche la valeur d'une variable
- `n` exécute la ligne courante du source
- `s` comme `n` mais en exécutant les appels de procédure de la ligne courante en mode pas-à-pas

– `c` continue l'exécution jusqu'au prochain point d'arrêt. Il est possible de mettre un point d'arrêt en mettant dans le texte source, le curseur là où on veut arrêter l'exécution du programme puis en tapant `Ctrl-X Ctrl-A Ctrl-B` dans la fenêtre source (ce qui exécute la commande `S` dans la fenêtre du debuggeur).

Pour en savoir plus sur la programmation, vous pouvez par exemple consulter avec Netscape les notes de B. Ycart et P. Zimmermann :
`http://www.math-info.univ-paris5.fr/Enseignements/demarre_mupad/`

8.1.3 Comment exécuter un programme

Vous sélectionnez maintenant le buffer `*MuPAD*` et tapez :
`read("prog1.mu");`
ce qui charge le fichier texte qui contient la définition de `ma_procedure` (qui calcule par exemple la moyenne de deux nombres cf exemple 8.1.1). Vous pouvez maintenant calculer la moyenne des deux entiers 5 et 13 en tapant :
`ma_procedure(5,13);`
qui doit vous renvoyer `9`.

8.1.4 Comment améliorer puis sauver sous un autre nom un programme

Si vous voulez avoir à la fois l'ancien et le nouveau programme il faut :
faire apparaître le texte de votre programme en ouvrant le menu `Buffers` et sélectionner le nom de votre programme (`prog1.mu`).
Vous pouvez corriger améliorer votre programme (pensez à changer le nom de la procédure (par exemple `ma_procedure2`), puis sauvegarder votre correction dans le fichier `prog2.mu` (par exemple) avec `Save Buffer As...` du menu `Files` : il faut alors donner le nom du nouveau fichier (ici `prog2.mu`).
Sélectionnez maintenant le buffer `*MuPAD*` et tapez :
`read("prog2.mu");`
ce qui charge le fichier texte qui contient la définition de `ma_procedure2`.

8.2 Les différentes instructions

8.2.1 Les commentaires

Il faut prendre l'habitude de commenter les programmes. En algorithmique un commentaire commence par

`//` et se termine par un passage à la ligne.

Traduction MuPAD

un commentaire est entouré de deux `#` ou
commence par `/*` et se termine par `*/`.
ceci est un commentaire
/* ceci est un commentaire */

8.2.2 Les variables

Leurs noms

Ce sont les endroits où l'on peut stocker des valeurs, des nombres, des expressions.
Un nom doit commencer par une lettre, ne pas contenir d'espace, de symbole opératoire (+, -, ...) et ne pas être un mot réservé.
Avec `MuPAD` on peut utiliser des noms ayant même plus de 8 caractères.

Notion de variables locales

Pour `MuPAD`, il faut définir les variables locales en début de programme en écrivant :
`local a,b`

Notion de paramètres

Quand on écrit une fonction il est possible d'utiliser des paramètres.
Par exemple si `A` et `B` sont les paramètres de la fonction `PGCD` on écrit :
`fonction PGCD(A,B)....ffonction`

Traduction MuPAD

```
PGCD :=proc(A,B)
begin
....
end_proc :
```

Ces paramètres se comportent comme des variables locales, la seule différence est qu'ils sont initialisés lors de l'appel de la fonction. L'exécution se fait en demandant par exemple : `PGCD(15,75)`

8.2.3 Les Entrées

Pour que l'utilisateur puisse entrer une valeur dans la variable `A` au cours de l'exécution d'un programme, on écrira, en algorithmique :
`saisir A`
Et pour entrer des valeurs dans `A` et `B` on écrira :
`saisir A,B`

Traduction MuPAD

```
input(''A='',A)
input(''A='',A,''B='',B )
```
mais la plupart des entrées se font par passage de paramètres.

8.2.4 Les Sorties

En algorithmique on écrit :
`Afficher "A=",A`

Traduction MuPAD

`print(''A='',A)`
Il fait savoir que lorsqu'une procèdure est appelée, la suite d'instructions entre `begin` et `end_proc` est ecécutée, et le résultat est égal au résultat de la dernière évaluation.

8.2.5 La séquence d'instructions ou action

Une action est une séquence d'une ou plusieurs instructions.
En langage algorithmique, on utilisera l'espace ou le passage à la ligne pour terminer une instruction.

Traduction MuPAD

Le : ou ; est un séparateur d' instructions.
Le ; génére une sortie alors que le : n'en génére pas.

8.2.6 L'instruction d'affectation

L'affectation est utilisée pour stocker une valeur ou une expression dans une variable.
En algorithmique on écrira par exemple : `2*A->B`
pour stocker `2*A` dans `B`
ATTENTION
Avec `MuPAD` on écrit :
`b :=2*a` pour stocker `2*a` dans `b`.

