ATARI ST

LOGICIEL

COMPILATEUR

GFA BASIC 3.0

GFA

EDITIONS MICRO APPLICATION

Frank Ostrowski
Gottfried P. Engels

Compilateur GFA 3.0

Editions Micro Application

MICRO APPLICATION
58, Rue du Faubourg Poissonnière
75010 PARIS

Auteurs : Frank Ostrowski - Gottfried P. Engels
Traduction française assurée par Pascal Haussmann

Collection dirigée par Mr Philippe OLIVIER
Edition réalisée par Frédérique BEAUDONNET

SUPPORT PRODUIT

Seules les personnes retournant la carte client dûment remplie, en incluant bien nom, adresse, nom du produit et numéro de série, seront enregistrées comme client Micro Application et pourront bénéficier du support produit.

Nous rappelons que le support produit est effectué par l'équipe technique de Micro Application.

Les horaires sont :

Du Lundi au Joudi : **14 h 30 à 17 h 30**

Le Vendredi : **14 h 30 à 16 h 30**

Toute personne n'ayant pas retourné chaque carte spécifique à chaque produit se verra refuser tout support.

Sommaire

Chapitre 1

Introduction

1.1. Initiation accélérée

Cette section décrit comment convertir un listing GFA-BASIC 3.0 en un programme. Elle est conçue pour les lecteurs impatients qui veulent pouvoir essayer le compilateur aussi vite que possible.

Avant toute opération, nous vous conseillons cependant vivement de préparer tout d'abord une copie de sécurité de votre disquette originale.

Pour développer un programme, vous avez besoin des fichiers suivants :

GFABASIC.PRG	L'interpréteur GFA-BASIC 3.0
GFA_BCOM.PRG	Le compilateur GFA-BASIC 3.0
GL.PRG	Le linker GFA-BASIC 3.0
GFA3BLIB	La bibliothèque du linker
GFA3BLIB.NDX	Le fichier d'index de la bibliothèque du linker
MENU.PRG	Le Shell de commande du système de développement GFA
*.GFA	Les fichiers source GFA-BASIC 3.0 que vous voulez compiler

Pour convertir un fichier source GFA-BASIC en un programme qui puisse tourner sans l'interpréteur, vous pouvez procéder de la façon suivante :

0 Lancez le programme MENU.PRG.

❷ Choisissez dans le menu "Fichier" l'entrée "Source".

❸ Sélectionnez le fichier GFA à compiler dans la boîte de sélection de fichier qui apparaît alors.

❹ Appuyez sur la touche F10. Le programme BASIC sera alors compilé et linké.

❺ Le programme BASIC se trouve maintenant sur le lecteur de disque sous le nom de TEST.PRG. Vous pouvez le lancer en sélectionnant l'entrée "Test" dans le menu "Fichier" ou bien en quittant le programme de menu et en lançant le programme TEST.PRG.

1.2. Fonction d'un compilateur

Un ordinateur n'est pas en mesure de comprendre directement des instructions en BASIC. C'est pourquoi un programme BASIC doit toujours être traduit en un langage compris de l'ordinateur. Le rôle d'un interpréteur consiste donc à lire un programme ligne par ligne, à traduire chaque ligne en instructions compréhensibles par l'ordinateur, puis à exécuter ces instructions au fur et à mesure.

Cette méthode présente des avantages et des inconvénients. Les deux principaux inconvénients sont que : 1) le programme BASIC ne peut tourner qu'avec l'interpréteur, 2) le programme est traité relativement lentement du fait que toute instruction BASIC doit d'abord être traduite avant d'être exécutée.

Un compilateur est un programme qui traduit en une fois toutes les instructions d'un programme BASIC en instructions compréhensibles de l'ordinateur, mais sans exécuter immédiatement les instructions ainsi traduites, qui sont alors sauvegardées sous forme d'un programme.

Un compilateur est, au sens premier du mot, une personne qui réunit des documents sur un sujet donné. En informatique, un compilateur sert à composer un programme exécutable à partir des différentes instructions composant un programme BASIC. Ce programme exécutable pourra tourner indépendamment de l'interpréteur et sera beaucoup plus rapide car uniquement composé d'instructions déjà traduites.

Le compilateur GFA-BASIC 3.0, qui est décrit dans ce guide, peut traduire uniquement des programmes GFA-BASIC de la version 3.0. Ces programmes doivent avoir été sauvegardés sous l'interpréteur GFA-BASIC 3.0 avec l'instruction "SAVE". Le compilateur ne peut traiter les programmes sauvegardés

avec SAVE,A. Il ne peut pas non plus traduire des programmes de la version GFA-BASIC 2 ni des programmes d'autres dialectes BASIC.

1.3. Instructions ne pouvant être traitées

Une série d'instructions ne peuvent être converties par le compilateur. Il s'agit notamment des instructions servant à afficher des lignes de programme, c'est-à-dire TRON, TROFF et TRACE$.

Le compilateur ne peut pas non plus traiter l'instruction DEFLIST, qui sera donc purement et simplement ignorée. DUMP n'est pas non plus possible car le programme compilé ne connaît plus le nom des variables.

Il n'est plus possible non plus de charger, sauvegarder et afficher le listing du programme sous le programme compilé. Les instructions LOAD, SAVE, PSAVE, LIST et LLIST ne peuvent donc plus être exécutées par un programme compilé.

1.4. Différences de fonctionnement de certaines instructions entre le compilateur et l'interpréteur

L'instruction CHAIN ne fonctionne pas, sous le compilateur, exactement comme sous l'interpréteur. Dans le programme compilé, le nom spécifié est transmis à la fonction AES shel_write et le programme en cours est terminé avec QUIT. Le programme appelé avec CHAIN est alors lancé. Cela fonctionne lors du retour au bureau GEM.

Le programme d'appel n'occupe alors plus de place en mémoire. Le contenu des 128 octets de mémoire à partir de BASEPAGE+128 est transmis au programme appelé comme ligne de commande.

Avant que l'instruction FILESELECT ne soit appelée, le programme compilé doit comporter encore au moins 32500 octets libres.

1.5. Travail avec le programme MENU

On appelle Shell un programme permettant d'appeler plusieurs programmes de façon très simple, en transmettant des paramètres à ces programmes. A la fin d'un programme qui a été appelé avec Shell, l'utilisateur se retrouve sous Shell.

Le Shell correspondant au compilateur GFA-BASIC 3.0, dont le texte source figure sur la disquette, permet d'appeler l'interpréteur, le compilateur et le linker, ainsi que de sélectionner les options du compilateur et du linker.

1.5.1. Le menu Fichier

Dans le menu déroulant portant le nom "Fichier", vous trouvez comme première entrée "Source". Lorsque vous appelez cette entrée, une boîte de sélection de fichier apparaît, dans laquelle vous pouvez sélectionner le fichier GFA que vous souhaitez traiter. Cette entrée, comme toutes les autres, peut aussi être sélectionnée au clavier. Le point "Source" du menu est appelé avec Control+S.

Le point "Compilateur" du menu transmet le fichier GFA défini avec Source au compilateur, qui produit à partir de là un fichier objet appelé TEST.O. Le compilateur peut aussi être appelé avec Control+C.

Le prochain point du menu (Editeur) sert à appeler l'interpréteur, auquel le fichier GFA sélectionné est transmis. Cette fonction peut aussi être activée avec Control+E.

L'entrée "Linker", qui peut être appelée avec Control+L, linke le fichier TEST.O produit par le compilateur et produit à partir de là le programme exécutable TEST.PRG.

Grâce au point de menu RCS, le programme RCS2 peut être appelé.

Le point de menu Tester (Control+T) permet de lancer le programme compilé.

L'option Exécuter permet de lancer n'importe quel programme choisi avec le sélecteur de fichiers.

L'option Quitter (Control+Q) permet de sortir du menu.

La touche F10 appelle successivement le compilateur et le linker. C'est donc une abréviation pour Control+C et Control+L.

1.5.2. Le menu "Options"

Le menu déroulant suivant comporte les options du compilateur et du linker. Ces options seront décrites dans les prochains chapitres. La table suivante vous présente la liste de ces options. En regard de chaque entrée du menu sont indiquées la combinaison de touches correspondante et le code Shell permettant de sélectionner chaque option.

Entrée	Combinaison de touches	Option
Interruptions	Alternate+I	I+
Select	Alternate+S	S& ou S< ou les deux
Fonctions	Alternate+F	F<
Procedures	Alternate+P	P
IntDiv	Alternate+/	%3
IntMul	Alternate+*	*&
Erreur	Alternate+E	E$
Mémoire	Alternate+M	mxxxx
DebugSym	Alternate+D	-s

En appuyant sur la combinaison de touches correspondante ou en sélectionnant l'entrée voulue, vous activez ou désactivez les options. Si vous choisissez "Mémoire", on vous demande combien de place mémoire le programme doit occuper.

Toutes ces options, sauf DebugSym, concernent le compilateur. Les entrées pour lesquelles plusieurs possibilités sont indiquées dans la table agissent sur deux options. En appelant une fois ces points du menu, on appelle la première option, en les appelant deux fois on sélectionne la seconde option, et en les appelant trois fois on sélectionne les deux.

Les options actuellement fixées sont incrustées dans le coin supérieur gauche de l'écran. Lors du travail du compilateur, le numéro de la ligne actuellement en cours de traduction est affiché dans la ligne des menus déroulants.

1.5.3. Le menu "Sélection"

Le dernier menu permet de fixer un certain nombre de paramètres que nous allons décrire ci-dessous.

Le paramètre G3WAIT détermine s'il faut ou non attendre qu'une touche soit actionnée avant d'ouvrir un fichier lors de la compilation et du linkage. Les fichiers devant ainsi être ouverts peuvent être par exemple le programme à compiler, le fichier objet, GFA3BLIB.NDX, GFA3BLIB et le fichier PRG.

Le but de cette option est de permettre de changer de disquette pendant que l'ordinateur attend qu'une touche soit actionnée. Cela peut être nécessaire si on ne dispose que d'un seul lecteur de disquette et que tous les fichiers nécessaires ne figurent pas sur cette disquette.

Après que ce point ait été sélectionné, tous les paramètres du menu "Sélection" sont affichés sur l'écran. Cet affichage vous indique si vous avez activé ou désactivé G3WAIT en cliquant ce point du menu.

Le point suivant du menu, "G3MOVE", est également activé et désactivé alternativement lorsqu'on le clique. Lorsque vous avez activé G3MOVE, "G3MOVE=ON" est affiché sur l'écran.

Le fait d'activer G3MOVE a pour effet d'amener le compilateur à économiser le plus possible la mémoire disponible lors de la compilation. Au début de son travail, il charge le code source du programme à traiter dans la mémoire, et commence la compilation, le programme compilé étant également sauvegardé dans la mémoire.

Si vous faites compiler un très grand programme alors que vous disposez de peu de mémoire libre, il peut arriver que le code source et le programme occupent ensemble trop de place pour pouvoir tenir entièrement dans la RAM.

C'est pourquoi, lorsque G3MOVE est activé, le code source de chaque procédure traduite est éliminé au fur et à mesure de la mémoire. Cette compression de la mémoire prend naturellement du temps et il est donc déconseillé d'employer G3MOVE lorsque cela ne s'impose pas.

Lorsque vous sélectionnez l'entrée G3OBJ, vous pouvez modifier le nom prédéfini pour le fichier objet produit par le compilateur (c'est normalement TEST.O). Vous pouvez de même, avec G3PRG, modifier le nom de programme prédéfini que le linker attribue au programme terminé (c'est normalement TEST.PRG).

Avec G3LIB, vous pouvez fixer le nom de la bibliothèque qui devra être utilisée (normalement GFA3BLIB). Ces options peuvent naturellement être utilisées non seulement pour fixer le nom d'un fichier, mais aussi, par exemple, pour fixer les chemins devant correspondre à ces fichiers.

Les paramètres peuvent aussi être fixés à l'aide du clavier, en tapant la lettre à la suite de G3. On aura donc :

W ou **w**	pour G3WAIT
M ou **m**	pour G3MOVE
O ou **o**	pour G3OBJ
P ou **p**	pour G3PRG
L ou **l**	pour G3LIB

La définition actuelle des paramètres du menu "Sélection" peut être consultée en appuyant sur la touche Help. L'écran se vide alors et la valeur actuelle de ces paramètres est incrustée sur l'écran. Le fait d'appuyer sur la touche Undo vide l'écran sans faire afficher ces valeurs.

L'option "PRG=GFA", ou la touche F2, commute un mode qui permet de définir le nom du fichier compilé à partir du source GFA, au lieu de TEST.PRG.

L'entrée Externes, ou la touche E, permet d'entrer les noms des fichiers objets qui doivent être liés avec le fichier objet généré par le compilateur, pour obtenir un programme.

1.5.4. Le listing MENU.GFA

Le listing du Shell figure sur la disquette. Les paramètres décrits à la section précédente peuvent également être fixés dans ce listing. Vous trouverez en outre dans la première partie de ce listing les lignes

```
gfaint$="GFABASIC.PRG"

gfacom$="GFA_BCOM.PRG"
gfalnk$="GL.PRG"
```

dans lesquelles sont fixés les noms prédéfinis de l'interpréteur, du compilateur et du linker. Vous pouvez bien sûr modifier ces lignes.

Les paramètres G3OBJ, G3PRG et G3LIB tirent leurs valeurs prédéfinies des lignes

```
tobj$="G3OBJ=TEST.O"
tprg$="G3PRG=TEST.PRG"
tlib$="G3LIB=GFA3BLIB"
```

Les paramètres G3WAIT et G3MOVE sont prédéfinis dans le listing Shell avec

```
t_wait$="G3WAIT=ON"
t_move$="G3MOVE=ON"
```

Vous pouvez activer ces options en éliminant le trait souligné du nom de la variable. Les lignes

```
coi&=0          !no I
cos&=3          !S& et s<
cof&=0          !F<
cod&=0          !no %3
com&=0          !no *&
coe&=0          !no E
cop&=0          !no P
dbsym&=0        !no -s
```

fixent les options prédéfinies pour le compilateur et le linker.

Dans la ligne INLINE irq%,&HD6 est logée une routine d'interruption qui affiche, au cours de la compilation, le numéro de la ligne actuellement compilée. Si vous voulez sauvegarder le Shell sous forme d'un fichier LST, vous devez donc sauvegarder cette zone INLINE séparément (en allant sur la ligne INLINE, en appuyant sur Help, puis en sauvegardant comme MENU.INL sur votre disquette). La ligne

```
{irq%+2}=V:a&
```

définit une variable 16 bits qui sera affichée dans l'interruption.

Avec les lignes

```
BYTE{irq%+6}=32+35
BYTE{irq%+7}=32+0
```

la colonne 35 est fixée comme position X et la ligne 0 comme position Y de la sortie de texte de la routine d'interruption.

1.6. Le travail avec les Shells DOS (non fourni)

Vous pouvez contrôler le compilateur et le linker non seulement avec Shell, mais aussi avec un Shell DOS. La présente section décrit le contrôle à travers un Shell DOS ou un outil semblable.

Les paramètres pour le compilateur peuvent être transmis sous la forme avec laquelle ils apparaissent dans l'affichage de MENU, par exemple

```
gfa_bcom nom S& %3
```

Il en va de même pour les paramètres du linker. La ligne

```
gl -s
```

permettra par exemple d'obtenir un linkage avec production d'une table de symboles. Pour linker un fichier appelé c_tst, vous pourrez spécifier

```
gl c_tst
```

Dans ce cas, test.o, c_tst.o et gfa3blib seront linkés. Le symbole # (dièse) dans la liste des paramètres a pour effet d'empêcher que le fichier test.o soit également linké. La ligne

```
gl # c c_tst
```

linkerait les fichiers c.o, c_tst.o et gfa3blib, mais pas test.o. Pour utiliser une autre bibliothèque que gfa3blib, vous pouvez en spécifier le nom en le faisant précéder d'un signe plus.

La ligne

```
gl +nou_lib
```

aurait par exemple pour effet de faire linker test.o et nou_lib. Le linker interprète comme le nom d'un fichier objet à linker tout paramètre qui n'est pas précédé d'un signe plus et qui n'est pas non plus -s ni #.

Chapitre 2

Le compilateur

2.1. Contrôle des index de tableau

Lorsque vous voulez utiliser des tableaux (arrays) dans vos programmes, vous devez les mettre en place avec DIM. Il vous faut à cette occasion fixer l'index maximal du tableau. Au cours de l'exécution du programme, l'interpréteur GFA-BASIC examine si une valeur trop élevée n'est pas spécifiée comme index de champ, auquel cas l'erreur 16 (index de tableau trop élevé) est déclenchée. Cela vaut également pour le compilateur de GFA-BASIC 2.0.

Ce contrôle n'est pas effectué dans un programme produit avec le compilateur 3.0. Si une valeur trop élevée apparaît comme index de tableau, aucun message d'erreur n'est provoqué. L'absence de contrôle de l'index de tableau permet de rendre les programmes plus courts et plus rapides.

L'exemple suivant aura pour objet de montrer quel effet peut avoir l'absence de contrôle de l'index de tableau. Le programme écrit sous l'interpréteur les nombres 0 et 5 sur l'écran, et déclenche alors le message "Index de tableau trop élevé". Le message d'erreur est déclenché parce que le dimensionnement fixe l'index maximal du tableau à 5, alors que la boucle FOR tente d'appeler l'élément x(6) du tableau.

```
DIM x(5)
FOR i=0 TO 6
  x(i)=i
  PRINT x(i)
NEXT i
```

Si vous compilez et lancez ce programme, il écrira les nombres de 0 à 6 sur l'écran et aucun message d'erreur n'apparaîtra. Le programme se comporte donc comme si le tableau x() avait été dimensionné suffisamment grand dans la première ligne du programme. Le comportement du programme compilé ne trahit donc nullement la présence d'une erreur grave.

Cet exemple soulève la question de savoir ce que fait le programme compilé lorsqu'un index de tableau trop élevé est utilisé. L'instruction DIM réserve en effet de la place mémoire pour le nombre spécifié d'éléments du tableau. L'adresse de chaque élément du tableau est calculée à partir de l'index du tableau, sans que le programmeur ait à s'en préoccuper.