8.2.7 Les instructions conditionnelles

`Si <`*condition*`> alors <`*action*`> fsi`
`Si <`*condition*`> alors <`*action1*`> sinon <`*action2*`> fsi`
Exemple :
`Si A = 10 ou A < B alors B-A->B sinon A-B->A fsi`

Traduction MuPAD

`if <`*condition*`> then <`*action*`> end_if`
`if <`*condition*`> then <`*action1*`> else <`*action2*`> end_if`
Exemple :
`if a = 10 or A < B then b :=b-a else a :=a-b end_if`
Lorsque il y a plusiuers `if else` à la suite on écrit `elif` au lieu de `else if`.

8.2.8 Les instructions "Pour"

`Pour I de A à B faire <`*action*`> fpour`
`Pour I de A à B (pas P) faire <`*action*`> fpour`

Traduction MuPAD

`for i from a to b do <action> end_for`
`for i from b downto a do <action> end_for`
`for i from a to b step p do <action> end_for`
Vous pouvez aussi ouvrir le menu `MuPAD` sous menu `Shapes` et séléctionner `for`.

8.2.9 L'instruction "Repeter"

`Repeter <action> jusqua < condition>`

Traduction MuPAD

`repeat <action> until < condition> end_repeat`

8.2.10 L'instruction "Tant que"

`Tant que <condition> faire <action> ftantque`

Traduction MuPAD

`while <condition> do <action> end_while`
Vous pouvez aussi ouvrir le menu `MuPAD` sous menu `Shapes` et séléctionner `while`.

8.2.11 Les conditions ou expressions booléennes

Une condition est une fonction qui a comme valeur un booléen, à savoir elle est soit `vraie` soit `fausse`.

Les opérateurs relationnels

Pour exprimer une condition simple on utilise les opérateurs :
`=` `<` `>` $\leq$ $\geq$ $\neq$

Les opérateurs logiques

Pour traduire des conditions complexes, on utilise les opérateurs logiques :
`ou et non`
qui se traduisent sur les calculatrices et en `MuPAD` par :
`or and not`

8.2.12 Les fonctions

Dans une fonction on ne fait pas de saisie de données : on utilise des paramètres qui seront initialisés lors de l'appel.
Dans une fonction on veut pouvoir réutiliser le résultat :
on n'utilise pas la commande `affichage` mais la commande `retourne`.
On écrit par exemple en algorithmique :

```
fonction addition(A,B)
retourne A+B
ffonction
```

Cela signifie que :
-Si on fait exécuter la fonction, ce qui se trouve juste après `retourne` sera affiché. Les instructions qui suivent `retourne` seront ignorées.

Traduction MuPAD

On écrit par exemple en `MuPAD` (`retourne` se traduit par `return`) :

```
addition:=proc(a,b)
begin
return(a+b)
end_proc;
```

REMARQUE : `return` fait sortir immédiatement de la fonction.

8.2.13 Les listes

On utilise les { } pour délimiter une liste.
Attention!!! En algorithmique on a choisit cette notation car c'est celle qui est employée par les calculatrices...à ne pas confondre avec la notion d'ensemble en mathématiques : dans un ensemble l'ordre des éléments n'a pas d'importance mais dans une liste l'ordre est important...
Par exemple {} désigne la liste vide et {1, 2, 3} est une liste de 3 éléments.
`append` sera utilisé pour concaténer 2 listes ou une liste et un élément ou un élément et une liste :
`{1, 2, 3}->TAB`
`append(TAB, 4) ->TAB` (maintenant `TAB` désigne {1, 2, 3, 4}
`TAB[2]` désigne le deuxième élément de `TAB` ici 2.

Traduction MuPAD

Une liste est une suite d'expressions entre un crochet ouvrant `[` et un crochet fermant `]`.
`[ ]` désigne la liste vide.
Exemple :
`l :=[1,2,2,3]`
`nops(l)` renvoie le nombre d'élements de la liste `l`.
`l[1]` ou `op(l,1)` renvoie le premier élement de la liste `l` : les élements sont numérotés de 1 à `nops(l)`.
`op(l)` renvoie `1,2,2,3`
`append(l,4)` ajoute l'élement `4` à la fin de la liste `l`.
De plus, les listes peuvent être concaténées avec le signe `.` (un point), par exemple `[1,2].[2,3]=[1,2,2,3]`.
`Attention` une suite d'expressions entre une accolade ouvrante { et une accolade fermante } désigne un ensemble.
Exemple :
`A :={a,b,c} B :={a,d}` alors `A union B` désigne `{a,b,c,d}`