Dans un programme compilé, il en sera ainsi même si l'index du tableau est trop élevé. L'adresse ainsi calculée se situe cependant en dehors de la zone de mémoire qui a été réservée pour le tableau. Si l'index du tableau est trop élevé, des données situées en dehors de la zone prévue pour ce tableau seront donc effacées.

Dans la zone de mémoire ainsi effacée figurent généralement des données nécessaires à une exécution sans erreur du reste du programme. Si ces données sont effacées, le programme risque de produire des résultats erronés, voire d'être planté.

Nous vous avons présenté jusqu'ici les inconvénients de l'absence de contrôle de l'index. Il convient toutefois de noter qu'un index de champ trop élevé ne doit pas normalement apparaître dans un programme dont le développement est achevé, car le programmeur doit avoir éliminé ces erreurs. L'interpréteur sous lequel s'était effectué le développement du programme aura en effet averti le programmeur de ces erreurs en contrôlant systématiquement les index de tableaux.

C'est pourquoi ce contrôle des index devient superflu dans un programme compilé. Il entraînerait une perte de temps et de place. Le compilateur 3.0 est de ce fait quatre à cinq fois plus rapide que le compilateur 2.0 pour les opérations simples avec des éléments de tableau (affectations, addition de deux éléments de tableau, etc.) et dix à onze fois plus rapide que l'interpréteur 3.0.

2.2. Dépassement de valeur entière

Lors de la modification des variables entières, l'interpréteur GFA-BASIC examine si la nouvelle valeur de la variable ne sort pas du domaine de valeurs correspondant à ce type de variable. Voici les intervalles de valeurs autorisées pour les trois types de variables entières :

Type	Suffixe	Minimum	Maximum
Byte	l	0	255
Word	&	- 32768	32767
Long	%	- 2147483648	2147483647

Lorsque l'intervalle numérique autorisé est dépassé, cela déclenche l'erreur 2, 3 ou 4 (suivant le type de variable dont il s'agissaitnombre n'est pas entier, octet ou mot). Cette erreur est ce qu'on appelle un dépassement de valeur entière.

Dans un programme produit avec le compilateur 3.0, on ne contrôle pas systématiquement s'il y a un tel dépassement de valeur entière. Ce contrôle de dépassement est superflu dans un programme sans erreur, alors qu'il occasionne une perte de temps et de place inutile. Il ne présente d'intérêt que lors du développement du programme, qui s'effectue sous l'interpréteur.

Le programme

```
x&=10000
y&=4*x&
PRINT y&
```

provoquera sous l'interpréteur le message d'erreur "Le nombre n'est pas un mot", alors que le programme compilé écrira la valeur -25536 sur l'écran, sans qu'aucun message d'erreur ne soit sorti.

Le compilateur produira à partir des deux premières lignes du programme le code suivant (sans les commentaires, naturellement) :

```
move.w  #$2710,-$8000(a5)
                ;$2710 représente 10000 en décimal,
                ;-$8000(a5) est l'adresse de x&,
                ;l'instruction équivaut donc à la
                ;ligne x&=10000.
move.w  -$8000(a5),d0    ;x& est écrit dans le
                         ;registre de données 0.
asl.w   #$2,d0  ;La multiplication par une
                ;puissance de deux
                ;(4 en l'occurrence) est exécutée à
                ;l'aide d'une instruction de
                ;décalage de bits, très rapide.
```

```
move.w  d0,-$7ffe(a5)     ;-$7ffe(a5) est l'adresse
                          ;de y&. Le résultat de la
                          ;multiplication est donc
                          ;affecté à y&.
```

L'avantage de l'absence du contrôle du dépassement de valeur entière réside dans la très grande rapidité du programme produit. Un programme avec une boucle FOR, des additions et des soustractions simples de variables entières sera, sous forme de programme compilé, environ 26 fois plus rapide que sous l'interpréteur.

2.3. Les options du compilateur

2.3.1. Liste des options du compilateur et mode de sélection

Cette section décrira tout d'abord brièvement quelles options offre le compilateur et comment fixer ces options. Chaque option sera ensuite décrite dans une section particulière.

%0	N'effectuer les divisions d'entiers sous forme de divisions entières que lorsque le compilateur constate que le résultat sera utilisé comme valeur entière.
%3	Effectuer les divisions d'entiers systématiquement sous forme de divisions entières.
mxxxx	Le programme ne doit utiliser que xxxx octets de mémoire.
*&	Effectuer les multiplications de longs mots avec muls.
*%	Ne pas effectuer les multiplications de longs mots avec muls.
F%	La valeur renvoyée par une fonction doit être entière.
RC&	Traiter les paramètres RC_INTERSECT comme des valeurs sur deux octets.
RC%	Traiter les paramètres RC_INTERSECT comme des valeurs sur quatre octets.
X nom	Tirer la routine nom d'un fichier objet au linkage.
U	Tester une fois Control+Shift+Alternate, EVERY et AFTER.
U+	Tester C+S+A, EVERY et AFTER après chaque instruction.
U-	Examiner si test de C+S+A, EVERY et AFTER.
I+	Activer les routines d'interruption.
I-	Désactiver les routines d'interruption.

S&	Traiter les paramètres de SELECT et CASE comme des valeurs sur deux octets.
S%	Traiter les paramètres de SELECT et CASE comme des valeurs sur quatre octets.
S	Optimiser SELECT-CASE par rapport à la vitesse d'exécution.
S<	Optimiser SELECT-CASE par rapport à la longueur du programme.
ES	Messages d'erreur sous forme de texte.
E#	Messages d'erreur sous forme de numéros
.B+	Un numéro d'erreur doit apparaître au lieu des bombes.
P>	Compiler les sous-routines comme des sous-routines GFA-BASIC.
P<	Compiler les sous-routines comme des sous-routines 68000.
F>	Générer ENDFUNC.
F<	Ne pas générer ENDFUNC.

2.3.2. Division entière

Les options S%0 et S%3 du compilateur agissent sur la division des variables entières (longs mots). Si l'option %0 est activée, les variables à diviser sont converties en variables à virgule flottante, puis divisées, le résultat obtenu étant une variable à virgule flottante.

Si l'option %3 est fixée, les deux variables entières ne sont pas converties en variables à virgule flottante. Le résultat de la division entière est alors interprété comme une valeur entière.

Exemple : le programme

```
$%0
x%=5
y%=2
PRINT x%/y%
```

produit la sortie 2.5 en version de programme compilé. Le fait que le résultat comporte un chiffre après la virgule vous indique que x% et y% ont été converties en nombres à virgule flottante avant que la division ne soit effectuée. Avec l'option %3 du compilateur, vous obtiendrez 2 comme résultat, car x% et y% seront interprétées comme des entiers sans chiffres après la virgule, et ce sera donc une division entière qui sera effectuée. Sous l'interpréteur, cette option n'a naturellement aucun effet, vous obtiendrez 2.5 dans les deux cas.

Le code produit par les deux options permet d'en souligner la différence. Les lignes

```
$%0
a%=x%/y%/z%
```

produisent le code

```
move.l  -$7ffc(a5),d0
bsr     FITOF
move.l  d0,-(a7)
move.w  d2,-(a7)
move.w  d1,-(a7)
move.l  -$7ff8(a5),d0
bsr     FITOF
move.w  (a7)+,d4
move.w  (a7)+,d5
move.l  (a7)+,d3
bsr     FXDIV
move.l  d0,-(a7)
move.w  d2,-(a7)
move.w  d1,-(a7)
move.l  -$7ff4(a5),d0
bsr     FITOF
move.w  (a7)+,d4
move.w  (a7)+,d5
move.l  (a7)+,d3
bsr     FXDIV
bsr     FFTOI
move.l  d0,-$8000(a5)
```

Dans ce listing, vous constatez que le sous-programme FITOF (integer to float), qui sert à convertir les valeurs entières en nombres à virgule flottante est appelé à trois reprises. Le résultat étant renvoyé à une variable entière, la routine FFTOI (float to integer) est également nécessaire. La division s'effectue avec (FXDIV), c'est-à-dire avec une routine de division avec résultat à virgule flottante. Par contre, au programme

```
$%3
a%=x%/y%/z%
```


correspondra le listing assembleur

```
move.l  -$7ffc(a5),d0
move.l  -$7ff8(a5),d1
bsr     LDIV
move.l  -$7ff4(a5),d0
bsr     LDIV
move.l  d0,-$8000
```

Ce petit listing appelle simplement une sous-routine assez simple de division de deux longs mots (LDIV) et peut donc être exécuté beaucoup plus vite qu'avec l'option $%0. La différence de vitesse se traduit environ par un facteur 10 ou 11.

Pour le cas où des variables entières doivent être divisées et où le résultat doit être renvoyé sans autre conversion, l'option $%0 du compilateur est dépourvue de signification. C'est alors la pure arithmétique entière qui est utilisée.

Lorsque ce sont deux variables sur deux octets qui sont divisées, c'est l'instruction DIV du processeur Motorola 68000 qui est utilisée au lieu de la routine LDIV (voyez la section Division dans le chapitre sur l'optimisation du programme).

2.3.3. Multiplication d'entiers

L'option $*& agit sur la multiplication de deux variables entières dont l'une au moins est une variable sur quatre octets. Dans ce cas, c'est normalement une sous-routine de multiplication de deux variables sur quatre octets qui est appelée. Si cependant l'option *& a été fixée, c'est l'instruction muls du processeur Motorola 68000 qui sera utilisé à la place.

Cela a deux conséquences :

1. l'exécution du programme s'en trouve accélérée,
2. l'instruction muls multiplie deux valeurs sur deux octets entre elles et produit une valeur sur quatre octets comme résultat. Si les valeurs sur quatre octets devant participer au calcul contiennent donc des valeurs sortant du domaine numérique d'une variable sur deux octets, on obtiendra, si on utilise deux octets seulement, un autre résultat que sans l'option *&.

Exemple : les lignes

```
$*&
b%=40000
c%=10
a%=b%*c%
PRINT a%
```

produiront sous l'interpréteur le résultat attendu 400000. Dans le programme compilé, on obtiendra par contre -255360 comme résultat car deux octets seulement de b% auront été utilisés, ce qui ne permet pas de représenter une valeur aussi élevée que 40000.

Le code produit sans et avec l'option *& pour la ligne a%=b%*c% montre bien la différence :

```
Avec *&                     Avec *%, sans *&
move.l   $7ff8(a5),d0       move.l   -$7ff8(a5),d0
move.l   $7ffc(a5),d1       move.l   -$7ffc(a5),d1
muls      d1,d0             bsr      LMUL
move.l   d0,-$8000(a5)      move.l   d0,-$8000(a5)
```

C'est normalement l'option *%, qui entraîne l'emploi de LMUL, qui est prédéfinie.

2.3.4. Réservation de place mémoire

L'option $mx du compilateur permet de produire un programme qui, après avoir été lancé, réservera pour lui-même seulement x octets, alors qu'un programme produit par le compilateur GFA-BASIC 3.0 se réserve normalement la totalité de la place mémoire disponible (moins 16 Ko).

L'instruction RESERVE permet bien sûr d'imposer à un programme de ne se réserver qu'une partie de la place mémoire libre, mais cela a un autre effet que l'option $mx du compilateur. Un programme commençant par exemple par l'instruction RESERVE 25600 réserve dans un premier temps, après avoir été lancé, la totalité de la mémoire disponible. Ce n'est qu'ensuite qu'il effectue la première instruction GFA-BASIC, l'instruction RESERVE en l'occurrence, qui libère alors une partie de la place réservée.

L'option $m25600 du compilateur a au contraire pour effet que seulement 25600 octets (s'il y en a autant de disponibles) soient utilisés par le programme et non pas d'abord la totalité de la place libre. Cette option du compilateur doit être employée dans les accessoires car ceux-ci ne doivent utiliser qu'une partie de la mémoire libre. Vous trouverez dans le chapitre consacré aux accessoires un exemple pour cette option du compilateur.

2.3.5. Valeur de fonction

L'option $F% doit être spécifiée comme première ligne d'une fonction pour avoir un effet. Elle impose au compilateur de fournir une valeur entière comme valeur de réponse de cette fonction. La valeur de réponse prédéfinie est une valeur à virgule flottante.

2.3.6. Paramètres RC_INTERSECT

Avec les options $RC& et $RC% du compilateur, vous pouvez définir le mode d'évaluation des paramètres RC_INTERSECT. Par prédéfinition, ces paramètres sont convertis en paramètres sur quatre octets.

Or l'intervalle de valeurs des variables entières sur deux octets suffit généralement pour ces paramètres. Avec l'option $RC&, vous pouvez imposer au compilateur de traiter les paramètres RC_INTERSECT comme des paramètres sur deux octets.

Cette option est très spéciale et n'est importante que dans des cas exceptionnels. La fonction RC_INTERSECT est essentiellement conçue pour le calcul des surfaces d'intersection de surfaces rectangulaires se chevauchant dans le cadre d'un Redraw.

Lors d'un Redraw, on a bien sûr seulement besoin de paramètres sur deux octets, puisque les coordonnées sous GEM ne peuvent comporter plus de deux octets. Le gain de temps obtenu avec l'option $RC& est cependant négligeable dans ce cas, car, d'une part, des instructions autres que celles du calcul des intersections des fenêtres, notamment les instructions de renouvellement d'une section de l'écran, prennent beaucoup plus de temps et, d'autre part, la fonction RC_INTERSECT n'est appelée qu'un nombre de fois restreint dans ce cas.

L'option $RC& permet cependant de gagner un peu de place mémoire lorsqu'on utilise des variables 16 bits. Une autre application est indiquée à propos de Hidden Line.

2.3.7. Routines externes à linker

L'option $X nom doit constituer la première ligne d'une procédure ou d'une fonction. Dans ce cas, le contenu de la procédure ou fonction n'est pas compilé, mais un message est laissé pour le linker, qui lui indique que la routine "nom" doit être tirée d'un fichier objet à linker.

La section "Intégration de routines C sans fonctions de bibliothèque" indique comment utiliser cette option.

2.3.8. Contrôle des touches Stop et de EVERY/AFTER

Au cours de l'exécution du programme, il est normalement possible, sous l'interpréteur GFA-BASIC, d'interrompre le programme en appuyant simultanément sur les touches Control, Shift gauche et Alternate.

Cette possibilité n'existe pas, par prédéfinition, dans le programme compilé. Vous pouvez cependant activer la possibilité d'interrompre le programme avec ces trois touches, à l'aide de l'option SUx du compilateur. Pour que les touches Stop soient au moins testées, encore faut-il cependant que l'option I+ soit activée.

Le test des touches Stop va de pair avec le test des conditions EVERY-AFTER. Les préparatifs nécessaires pour un RESUME/NEXT sont en outre également effectués. Les instructions pouvant provoquer des erreurs doivent être placées à cet effet entre SU. (Plus exactement, SU sauvegarde le compteur de programme PC, le pointeur de pile SP et le pointeur de pile BASIC A3. Il ne faut donc pas que des DIMs, ERASEs ou GOSUBs, FNs aient été exécutés entre le $U et l'erreur.)

Le paramètre x peut revêtir les valeurs suivantes, qui entraînent les possibilités indiquées :

$U	insère un test et un seul.
SU +	insère un test à la suite de chaque instruction générant du code.
SU -	désactive le test.

Le programme suivant ne peut être interrompu en appuyant sur les trois touches Stop au cours de la première boucle, mais il pourra par contre être ainsi interrompu pendant la seconde boucle (après qu'on ait appuyé sur un bouton de la souris).


```
$U-
REPEAT
UNTIL MOUSEK
DO
$U
LOOP
```

La boucle DO-LOOP ne peut être interrompue si le programme comporte seulement

```
$U+
DO
LOOP
```

car un test serait bien, dans ce cas, inséré à la suite de LOOP, mais pas à la suite de DO, DO n'étant pas une instruction génératrice de code.

2.3.9. Routines d'interruption

L'option $I+ et $I- du compilateur permet d'activer ou de désactiver les routines d'interruption pour

```
EVERY et AFTER
```

le fait d'appuyer sur les touches Stop (Control-Shift-Alternate), la saisie du code ASCII avec Alternate+chiffres, la définition des touches de fonction, l'activation du pointeur de la souris en cas d'erreurs disquette (par exemple : veuillez introduire le disque B dans le lecteur A).

N'oubliez pas de positionner l'interrupteur U ou U+ dans votre fichier source.

L'emploi de ces options augmente la longueur du programme compilé.

2.3.10. Paramètres SELECT-CASE

L'option $S& vous permet d'obtenir, pour les instructions SELECT-CASE, que les expressions placées à la suite de ces instructions soient traitées comme des

paramètres sur deux octets. L'option prédéfinie $S% a pour effet que les valeurs placées après SELECT et CASE soient traitées comme des paramètres sur quatre octets.

L'examen du code produit met bien en évidence les différences. Le listing

```
SELECT a%
CASE 1
  case_1:
  INC a%
CASE 2
  case_2:
  INC a%
DEFAULT  default:
  INC a%
ENDSELECT
endsel:
INC a%
```

contient des marques rendant plus lisible le code désassemblé symbolique. Chaque marque est suivie d'une instruction (INC a%) représentant les différents blocs d'instructions.

En employant l'option S& on obtiendrait le code suivant :

```
            move.l      -$8000(a5),d0
            bra.s       L1
_CASE_1:    addq.l      #$1,-$8000(a5)
            bra.s       _ENDSEL
_CASE_2:    addq.l      #$1,-$8000(a5)
            bra.s       _ENDSEL
_DEFAULT:   addq.l      #$1,-$8000(a5)
            bra.s       _ENDSEL
L1:         cmpi.w      #$1,d0   ;Evaluation CASE
            beq.s       _CASE_1
            cmpi.w      #$2,d0
            beq.s       _CASE_2
            bra.s       _DEFAULT
_ENDSEL     addq.l      #$1,-$8000(a5)
```


Le label L1 ne sera pas produit.

Ce listing place tout d'abord la valeur sur quatre octets a% dans d0. Ensuite est appelée la routine de branchement sur les divers blocs d'instructions CASE. Cette routine est marquée dans le listing par le commentaire "Evaluation CASE". Les valeurs placées après CASE y sont comparées avec le contenu de d0, à l'aide de cmpi.w's (compare integer words), sur une longueur de mot seulement.