`A intersect B` désigne `{a}`
`A minus B` désigne `{b,c}`

8.3 Exemple : le PGCD par l'algorithme d'Euclide

Soient A et B deux entiers positifs dont on cherche le $PGCD$.
L'algorithme d'Euclide est basé sur la définition récursive du $PGCD$:

$$\begin{aligned} PGCD(A,0) &= A \\ PGCD(A,B) &= PGCD(B, A \mod B) \; si \; B \neq 0 \end{aligned}$$

où $A \mod B$ désigne le reste de la division euclidienne de A par B.
Voici la déscription de cet algorithme :
on effectue des divisions euclidiennes successives :

$$\begin{aligned} A = &\quad B \times Q_1 + R_1 \quad 0 \leq R_1 < B \\ B = &\quad R_1 \times Q_2 + R_2 \quad 0 \leq R_2 < R_1 \\ R_1 = &\quad R_2 \times Q_3 + R_3 \quad 0 \leq R_3 < R_2 \\ &..... \end{aligned}$$

Après un nombre fini d'étapes, il existe un entier n tel que : $R_n = 0$.
on a alors : $PGCD(A,B) = PGCD(B,R_1) = ... = PGCD(R_{n-1},R_n) = PGCD(R_{n-1},0) = R_{n-1}$

8.3.1 Traduction algorithmique

-Version itérative
Si B $\neq$ 0 on calcule R=A mod B, puis avec B dans le rôle de A (en mettant B dans A) et R dans le rôle de B (en mettant R dans B) on recommence jusqu'à ce que B=0, le PGCD est alors A.

```
Fonction PGCD(A,B)
Local R
tant que B ≠ 0 faire
  A mod B->R
  B->A
  R->B
ftantque
r\'esultat A
ffonction
```

-Version récursive
On écrit simplement la définition récursive vue plus haut.

```
Fonction PGCD(A,B)
Si B ≠ 0 alors
  r\'esultat PGCD(B,A mod B)
  sinon
  r\'esultat A
fsi
ffonction
```

8.3.2 Traduction MuPAD

-Version itérative :

```
pgcd:=proc(a,b)
local r:
begin
While (b>0) do
r:=a mod b:
a:=b:
b:=r:
end_while:
return(a):
end_proc;
```

-Version récursive :

```
pgcd:=proc(a,b)
begin
if (b=0) then
 return(a)
 else
return(pgcd(b,a mod b)):
end_if:
end_proc;
```

Chapitre 9

Algorithmique, Pascal, C++ et calcul formel

9.1 Aspects non mathématiques

9.1.1 Où trouver un compilateur.

Le mieux est d'utiliser les compilateurs libres du projet GNU. Ils peuvent être téléchargés à partir du site `www.gnu.org`, ou plus simplement :

- sous Linux, vous avez toujours un compilateur `C++` sur le CD-ROM de votre distribution. Il faut juste vérifier qu'il est installé, il vous faut des packages nommés `gcc`, `gcc-c++`, `glibc`, `glibc-devel`, `libstdc++`, `libstdc++-devel`. Il existe aussi des compilateurs `Pascal` (`gpc` et `Free Pascal Compiler` par exemple)
- sous Windows, procurez-vous `CYGWIN`, sur `sourceware.redhat.com`

Il existe aussi des environnements de développement `Pascal` et `C++` commerciaux.

9.1.2 Comment éditer, corriger et sauver un programme ?

On utilise un éditeur de texte par exemple :
`emacs, notepade, nedit`
Attention, un éditeur de texte n'est pas un traitement de texte : il ne faut pas par exemple utiliser `Word` qui rajoute des informations de mises en page incompréhensibles pour le compilateur.
On tape donc le programme à l'aide de son éditeur de texte préféré et on le sauve sous un nom de fichier ayant comme extention `.pas` pour les programmes `Pascal` et `.cc` pour les programmes `C++`. Pour sauver, on utilise bien sûr la commande de sauvegarde de l'éditeur.
Pour corriger une erreur, il suffit de rééditer le programme. L'éditeur permet aussi de sauver sous un autre nom ce qui permet facilement d'avoir deux versions d'un programme ou de réutiliser un morceau de programme.

La forme d'un programme Pascal

```
program <idendificateur> ;
partie inclusion de fichiers (uses ...)
directives de compilation
partie declaration de variables et de fonctions
begin
partie instructions
end.
```

La partie déclaration consiste en la déclaration des variables et la définition des fonctions utilisées dans le programme.
Par exemple on déclare dans ce qui suit deux variables entières (x et y) et une fonction (la fonction pgcd) :

```
var x,y:integer;
pgcd(a,b:integer):integer;
var r:integer;
begin
while (b<>0)
  begin
   r:=a mod b;
   a:=b;
   b:=r
 end;
pgcd:=a;
end;
```

La forme d'un programme C++ :

```
/* commentaires indiquant ce que fait le programme*/
partie inclusion de fichiers (#include <iostream>)
partie declaration de variables et de fonctions
int main(){
partie instructions ;
return(0) ;
}
```

La partie déclaration consiste en la déclaration des variables et la définition des fonctions utilisées dans le programme.
Par exemple on déclare dans ce qui suit deux variables entières (x et y) et une fonction (la fonction pgcd) :

```
int x,y;
int pgcd(int a,int b){
  int r;
  while (b!=0){
    r=a%b;
    a=b;
    b=r;
  }
  return(a);
}
```

9.1.3 Comment compiler un programme

Il existe des environnements de développement intégré pour `Pascal` et `C++`, dans ces environnements on peut utiliser la commande de compilation qui se trouve dans un des menus. Sous Unix ou sous Windows avec les outils de développement GNU, on tape en ligne de commande :

```
gpc essai.pas
```

ou :

```
g++ essai.cc
```

Si le programme est syntaxiquement correct, on obtient un fichier exécutable appelé `a.out` sous Unix et `a.exe` sous Windows.