En cas d'égalité, on saute, avec beq.s (branch if equal), au bloc placé à la suite du CASE correspondant. Chaque bloc se termine par un saut après le bloc SELECT-ENDSELECT. Ce saut est effectué ici avec bra.s _ENDSEL. Si aucune valeur après CASE ne correspond, le bloc DEFAULT est appelé. Sans l'option S&, les comparaisons figurant ici seraient effectuées sur une longueur de mot.

2.3.11. Optimisation SELECT-CASE

Les options S et S< influent sur le critère d'optimisation pour les instructions SELECT-CASE. Avec S, c'est une optimisation par rapport au temps d'exécution qui est effectuée, avec S< une optimisation par rapport à la taille du programme.

Nous avons indiqué, à la section précédente, qu'à la fin des blocs de programme après CASE et l'instruction DEFAULT, on saute à la ligne après ENDSELECT avec BRA.S. Si cependant les blocs de programme sont plus longs que ceux de la section précédente, où ils ne se composaient que d'un ADDQ, BRA doit être employé au lieu de BRAS.

Avec l'instruction BRA, la place nécessaire pour indiquer la distance du saut est plus grande qu'avec BRA.S. Pour économiser cette place, on peut utiliser l'option S<, qui a pour effet de diriger l'instruction BRA figurant à la fin d'un bloc de programme CASE vers l'instruction BRA du bloc de programme suivant (à la suite du prochain CASE ou à la suite de DEFAULT), si toutefois la distance de saut jusque-là n'est pas trop longue.

Ainsi peut se constituer une chaîne de sauts courts, qui débouche finalement à la suite de ENDSELECT. Cette chaîne de sauts de BRA.S en BRA.S peut éventuellement être encore modifiée au cours d'une phase d'optimisation ultérieure.

Ce problème n'étant pas particulièrement simple, le listing suivant nous aidera à illustrer le principe dont il s'agit :

```
              move.l      -$8000(a5),d0
              bra         L1
_CASE_1:      addq.l      #$1,-$8000(a5)
              bra.s       L0
_CASE_2:      addq.l      #$1,-$8000(a5)
L0:           bra         L2
_DEFAULT:     addq.l      #$1,-$8000(a5)
              ...
              Nombreuses instructions                ...
L2:           bra.s       _ENDSEL
L1:           cmpi.w      #$1,d0   ;Evaluation CASE
beq           _CASE_1
              cmpi.w      #$2,d0
              beq         _CASE_2
              bra         _DEFAULT
_ENDSEL       addq.l      #$1,-$8000(a5)
```

Le point décisif est constitué ici par la ligne bra.s L0. Cette instruction Branch devrait en fait sauter après ENDSELECT. Cette distance serait cependant trop longue pour un bra.s, à cause des nombreuses instructions figurant dans le bloc de programme DEFAULT. C'est pourquoi on saute à l'instruction bra du prochain bloc de programme CASE.

Cette construction ralentit l'exécution mais permet de gagner un peu de place par rapport au code produit avec S>, qui ne recourrait pas à ces genres de sauts.

2.3.12. Messages d'erreur

Si une erreur devait apparaître dans un programme compilé, un message d'erreur décrivant le type d'erreur est affiché. Les options E$ et E# vous permettent de décider si ce message doit apparaître sous forme de texte ou sous forme d'un numéro.

Si vous utilisez l'option E$, le message d'erreur apparaîtra sous forme de texte. Le programme compilé sera cependant un peu plus long car il devra comporter les textes d'erreur.

L'option message d'erreur s'applique toujours au programme entier. Elle ne peut pas, comme certaines options, être définie différemment pour les diverses sections du programme.

2.3.13. Numéros d'erreur au lieu de bombes

L'option B+ a pour effet de faire apparaître les numéros d'erreur correspondant aux erreurs produisant normalement des "bombes". Cette option ne doit pas être employée dans les accessoires. L'option B+ peut seulement être activée. Elle est à nouveau annulée à la fin du programme.

2.3.14. Sous-routines

Les procédures et les fonctions sont compilées sous forme de sous-programmes GFA-BASIC lorsqu'est appliquée l'option P>, qui est prédéfinie. Cela permet des transmissions de paramètres et des appels récursifs sans aucune difficulté.

L'option P< entraîne la production de procédures et fonctions sans paramètres et sans variables locales, sous forme de simples sous-routines 68000. Des problèmes peuvent en résulter en cas d'appels récursifs, car la pile risque de déborder. Ces routines présentent cependant l'avantage de pouvoir être exécutées plus rapidement. En cas de P<, aucun RESUME ou RESUME NEXT n'est possible.

2.3.15. Génération de ENDFUNC

Dans un listing GFA-BASIC, une fonction se termine toujours par ENDFUNC. Lorsque cette ligne est atteinte, le message d'erreur 69 (ENDFUNC sans RETURN) apparaît, car une fonction doit avoir été auparavant abandonnée avec RETURN et ne doit jamais être traitée jusqu'à la fin.

Si vous utilisez l'option F>, cela vaudra aussi pour le programme compilé. L'option F< empêche que ce message d'erreur apparaisse. Le code à la suite de la fonction continuera alors à être exécuté, ce qui peut avoir des effets inattendus.
Les programmes compilés sont toutefois généralement des programmes qui ont été testés à fond. L'erreur 69 n'a donc pas de raison de survenir et F< vous permet d'obtenir un programme un peu plus court.

Chapitre 3

Le linker

3.1.Les options du linker

3.1.1.Les options du linker et leur mode de sélection

Cette section décrira tout d'abord brièvement quelles options offre le linker et comment fixer ces options. Chaque option sera ensuite décrite dans une section particulière.

```
-s          Linker avec la table des symboles
+NOMLIB     Utilise la bibliothèque NOM au lieu de
            GFA3BLIB.
NOM         Linke avec le fichier objet NOM.O.
#           Ne pas linker le fichier TEST.O.
```

Les sélections contenues dans G3OBJ, G3PRG, G3LIB, G3WAIT sont transmises au linker par le programme MENU.

3.1.2.Table de symboles

L'option -s fait créer une table de symboles par le linker, pour que le programme puisse être désassemblé et débogué en utilisant des symboles. Les noms des procédures et des fonctions ainsi que les noms de label dans le listing GFA-BASIC sont utilisés comme symboles.

Seuls seront utilisés les sept premiers caractères d'un nom de procédure, de fonction ou de label précédé d'un trait souligné.

3.1.3.Sélection de bibliothèque

L'option +nom_lib du linker permet d'imposer au linker d'utiliser la bibliothèque nom_lib. Si vous n'utilisez pas cette option, le linker recherche une bibliothèque portant le nom GFA3BLIB.

Dans la section "Intégration avec emploi des bibliothèques C", nous décrivons plus précisément comment créer ou compléter une bibliothèque. Cette section explique comment linker avec les bibliothèques de Turbo C.

3.1.4.Linkage de fichiers objets

Pour linker un fichier objet, qui doit se présenter en format DR (Digital Research), vous pouvez transmettre au linker, comme paramètre, le nom de ce fichier sans l'extension .O.

Dans la section "Intégration de fonctions C sans bibliothèques C", nous décrivons plus en détail comment vous pouvez utiliser dans vos programmes GFA-BASIC les routines contenues dans les fichiers objet.

3.1.5.Ne pas linker TEST.O

En spécifiant le caractère # comme paramètre, vous interdisez au linker de linker le fichier TEST.O. Le linker recherche en effet, par prédéfinition, un fichier de ce nom, car le compilateur appelle ainsi tout fichier objet qu'il produit.

3.2.Intégration de fonctions C

Cette section décrit comment intégrer dans des programmes GFA-BASIC 3.0 compilés des routines qui ont été écrites en langage C. La méthode d'intégration ici décrite suppose que le compilateur C soit capable de produire des fichiers objet en format DR, comme par exemple DRC, Turbo C et LATTICE C.

3.2.1.Intégration de fonctions C sans fonctions de bibliothèque

Nous allons maintenant illustrer par un exemple la méthode de base de l'intégration. La routine à intégrer devra remplir avec les nombres de 0 à n les éléments d'un tableau de valeurs sur deux octets. L'adresse et la taille du tableau devront être communiquées à la routine. C'est seulement une variable sur deux octets qui est utilisée comme paramètre de communication de la taille du tableau, qui se trouve ainsi limitée à 32267 éléments. La fonction chargée de cette tâche travaille sans recourir à des fonctions de bibliothèque.

Le programme GFA-BASIC doit être construit de façon à pouvoir tourner également sous l'interpréteur. Sous l'interpréteur, la routine C ne peut naturellement pas être intégrée à l'aide du linker. Pour permettre le fonctionnement sous l'interpréteur, une routine GFA-BASIC doit donc prendre en charge la fonction de la routine C à linker.