On peut utiliser des options pour la compilation séparée (`-c`), pour renommer le fichier exécutable (`-o nomdufichierexecutable`) et pour inclure des informations de mise au point (`-g`), ce qui permet de localiser plus facilement les erreurs d'exécution avec un déboggueur comme `gdb`.

Avec certains éditeurs de texte (par exemple `emacs`), on peut automatiser la compilation. Dans l'exemple d'`emacs`, on écrirait en première ligne de `essai.cc` :

```
// -*- compile-command: "g++ -g essai.cc" -*-
```

9.1.4 Comment exécuter un programme

Sous Unix, taper dans la fenêtre de commandes `./a.out`.
Sous Windows, on tape `./a.exe`.

9.1.5 Remarques

Les compilateurs `C++` font toujours la différence entre les majuscules et les minuscules. Pour `Pascal`, cela dépend des compilateurs.

Lorsqu'on écrit une fonction en `Pascal` ou en `C++` on ne peut pas l'utiliser directement, on doit aussi écrire un programme (qu'il faudra compiler) qui utilise cette fonction (programme qui affiche par exemple la valeur de cette fonction).

En `Pascal et C++` on peut compiler des fonctions seules puis les réutilser dans différents programmes à condition de mettre au début du programme une inclusion de fichiers.
Exemple :

- En `Pascal` :
  `uses <`*`nomdufichier`*`>`
- En `C++` :
  `#include <`*`nomdufichier`*`>` si ce fichier se trouve dans la bibliothèque standard.
  `#include "`*`nomdufichier.h`*`"` si ce fichier se trouve dans le même répertoire que le programme (en général vos propres fichiers). Notez que `nomdufichier.h` contient uniquement les déclarations des fonctions, il faut alors compiler `nomdufichier.cc` avec l'option `-c`, et rajouter `nomdufichier.o` dans la commande de compilation du programme utilisant `nomdufichier.h`.

9.2 Les différentes instructions

9.2.1 Les commentaires

En Pascal

Les commentaires sont parenthésés par `(*` et `*)` ou par { et }

En C++

Les commentaires sont parenthésés par `/*` et `*/` ou débutent par `//` et se terminent à la fin de la ligne.

9.2.2 Les variables

Contrairement aux langages des programmes de calcul formel, les variables en `C++` et en `Pascal` sont typées. On doit donc savoir à l'avance si une variable contient un réel, un entier, une chaine de caractères, etc. L'inconvénient est une certaine perte de souplesse, l'avantage est que le compilateur détectera plus facilement des erreurs de typage.

Leurs noms

En Pascal

On peut utiliser des noms ayant jusqu'à 8 caractères, parfois plus selon les compilateurs.

En C++

On peut utiliser des noms ayant jusqu'à 32 caractères dans tous les cas, parfois plus selon les compilateurs.

Notion de variables locales

En Pascal

Il faut définir les variables locales au début du programme (ou au début d'une procèdure (ou fonction) en écrivant par exemple :
`var a,b :real ;`
La syntaxe est :
`var <identificateurdevariable> :<identificateurdetype> ;`

En C++

Il faut définir les variables locales à l'intérieur du bloc définissant la fonction en écrivant par exemple :
`float a,b ;`
La syntaxe est :
`<identificateurdetype> <identificateurdevariable>`
Remarque : en `C++` on peut déclarer les variables au moment où on en a besoin : leur espace de validité commence là où elles sont déclarées jusqu'à

la fin du bloc.
Par exemple dans :
```
for (int i=0 ;i<s ;i++)
      s=s+i ;
```
la durée de vie de `i` est la boucle `for`.

Notion de paramètres

En Pascal

Les procédures ou fonctions peuvent avoir des paramètres : ils se comportent comme des variables locales et sont initialisés lors de l'appel de la procédure (ou fonction).
Par exemple :
l'entête d'une procédure sera : `addition(a,b :integer) ;`
l'entête d'une fonction sera : `addition(a,b :integer) :integer ;`

En C++

Les procédures ou fonctions peuvent avoir des paramètres : ils se comportent comme des variables locales et sont initialisés lors de l'appel de la procédure (ou fonction).
Par exemple :
l'entête d'une procédure sera : `void addition(int a,int b) ;`
l'entête d'une fonction sera : `int addition(int a,int b) ;`

9.2.3 Les Entrées

En Pascal

On écrit pour demander l'entrée de `n` :
```
readln(n)
```

En C++

Il faut tout d'abord inclure le fichier `iostream (#include <iostream>)`.
on écrit alors pour demander l'entrée de `n` :
`cin` ≫ `n`;

9.2.4 Les Sorties

En Pascal

On écrit pour demander l'affichage de `n` :
`writeln('n=',n)` si on veut faire passer le curseur au début de la ligne suivante après l'écriture de `n` ou
`write('n=',n)` si on ne veut pas passer à la ligne.