Le programme GFA-BASIC se présente ainsi :

```
DIM x&(32000),y&(32000)
'
t1%=TIMER
f_gfa(32000)
t2%=TIMER
PRINT "En GFA-BASIC :"'(t2%-t1%)/200
FOR i&=0 TO 39
  PRINT x&(i&),
NEXT i&
t1%=TIMER
f_c(V:y&(0),32000)
t2%=TIMER
PRINT "En Turbo C   :"'(t2%-t1%)/200
FOR i&=0 TO 39
  PRINT y&(i&),
NEXT i&
'
PROCEDURE f_gfa(n&)
  FOR i&=0 TO n&
    x&(i&)=i&
  NEXT i&
RETURN
'
```

```
PROCEDURE f_c(adr%,n&)
  $X remplir
  i&=0
  a%=adr%
  WHILE i&<=n&
    WORD{a%}=i&
    INC i&
    ADD a%,2
  WEND
RETURN
```

On commence par dimensionner deux tableaux, puis deux chronométrages sont effectués. Le premier chronométrage appelle la routine f_gfa, qui affecte les valeurs de 0 à 32000 aux éléments du tableau x&().

Les 40 premiers éléments du tableau sont ensuite sortis sur l'écran pour contrôle.

Le second chronométrage appelle la routine f_c. Cette routine doit également remplir un tableau avec les valeurs de 0 à 32000, mais ce travail devra être réalisé par une routine C dans le programme compilé et linké. Cette routine reçoit à cet effet l'adresse du tableau et le nombre de valeurs à écrire.

Dans la procédure f_c figure, dans la première ligne, l'instruction ordonnant au linker d'intégrer la fonction C "remplir". Le nom de la fonction à intégrer est indiqué à la suite de l'option $X. Cette ligne doit toujours être la première ligne de la procédure.

Le linker remplace le contenu de cette procédure par la routine spécifiée à la suite de $X. Cette ligne sera ignorée sous l'interpréteur et ce sont les instructions dans les lignes à la suite de $X remplir qui seront exécutées. Dans notre exemple, ce sont trois lignes remplissant exactement la même fonction que la routine C et effectuant l'affectation de valeur à l'aide d'un pointeur.

Venons-en maintenant au programme C chargé de l'affectation. Il a été mis au point sous Turbo C. En voici la teneur :

```
void cdecl remplir(n,adr)
int  n  ;
int *adr;
{
   int i; int *a;
   a = adr;
   for( i=0 ; i<=n ; *a++ = i++ );
}
```


Vous trouvez sur la première ligne le nom de la fonction qui apparaît après $X dans le listing GFA-BASIC. Cette fonction ne fournit pas de valeur de réponse et a donc été déclarée comme void.

Avant le nom de la fonction, vous trouvez encore l'instruction cdecl, qui indique à Turbo C qu'il doit recevoir les paramètres à travers la pile. Il en est automatiquement ainsi sous certains autres compilateurs C, par exemple sous DRC. Dans le listing GFA-BASIC, les paramètres sont transmis dans l'ordre adr%,n&.

Le C retire ces paramètres de la pile dans l'ordre inverse, donc la valeur n sur deux octets en premier, puis l'adresse adr d'une variable de deux octets. Les lignes suivantes effectuent alors l'affectation des valeurs aux éléments du tableau.

La structure des listings GFA-BASIC et C étant maintenant connue, il nous reste encore à expliquer comment vous pouvez imposer au linker d'utiliser la fonction C à l'endroit $X. Il vous suffit pour cela de lui communiquer le nom du fichier objet que vous aviez réalisé à l'aide du compilateur C.

Si vous utilisez un Shell DOS, ce nom peut tout simplement être inscrit dans la ligne de commande. Dans le Shell fourni avec le compilateur GFA-BASIC 3.0, vous pouvez compléter la variable G3OBJ.

L'exécution de la boucle GFA-BASIC dure environ 0,51 secondes, alors que la boucle Turbo C n'a besoin que de 0,115 secondes environ pour réaliser la même affectation. Les affectations à l'aide d'opérations de pointeur peuvent être optimisées de façon très intéressante en Turbo C.

Les opérations suivantes doivent donc être effectuées :

1. Il faut créer dans le programme GFA-BASIC une procédure dans laquelle on placera sur la première ligne l'option $X suivie du nom de la fonction C.

2. On crée un programme C comportant cette fonction. Les paramètres doivent être reçus à travers la pile, où ils figureront dans un ordre inverse par rapport à la transmission de paramètres par le programme GFA-BASIC.

❸ Lors du linkage, le fichier objet du listing C doit être indiqué en même temps que la routine à linker.

Lorsqu'on utilise le compilateur DRC, il faut tenir compte du fait que ce compilateur place encore un trait souligné avant le nom de la fonction C. Au lieu de $X remplir, il faudrait donc spécifier SX _remplir.

Dans notre exemple, la fonction C ne renvoyait pas de valeur de réponse et avait donc été déclarée comme void. Une fonction C peut cependant bien sûr être utilisée non seulement à la place d'une procédure GFA-BASIC, mais aussi à la place d'une fonction GFA-BASIC.

Voici par exemple comment se présenterait un listing GFA-BASIC approprié :

```
valeur%=100
PRINT @dou(valeur%)
'
FUNCTION dou(a%)
  $X doubler
  ALERT 1,"Réserve la place|d'une fonction
C.",1,"Return",a&
  RETURN 0
ENDFUNC
```

Dans cet exemple, la fonction GFA-BASIC ne se charge pas du travail de la routine C à linker. Le programme C tout simple se présente ainsi :

```
long cdecl doubler(x)long x;{
   return x+x;}
```

La valeur de réponse d'une fonction externe de ce type est toujours une valeur entière.

Vous pouvez naturellement aussi linker des routines assembleur. Celles-ci doivent tirer leurs paramètres de la pile et ne doivent pas modifier le pointeur de pile SP ni les registres A3, A4, A5 et A6.

3.2.2. Intégration avec utilisation des bibliothèques C

Dans les routines C utilisées jusqu'ici, seuls des mots-clés C ont été utilisés, mais pas des fonctions tirées de bibliothèques. Si vous voulez par exemple utiliser un printf dans l'une des routines C, vous devrez faire linker également la routine correspondante figurant dans les libraries.

Vous pouvez pour cela compléter la bibliothèque du compilateur GFA-BASIC. Le petit programme suivant vous permet d'intégrer les bibliothèques de Turbo C dans la bibliothèque du compilateur GFA-BASIC, après quoi toutes les routines C intégrées pourront comporter n'importe quelles instructions C figurant dans les bibliothèques. Voici ce programme :

```
OPEN "O",#2,"NOUV_LIB"
OPEN "I",#1,"GFA3BLIB"
l%=LOF(#1)
WHILE l%>32000
  PRINT #2,INPUT$(32000,#1);
  SUB l%,32000
WEND
PRINT #2,INPUT$(l%,#1);
CLOSE #1
REI.SEEK #2,-2
ajouter("TCSTDLIB.LIB")
ajouter("TCEXTLIB.LIB")
ajouter("TCGEMLIB.LIB")
ajouter("TCTOSLIB.LIB")
ajouter("TCFLTLIB.LIB")
ajouter("TCLNALIB.LIB")
PROCEDURE ajouter(lib$)
  OPEN "I",#1,lib$
  l%=LOF(#1)-2
  SEEK #1,2
  WHILE l%32000
    PRINT #2,INPUT$(32000,#1);
    SUB l%,32000
  WEND
  PRINT #2,INPUT$(l%,#1);
  CLOSE #1
RETURN
```

Pour utiliser ce petit programme, vous devrez le cas échéant modifier les noms de chemin des fichiers. Pour pouvoir utiliser la nouvelle bibliothèque, vous avez encore besoin du fichier d'index correspondant (fichier NDX), que vous pouvez créer par exemple avec la gestion de library du GFA-ASSEMBLER ou avec des programmes semblables (DOINDEX).

Dès que les deux fichiers sont prêts, vous pouvez utiliser toutes les instructions des bibliothèques Turbo C dans les routines C à intégrer. Il vous suffit pour cela d'indiquer au linker GFA-BASIC que vous voulez utiliser la nouvelle bibliothèque et non la GFA3BLIB. Vous devez pour cela utiliser l'option +NOMLIB du linker, si vous utilisez un Shell DOS (avec +NOUV_LIB en l'occurrence).

Important : Il se peut alors que le linker annonce qu'il ne connaît pas la variable errno. Les compilateurs C ne la tirent pas, en effet, des bibliothèques, car elle fait partie du code Startup.

L'utilisation sans problème de la nouvelle bibliothèque suppose toutefois que les bibliothèques Turbo C et la bibliothèque GFA3LIB ne comportent aucune fonction dont le nom figure dans les deux bibliothèques. Dans la version actuelle de Turbo C, il n'existe, à notre connaissance, aucune fonction portant ainsi un double nom.

3.2.3.Quelques particularités

Dans les programmes d'une certaine taille, il sera généralement nécessaire de ne pas utiliser d'appels relativement au PC. Vous pouvez éviter qu'il en soit ainsi, sous Turbo C, avec l'option -P du compilateur.

Les routines C ne doivent pas modifier les registres A3 à A6 ni attendre des valeurs déterminées dans ces registres. Turbo C, DRC respectent cette condition, au contraire de Megamax C par exemple. Une pile d'environ 4 Ko est mise à la disposition des programmes C.

Chapitre 4

La programmation des accessoires

4.1.La structure des accessoires

Un accessoire est un programme qui est mis en place dans la mémoire après la mise en marche du système ou après un "Reset" (réinitialisation complète du système). Il attend dans la mémoire jusqu'à ce qu'il reçoive un message lui indiquant qu'il doit entrer en activité. Ce message est déclenché lorsque l'utilisateur a appelé cet accessoire.