En C++

Il faut tout d'abord inclure le fichier `iostream (#include <iostream>)`.
on écrit alors pour demander l'affichage de `n` :
`cout ≪ n ≪ endl;`
`endl` signifie que le curseur sera mis au début de la ligne suivante.

9.2.5 La séquence d'instructions ou action

En Pascal

Le ; est un séparateur d'instructions.
Une instuction composée est un groupe d'instuctions parenthèsées par :
`begin end`

En C++

Le ; termine toutes les instructions même la dernière !
Une instuction composée est un groupe d'instuctions parenthèsées par :
`{ }`.
Il est inutile de mettre ; après `}` sauf après la définition d'une `struct`, d'une `class` ou l'initialisation d'un tableau.

9.2.6 L'instruction d'affectation

En Pascal

L'affectation se traduit par `:=`
Exemple : Pour stocker 2 dans la variable `a` on écrit `a :=2`
Pour stocker la valeur contenue dans `b` dans la variable `a` on écrit `a :=b`

En C++

L'affectation se traduit par `=`
Exemple : Pour stocker 2 dans la variable `a` on écrit `a=2 ;`
Pour stocker la valeur contenue dans `b` dans la variable `a` on écrit `a=b ;`

9.2.7 Les instructions conditionnelles

En Pascal

`If` *condition* `then {`*action*`}`
`if` *condition* `then {`*action1*`} else {`*action2*`}`
Exemple :
`if (a = 10) or (a < b) then b :=b-a else a :=a-b`

En C++

`If` *(condition)* `{`*action* `;}`
`if` *(condition)* `{`*action1* `;} else {`*action2* `;}`

Exemple :
```
if ((a == 10) || (a < b)) {b=b-a ;} else {a=a-b ;}
```

9.2.8 Les instructions "Pour"

En Pascal

```
For i :=a to b do action
For i :=b downto a do action
```

En C++

```
for (int i=a ;i<=b ;i++){action ;}
for (int i=b ;i>= a ;i--){action ;}
```

9.2.9 L'instruction "Repeter"

En Pascal

```
repeat action until  (condition)
```

En C++

```
do {action ;} while  (condition) ;
```

9.2.10 L'instruction "Tant que"

En Pascal

```
While (condition) do action
```

En C++

```
while (condition) {action ;}
```
On peut aussi utiliser l'instruction `for` pour traduire un `tant que` :
```
for ( ;condition ; ) {action ;}
```

Remarques
En `C++` on peut toujours interrompre une boucle grâce à l'instruction :
```
break ;
```
Exemples
Schéma `tant que` :
```
for ( ; ; ) { if  !(condition)  {break ;} action ;}
```
Schéma `repeter` :
```
for ( ; ; ) { action ; if  (condition)  {break ;} }
```
On peut aussi passer à l'itération suivante grâce à l'instruction :
```
continue
```
Exemple
```
s=0 ;
for (int i=1 ;i<= 11 ;i++) { if (i==6) { continue ;} ; s=s+i ; }
```
On obtient alors `s=60` car pour `i=6` l'instruction `s=s+i` n'est pas effectuée.

9.2.11 Les conditions ou expressions booléennes

En Pascal

`ou` se traduit par `or`
`et` se traduit par `and`
`non` se traduit par `not`
`égalité` se traduit par `=`
`différent` se traduit par `<>`
`inférieur ou égal` se traduit par `<=`
`supérieur ou égal` se traduit par `>=`

En C++

`ou` se traduit par `||`
`et` se traduit par `&&`
`non` se traduit par `!`
`égalité` se traduit par `==`
`différent` se traduit par `!=`
`inférieur ou égal` se traduit par `<=`
`supérieur ou égal` se traduit par `>=`

9.2.12 Les listes

En Pascal

On utilise des tableaux. Contrairement aux listes, on doit donner dans la déclaration du tableau sa dimension par exemple :
`var T : array[5..20] of integer`
Dans cet exemple les indices de `T` vont de 5 à 20 et `T[10]` désigne l' élément de `T` d'indice 10.

En C++

On utilise le type `vector`. Ce type n'est pas un type de base, il faut donc écrire au début de votre programme :

```
#include <vector>
vector<int> v;
vector<int> w(15);
```

Dans cet exemple `v` est un vecteur vide et `w` est un vecteur de 15 éléments d'indices allant de 0 à 14.
`w[10]` désigne l' élément de `w` d'indice 10.
`int s=w.size() ;` met dans `s` la taille du vecteur `w` (ici 15).
`w.push_back(55) ;` rajoute un élément (55 ici) au vecteur `w`.
`w.pop_back() ;` enlève le dernier élément au vecteur `w`.
`w.insert(w.begin()+1,55)` insère un élément (55 ici) avant l'élément d'indice 1 (donc avant le 2ième élément).
`w.erase(w.begin()+2)` enlève l'élément d'indice 2 (donc le 3ième élément).