La structure d'ensemble d'un accessoire découle de cette simple description. Nous avons indiqué tout d'abord que l'accessoire doit être mis en place. Le programmeur doit à cet effet déclarer l'accessoire comme une application GEM et transmettre le nom de l'accessoire à GEM, pour qu'il puisse être intégré dans le premier des menus déroulants.

C'est la fonction APPL_INIT() qui permet de mettre en place une application GEM. Elle renvoie ce qu'on appelle l'identification de l'application.

A l'aide de cette valeur renvoyée par APPL_INIT(), on peut aussi déterminer si le programme a été lancé comme accessoire ou comme un programme normal. Si le message de réponse est 0, c'est que le programme n'a pas été lancé comme un accessoire, mais comme un programme normal. Vous pouvez donc écrire en GFA-BASIC 3.0 des programmes de façon à ce qu'ils puissent tourner aussi bien comme des programmes normaux que comme des accessoires.

Le numéro d'identification de l'application est également nécessaire pour transmettre le nom de l'accessoire, qui s'effectue avec MENU_REGISTER(ap_id, texte$).

MENU_REGISTER indique en réponse le numéro d'accessoire (0 à 5) sous lequel le programme actuel a été déclaré. Cette valeur est appelée numéro d'identification du menu. Une réponse de -1 indique que 6 accessoires sont déjà installés et qu'il ne reste donc plus de place pour un accessoire supplémentaire. Les deux paramètres transmis sont l'identification de l'application et le nom d'accessoire.

L'accessoire a donc été déclaré. La prochaine étape consistera alors à lui faire attendre qu'il soit appelé. Un accessoire doit pour cela être programmé sous forme d'une boucle sans fin dans laquelle la mémoire-tampon de messages est constamment surveillée.

Cette surveillance peut par exemple être réalisée à l'aide de la fonction EVNT_MESAG(0). Si on transmet un zéro à cette fonction, elle écrira les messages provenant de GEM dans la mémoire-tampon de messages propre à GFA-BASIC, qui peut être lue avec MENU(1) à MENU(8).

Le message le plus important pour un accessoire est MENU(1)=40 (AC_OPEN). Ce message indique en effet que l'utilisateur a sélectionné cet accessoire. Si cependant l'accessoire fonctionne à l'intérieur d'une fenêtre, les messages concernant la sélection des éléments de bord ou le Redraw (redessiner l'écran) sont naturellement également intéressants.

Un accessoire offre normalement à l'utilisateur la possibilité d'éliminer cet accessoire à nouveau de l'écran. Sur un plan interne au programme, cela signifie que l'accessoire doit revenir à la boucle sans fin avec l'instruction EVNT_MESAG(0). Il ne faut utiliser ni instruction END, ni une instruction comparable dans un accessoire, faute de quoi l'ordinateur sera "planté".

Cette description doit encore être complétée sur un point. Les programmes GFA-BASIC essayent tout d'abord, après avoir été lancés, de prendre possession de la totalité de la mémoire libre. Cette zone de la mémoire peut bien sûr être à nouveau libérée avec RESERVE, mais la routine d'initialisation d'un programme compilé effectue la réservation de place mémoire avant que l'instruction RESERVE ne puisse être exécutée.

Pour éviter cela, un accessoire GFA-BASIC doit comporter une ligne commandant le compilateur de telle façon que le programme qu'il aura produit ne confisque pas toute la mémoire libre. Cette instruction est l'option $mx, où x représente le nombre d'octets que le programme compilé doit se réserver.

La section de l'écran qui a été occupée par un accessoire doit être restaurée. Cette opération s'appelle Redraw (redessiner). Si vous utilisez par exemple des boîtes de dialogue, vous pouvez faire effectuer ce Redraw à partir de votre accessoire, en lisant la section de l'écran voulue avec GET, puis en la restaurant avec PUT.

4.2.Un programme d'exemple

Après ce cours très théorique sur la programmation des accessoires, voici un programme d'exemple. Voici d'abord un exemple particulièrement bref :

```
$m1000
ap_id&=APPL_INIT()
IF ap_id&=0
  ALERT 1,"Je ne suis pas|un accessoire pour le
moment.",1,"Return",a&
  END
ENDIF
me_id&=MENU_REGISTER(ap_id&,"  Commcoul ")
DO
  EVNT_MESAG(0)
  IF MENU(1)=40
    ALERT 1,"Commutation des couleurs",1,"nb|bn",a&
    SETCOLOR 0,a&
  ENDIF
LOOP
```

Dans la première ligne, le programme se réserve 1000 octets de place mémoire. Il se déclare alors comme application GEM et se procure ainsi une identification d'application, qu'il dépose dans la variable ap_id&. Si ap_id& vaut 0, c'est que le programme n'a pas été lancé comme un accessoire. Il l'annonce et se termine.

La ligne comportant MENU_REGISTER transmet le nom de l'accessoire à GEM et procure un numéro d'identification du menu me_id&. Vient ensuite une boucle DO-LOOP sans fin, avec surveillance de la mémoire-tampon de messages par EVNT_MESAG(0).

Lorsque l'accessoire reçoit le message lui indiquant qu'il a été appelé (MENU(1)=40), il entre en action. Cette action est assez sommaire en l'occurrence. L'accessoire permet d'inverser l'écran avec SETCOLOR, après quoi il retourne à sa boucle sans fin.

4.3.Programmes d'exemple d'envergure

Nous vous présenterons dans cette section deux exemples d'accessoire, qui ne remplissent pas véritablement une fonction utile mais servent uniquement en guise d'illustration. Le premier programme d'exemple produit une fenêtre dont le contenu est constitué par un motif de remplissage. Ce programme tourne comme programme et comme accessoire. L'autre exemple se présente sous la forme d'une boîte de dialogue avec des boutons radio, un champ d'édition et des boutons.

Commençons par le programme de fenêtre : le programme se réserve tout d'abord 5120 octets et se procure alors une identification d'application. Il examine ensuite si l'identification est différente de 0. La valeur 0 signifierait que le programme a été lancé comme un programme et non comme un accessoire.

S'il a été lancé comme accessoire, il inscrit son nom dans la ligne de menus. Il saute ensuite à une boucle sans fin se composant uniquement de deux instructions. La première instruction est la fonction EVNT_MESAG(0), qui attend l'intervention d'événements. Le paramètre 0 fait que les messages pourront être saisis à l'aide des fonctions MENU(1) à MENU(8). La seconde instruction appelle la sous-procédure message qui réagit aux messages reçus.

Si le programme n'a pas été lancé comme accessoire, il ouvre une fenêtre et fixe une routine d'évaluation de message. Cette routine est identique à celle chargée de l'évaluation des messages dans la variante accessoire. Dans la boucle REPEAT-UNTIL, on attend alors l'apparition des événements, jusqu'à ce que la variable exit! soit fixée sur TRUE dans la procédure message.

La procédure message a presque exactement la même structure qu'une routine normale de gestion d'une fenêtre dans un programme normal, qui n'est pas destiné à tourner en tant qu'accessoire. Elle se procure tout d'abord les coordonnées de la fenêtre actuelle dans les variables x&, y&, b& et h&.

Dans l'instruction SELECT-CASE, des branchements sont effectués en fonction des différents messages pouvant être renvoyés dans MENU(1). Le premier message traité est aussi le plus important, puisqu'il s'agit du message Redraw.

Dès que ce message intervient, l'AES est informé avec WIND_UPDATE(1) de la structure actuelle de l'écran. La construction qui suit parcourt alors la liste des rectangles de la fenêtre qu'il s'agit de redessiner.

Chaque rectangle est dessiné avec un motif de remplissage aléatoire. Vous pouvez ainsi surveiller la gestion de la liste des rectangles par GEM, sauf si deux rectangles

contigus se voient, par hasard, attribuer le même motif de remplissage. Pour le renouvellement des rectangles, il suffirait en fait de ne dessiner le rectangle que dans la zone dotée d'un clipping.

On procède cependant ici comme dans un programme de fenêtre ordinaire. Le clipping est placé sur le rectangle à redessiner, après quoi la zone de travail complète de la surface de la fenêtre est redessinée. Dans un programme à vous, il vous faudrait remplacer l'instruction PBOX par la routine redessinant le contenu de la fenêtre dans votre programme.

Un petit nombre d'instructions suffit pour répondre aux autres messages. On utilise généralement des instructions spécifiques de GFA-BASIC telles que TOPW, FULLW, etc. Avec le message WM_CLOSED, outre le CLOSEW, la variable exit! est encore fixée sur TRUE, au cas où le programme ne tournerait pas comme accessoire.

Cette routine se distingue d'une routine de gestion de fenêtre normale par sa faculté de réagir au message AC_OPEN. Ce message indique que l'utilisateur a sélectionné l'accessoire, auquel cas la fenêtre est ouverte.