9.3 Exemple : le PGCD par l'algorithme d'Euclide

En Pascal

-Version itérative

```
program pgcd;
var x,y : integer;
function pgcd(a,b:integer):integer;
  var r: integer;
  begin
  while b <> 0 do
    begin
    r:=a mod b;
    a:=b;
    b:=r
    end;
  pgcd:=a;
  end;
begin
readln(x,y);
writeln('le pgcd est',pgcd(x,y))
end.
```

-Version récursive

```
program pgcd;
var x,y : integer;
function pgcd(a,b:integer):integer;
  var r: integer;
  begin
  if  b <> 0 do
    pgcd:=pgcd(b,a mod b)
  else
    pgcd:=a;
  end;
begin
readln(x,y);
writeln('le pgcd est',pgcd(x,y))
end.
```

En C++

-Version itérative

```
#include <iostream>
int pgcd(int a,int b){
  int r;
  while (b!=0){
    r=a%b;
    a=b;
    b=r;
```

```
  }
  return(a);
}
int main(){
  int x,y;
  cin>>x>>y;
  cout<<pgcd(x,y)<<endl;
  return(0);
}
```

-Version récursive

```
#include <iostream>
int pgcd(int a,int b){
  if (b!=0)
    return(pgcd(b,a%b);
  else
    return(a);
}
int main(){
  int x,y;
  cin>>x>>y;
  cout<<pgcd(x,y)<<endl;
  return(0);
}
```

9.4 Programmer en C++ avec du calcul formel

Il existe actuellement deux projets pour pouvoir programmer en C++ en utilisant des types symboliques : GiNaC (`http://www.ginac.de`) et Giac : pour installer giac, récupérez le fichier
`ftp://fourier.ujf-grenoble.fr/pub/hp48/giac.tgz`
puis :

```
tar xvfz giac.tgz
cd giac-0.2.2
./configure
make
make install
```

(la dernière étape nécessite d'être `root`).

9.4.1 La classe entier.

Mettre en en-tête du programme :

```
#include <giac/gaussint.h>
using namespace giac;
```

Ensuite `entier` s'utilise comme le type de base `int` dans presque toutes les situations. Attention l'opérateur de division `/` n'est pas défini, utilisez `iquo` pour le quotient euclidien de deux entiers.

Pour compiler votre programme, vous devrez rajouter `-lgmp -lgiac` en fin de ligne de commande.

Pour voir la valeur d'un `entier` avec le déboggueur, créez un fichier `.gdbinit` contenant :

```
echo Defining v as print command for giac types\n
define v
print $arg0.dbgprint()
end
```

L'impression se fait par la méthode `dbgprint()` de `$arg0`.

9.4.2 Les objets symboliques

La classe entier de giac permet aussi de stocker des objets non entiers : par exemple des expressions :

```
#include <iostream>
#include <giac/gaussint.h>
#include <giac/sym2poly.h> // pour factor
using namespace giac;

int main(){
  entier e(string("x^2-1"));
  cout << "La factorisation de x^2-1 est: " << factor(e) << endl;
  cout << "Entrez une expression a factoriser ";
  cin >> e;
  cout << factor(e) << endl;
}
```

Sauvez ce texte par exemple sous le nom `essai.cc` et compilez-le :

```
c++ -g essai.cc -lgmp -lgiac
```

puis exécutez-le :

```
./a.out
```

Pour plus de détails, vous pouvez consulter le fichier `doc/classes.tex` de giac et les fichiers d'en-tête.

9.5 xcas : un logiciel de calcul formel écrit en C++

9.5.1 Installation de xcas

Le programme `xcas` est un logiciel libre écrit en C++, (disponible sous licence GPL). La version à jour se récupère sur :
`http` : `//www − fourier.ujf − grenoble.fr/~parisse` ou
`ftp ://fourier.ujf-grenoble.fr/pub/hp48`
où l'on trouve le code source (`giac.tgz`) ou des versions précompilées pour Linux (PC ou ARM) ou Windows (`xcas.zip, xcas_user.tgz, xcas_root.tgz`).

9.5.2 Éditer, sauver, exécuter un programme avec xcas

On édite un programme avec son éditeur préféré : on peut écrire dans un même fichier (par exemple `bidon`) la définition de plusieurs fonctions séparée par des points virgules (;)
On sauve ce fichier, puis dans `xcas` on tape :

`eval(bidon) ;`
cela a pour effet de créer des fichiers ayant le même nom que les fonctions qui sont définies dans `bidon`.
En tapant le nom de la fonction dans `xcas` on peut voir le contenu de cette variable (ici la définition de la fonction).
En reéditant le programme, on peut facilement le corriger, le sauver sous un autre nom etc...
On peut aussi écrire directement le programme dans la ligne de commandes :
- si le programme est faux, il est mis dans l'historique en tant que chaîne de caractères et on peut ainsi le retrouver dans la ligne de commande en cliquant sur `Q`
- si le programme est juste il est conservé dans un fichier de même nom.

9.5.3 Les instructions avec xcas

Les commentaires sont des chaînes de caractères : ils sont donc parenthèsés par " et "

Les variables sont les endroits où l'on peut stocker des valeurs, des nombres, des expressions, des objets.
Le nom des variables est formé par une suite de caractères et commence par une lettre : attention on n'a pas droit aux mots réservés ...ne pas utiliser par exemple la variable `i` dans un `for` car `i` représente le nombre complexe de module 1 et d'angle $\frac{\pi}{2}$.

Un `bloc` est une suite d'instructions séparées par `;` ou `:`
Un `bloc` est parenthèsé par `{` et `}` et commence eventuellement par la déclaration des variables locales (`local...`).
Les variables locales doivent être déclarées au début d'un bloc (mot réservé `local` puis on met les noms des variables séparés par des virgules (`,`). Ces variables locales peuvent être initialisées lors de leur déclaration.
Voici un exemple :

```
idiv2(a,b) := {local q := 0,r := a;
               if (b! = 0) {q := iquo(a,b);r := irem(a,b);} [q,r];}
```

Les paramètres sont utilisés dans la définition de fonctions : ils sont initialisés lors de l'appel de la fonction et se comportent comme des variables locales.

L'affectation se fait avec `:=` (par exemple `a :=2 ;b :=a ;`)

Les entrées se font par passage de paramètres

Les sorties se font en mettant le nom de la variable à afficher (ou entre crochets la liste des variables à afficher séparées par une virgule) comme dernière instruction.

Une action (ou bloc) est une séquence d'une ou plusieurs instructions.
Quand il y a plusieurs instructions il faut les parenthéser avec `{ }` et séparer
les instructions par un point virgule (`;`) ou par deux points (`:`)
Les tests et boucles ont une syntaxe similaire au langage `C++`.
`testif(a,b)` := {`if` ((`a` == 10) `or` (`a` < `b`)) `b` := `b` − `a`; `else a` := `a` − `b`; [`a`, `b`];}

`testfor1(a,b)` := {`local s` := 0; `for` (`j` := `a`; `j` <= `b`; `j`$^{++}$) `s` := `s` + 1/`j`^{2}; `s`;}
`testfor2(a,b)` := {`local s` := 0; `for` (`j` := `b`; `j` >= `a`; `j`$^{--}$) `s` := `s` + 1/`j`^{2}; `s`;}

`testwhile(a,b)` := {`while` ((`a` == 10) `or` (`a` < `b`)) `b` := `b` − `a`; [`a`, `b`];}

9.5.4 Le PGCD avec xcas

- Version itérative
`pgcd(a,b)` := {`local r`; `while` (`b`! = 0) {`r` := `irem(a,b)`; `a` := `b`; `b` := `r`; } `a`;}
-Version récursive
`pgcdr(a,b)` := {`if` (`b` == 0) `a`; `else pgcdr(b,irem(a,b))`;}
ou
`pgcdr(a,b)` := `if` (`b` == 0) `a`; `else pgcdr(b,irem(a,b))`;

Annexe A

Récupérer et installer un logiciel

La description qui suit s'applique au navigateur Netscape. Elle s'applique de manière identique pour d'autres navigateurs à quelques détails près.

Lorsqu'un lien mentionne un logiciel que vous voulez télécharger, maintenez la touche Shift appuyée et cliquez avec la souris sur ce lien. Une fenêtre s'ouvre et vous propose de sauvegarder un fichier sur votre disque dur, changez éventuellement le nom de ce fichier et notez l'emplacement du répertoire où il a été sauvegardé.

La méthode d'installation du logiciel dépend de votre système d'exploitation et du type de fichier téléchargé, que l'on détermine en général par son extension, c'est-à-dire par les lettres (en principe 3) qui suivent le caractère point dans le nom de ce fichier.

Sous Windows, vous récupérerez en général un fichier exécutable `exe` il suffit alors de cliquer sur son icone pour installer le logiciel. Ou alors il s'agira d'une archive, il vous faudra alors un logiciel de décompression pour l'installer (l'un des plus populaires s'appelle Winzip et propose une version à l'essai gratuitement). Une fois désarchivé votre logiciel, vous trouverez dans l'archive un fichier donnant des instructions complémentaires ou vous devrez vous référer au site où vous avez téléchargé l'archive.

Sous Linux, les formats les plus courants sont `rpm`, `deb` et `tgz` (ou `tar.gz`).

Les deux premiers correspondent à des distributions Linux (Red Hat, Mandrake, Suse par exemple pour `rpm`, Debian pour `deb`) ils doivent être installés par `root` (tapez `su` dans une fenêtre de commandes pour passer `root`) en utilisant respectivement les commandes :
`rpm -U nom_de_fichier.rpm` pour Red Hat, Mandrake, Suse
`apt-get install nom_de_fichier_deb` pour Debian
Regardez bien les éventuels messages d'erreurs qui apparaissent. Ils peuvent signaler que vous devez installer d'autres logiciels de votre distribution Linux pour faire fonctionner votre nouveau logiciel. En général ces logiciels non installés se trouvent sur le CD-ROM de votre distribution Linux. Ils s'installent par la même commande, mais vous devrez d'abord vous déplacer dans le répertoire du CD-ROM qui les contient. Pour cela, introduisez le CD-ROM dans le lecteur, attendez quelques secondes, tapez : `ls /mnt/cdrom`

(si rien n'apparait, tapez : `mount /mnt/cdrom` et recommencez).
Ensuite tapez `cd /mnt/cdrom` puis allez dans le répertoire correct (par exemple `cd Mandrake/RPMS` pour une distribution Mandrake) et lancez les commandes d'installation des logiciels manquants (`rpm -U ...` ou `apt ...`). Certaines distributions Linux proposent des interfaces plus ou moins conviviales pour installer des logiciels, par exemple `rpmdrake` pour Mandrake, `dselect` pour Debian...

Les archives `tar.gz` et `tgz` s'installent sur toute distribution Linux. Commencez par en regarder la table des matières par la commande :
`tar tvfz nom_de_fichier.tgz`
afin de déterminer depuis quel répertoire il faut décompacter le fichier ou lisez la documentation du logiciel disponible en général sur le site où vous avez téléchargé l'archive. Si vous voyez apparaitre des chemins tels que `usr/local/bin` ou `usr/bin`, placez vous dans le répertoire racine (`cd /`) puis tapez :
`tar xvfz nom_du_repertoire/nom_de_fichier.tgz`
(si vous avez sauvegardé l'archive dans votre répertoire, vous pouvez utiliser `~votre_nom_de_login` comme `nom_de_repertoire`). Sinon, tapez simplement depuis le répertoire où se trouve l'archive :
`tar xvfz nom_de_fichier.tgz`
placez-vous dans le répertoire créé et lisez les fichiers `README` ou/et `INSTALL` qui s'y trouvent sans doute.
Remarques :

1/ N'oubliez pas de quitter le compte de `root` avant d'utiliser votre logiciel (en tapant simultanément sur les touches Ctrl et D).

2/ les archives `.tar.bz2` se traitent de manière identique, à ceci près qu'il faut enlever le `z` de `tvfz` ou `xvfz` et rajouter
`--use-compress-program bunzip2`, par exemple :
`tar tvf nom_de_fichier.tar.bz2 --use-compress-program bunzip2`

3/ les archives `.zip` se visualisent par la commande :
`unzip -v nom_de_fichier.zip` et se désarchivent par la commande :
`unzip nom_de_fichier.zip`

Sous linux et autres Unix, vous trouverez également des fichiers sources qu'il vous faudra compiler avant de pouvoir utiliser le logiciel correspondant. De nombreux fichiers sources suivent la procédure d'installation du projet GNU. Pour les compiler la procédure est la suivante (après avoir décompressé l'archive) :
`cd nom_d_archive`
`./configure`
`make`
Puis on passe `root` en tapant `su`, puis : `make install-strip`
puis quittez le compte `root` (en tapant simultanément sur les touches `Ctrl` et `D`). Vous pourrez alors utiliser le logiciel nouvellement installé (éventuellement après avoir tapé la commande `rehash` si votre interpréteur de commandes est `tcsh`).

Si cela ne fonctionne pas vous devriez trouver un fichier `README` ou `INSTALL` qui explique la procédure à suivre.

Annexe B

GNU Free Documentation License

Version 1.1, March 2000

Copyright (C) 2000 Free Software Foundation, Inc. 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom : to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation : a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals ; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts : Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with

changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version :

* A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission. * B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five). * C. State on the Title page the name of the publisher of the Modified Version, as the publisher. * D. Preserve all the copyright notices of the Document. * E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices. * F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below. * G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice. * H. Include an unaltered copy of this License. * I. Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence. * J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission. * K. In any section entitled "Acknowledgements" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein. * L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles. * M. Delete any section entitled "Endorsements". Such a section may not be included in the Modified Version. * N. Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties–for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgements", and any sections entitled "Dedications". You must delete all sections entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document. If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their

copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http ://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page :

Copyright (c) YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation ; with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have no Invariant Sections, write "with no Invariant Sections" instead of saying which ones are invariant. If you have no Front-Cover Texts, write "no Front-Cover Texts" instead of "Front-Cover Texts being LIST" ; likewise for Back-Cover Texts.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Table des matières