Les zones qui ont été recouvertes par l'accessoire doivent être renouvelées par le programme qui a appelé l'accessoire. C'est GEM qui se charge d'envoyer un message Redraw approprié à ce programme. Cela n'a donc pas à être réalisé dans ce listing, sur votre disquette, le programme s'appelle "WIND-ACC.GFA".

```
$m5120
ap_id&=APPL_INIT()
IF ap_id&<0
  me_id&=MENU_REGISTER(ap_id&," Fenêtre-ACC")
  DO
    ~EVNT_MESAG(0)
    message
  LOOP
ELSE
  TITLEW #1,"Essai de fenêtre"
  INFOW #1," en GFA-BASIC 3.0"
  OPENW #1,50,50,200,100,&X111111
  handle&=W_HAND(#1)
  ON MENU MESSAGE GOSUB message
  exit!=FALSE
```

```
  REPEAT
    ON MENU
  UNTIL exit!
ENDIF'
PROCEDURE message
  x&=MENU(5)
  y&=MENU(6)
  b&=MENU(7)
  h&=MENU(8)
  '
  SELECT MENU(1)
  CASE 20       ! /*** WM_REDRAW ***/
    WIND_UPDATE(1)
    WIND_GET(handle&,11,rx&,ry&,rb&,rh&)    !
Premier rectangle de la liste.
    WIND_GET(handle&,4,ax&,ay&,ab&,ah&)     !
Fenêtre de zone de travail.
    REPEAT
      IF
RC_INTERSECT(ax&,ay&,ab&,ah&,rx&,ry&,rb&,rh&)
        CLIP rx&,ry&,rb&,rh& OFFSET ax&,ay&
        DEFFILL 1,2,RAND(25)
        PBOX 0,0,PRED(ab&),PRED(ah&)
        CLIP 0,0,WORK_OUT(0),WORK_OUT(1)
      ENDIF
      WIND_GET(handle&,12,rx&,ry&,rb&,rh&) !
Rectangle suivant de la liste.
    UNTIL rb&=0 AND rh&=0                   ! Plus
de rectangle dans la liste.
    WIND_UPDATE(0)
  CASE 21       ! /*** WM_TOPPED ***/
    TOPW #1
  CASE 22,41    ! /*** WM_CLOSED et AC_CLOSE ***/
    CLOSEW #1
    exit!=TRUE
  CASE 23       ! /*** WM_FULLED ***/
    FULLW #1
  CASE 27       ! /*** WM_SIZED ***/
    WIND_SET(handle&,5,x&,y&,MAX(180,b&),MAX(80,h&))
  CASE 28       ! /*** WM_MOVED ***/
    WIND_SET(handle&,5,x&,y&,b&,h&)
```



```
  CASE 40       ! /*** AC_OPEN ***/
    TITLEW #1,"Accessoire de fenêtre"
    INFOW #1," en GFA-BASIC 3.0"
    OPENW #1,50,50,200,100,&X111111
    handle&=W_HAND(#1)
  ENDSELECT
RETURN
```

Venons-en maintenant au prochain programme d'exemple, qui utilise une boîte de dialogue. Ce programme se réserve beaucoup de place mémoire au départ, presque 13 Ko. Il a surtout besoin de cette place pour sauvegarder avec GET la zone de l'écran qui sera recouverte par la boîte de dialogue.

Dans le programme de fenêtre, GEM émet un message Redraw lorsque la fenêtre de l'accessoire est refermée. Si la boîte de dialogue d'un accessoire doit disparaître de l'écran et que la zone qu'elle recouvrait doit être reconstituée, il faut avoir recours à d'autres possibilités.

Une possibilité consiste à déclencher un message Redraw pour la zone de l'écran concernée. C'est ce que permet la routine FORM_DIAL(3,...). Le programme que nous allons vous proposer bientôt illustre l'autre possibilité, qui consiste à lire avec GET, puis à restaurer ultérieurement avec PUT la zone qui sera recouverte par la boîte de dialogue.

L'emploi de FORM_DIAL requiert moins de place mémoire. Il se peut cependant que le programme chargé d'exécuter le Redraw ait alors besoin d'un temps assez long pour mener cette opération à bien. La méthode avec GET et PUT est nettement plus rapide.

Ce programme appelle, par ailleurs, aussi une boîte d'alerte. La zone recouverte par celle-ci est automatiquement renouvelée par GEM. GEM recourt pour cela à un procédé semblable à celui que nous avons employé dans ce programme avec GET et PUT. La zone sous la boîte d'alerte n'a donc pas à être reconstituée par le programme BASIC avec un Redraw, puisque GEM se charge de ce travail.

Après avoir déterminé l'identification d'application, le programme examine s'il a été lancé comme accessoire. Si ce n'est pas le cas, il se termine. Contrairement à la démonstration de fenêtre que nous venons de vous présenter, cette démonstration ne tourne donc qu'en tant qu'accessoire.

L'accessoire essaye ensuite de charger son fichier Resource. S'il ne le trouve pas, il ne doit pas se terminer simplement comme un programme ordinaire. Il parviendrait en effet dans une boucle sans fin avec une instruction EVNT qui attendrait tout simplement l'arrêt de l'alimentation en courant électrique.

Si le fichier Resource a été trouvé, le programme affecte les numéros d'objet de la boîte de dialogue à des variables. Cette partie du programme est le fichier LST sauvegardé par le Resource Construction Set. Après l'affectation, l'adresse de l'arbre de la boîte de dialogue est déterminée et un texte prédéfini est affecté au champ d'édition qu'elle contient.

Ce n'est qu'alors que l'accessoire est inscrit dans la ligne de menus déroulants. Dans la boucle qui suit, il attend ensuite d'être appelé (MENU(1)=40). Dès qu'il est appelé, les dimensions de la boîte de dialogue sont déterminées et la zone correspondante est sauvegardée, avec une petite marge de sécurité. La boîte de dialogue peut ensuite être dessinée et être gérée avec FORM_DO.

Une fois le dialogue achevé, l'objet Exit est replacé en état non sélectionné et la zone de l'écran effacée par la boîte est réécrite avec PUT.

Les lignes suivantes lisent les informations inscrites par l'utilisateur dans la boîte de dialogue et les affichent dans une boîte d'alerte, après quoi l'accessoire retourne à la boucle sans fin, où il attend le prochain appel. Ce programme se trouve sur la disquette sous le nom de "DIAL-ACC.GFA".

```
$m12800
ap_id&=APPL_INIT()
IF ap_id&=0
  ALERT 1,"Ce programme ne tourne que|comme
accessoire.",1," Return ",a|
  END
ENDIF
'
IF RSRC_LOAD("dial_acc.rsc")=0
  DO                !Si le fichier RSC n'a pu être
                    !trouvé,
  ~EVNT_TIMER(-1)   !attendre simplement pendant
                    !longtemps
  LOOP              !qu'il n'y ait rien à faire.
ENDIF
```


* Numéros d'objet provenant du Construction Set

```
LET dialogue&=0 !RSC_TREE
LET texte&=8 !Obj in #0
LET arret&=2 !Obj in #0
LET return&=3 !Obj in #0
LET radio1&=5 !Obj in #0
LET radio2&=6 !Obj in #0
LET radio3&=7 !Obj in #0
'
RSRC_GADDR(0,dialogue&,adr_dialogue%)
CHAR{{OB_SPEC(adr_dialogue%,texte&)}}="Texte de
test"
me_id&=MENU_REGISTER(ap_id&,"  ACC de dialogue ")
'
DO
  EVNT_MESAG(0)
  IF MENU(1)=40
    FORM_CENTER(adr_dialogue%,x&,y&,b&,h&)
    WIND_UPDATE(1)
    GET x&-4,y&-4,x&+b&+4,y&+b&+4,sauver$          !
pour Redraw.
    OBJC_DRAW(adr_dialogue%,0,3,x&,y&,b&,h&)
    ex&=FORM_DO(adr_dialogue%,0)
    OBJC_CHANGE(adr_dialogue%,ex&,0,x&,y&,b&,h&,0,0)
    PUT x&-4,y&-4,sauver$                          !
Redraw.
    WIND_UPDATE(0)
    '
    t$=CHAR{{OB_SPEC(adr_dialogue%,texte&)}}
    IF BTST(OB_STATE(adr_dialogue%,radio1&),0)
      r$="1"
    ELSE IF BTST(OB_STATE(adr_dialogue%,radio2&),0)
      r$="2"
    ELSE IF BTST(OB_STATE(adr_dialogue%,radio3&),0)
      r$="3"
    ENDIF
    '
```

```
    a$="Texte : "+t$+"|Bouton radio : "+r$+"|Bouton
Exit : "+STR$(ex&)
    ALERT 1,a$,1," Ok ",a|
  ENDIF
LOOP
```


Chapitre 5

Optimisation du programme

Le compilateur 3.0 produit du code assembleur. Vous pouvez le soutenir dans ce travail par une programmation appropriée. L'objet de ce chapitre sera donc de décrire comment un programme GFA-BASIC devrait se présenter pour que le programme compilé qui en résulte soit aussi rapide et court que possible. Pour atteindre cet objectif, il vous faut savoir comment le compilateur procède lui-même pour optimiser un programme.

Nous vous présenterons à cet effet, dans ce chapitre, un certain nombre de listings assembleur représentant un code produit par le compilateur. Si vous voulez désassembler vous-même les programmes produits par le compilateur, il est préférable d'intégrer la table des symboles lors du linkage. Le programme compilé commence toujours par les lignes

```
STARTADR:   lea.l    STARTADR-256(pc),a0
            lea.l    $100(a0),a7
            jsr      INIT
```

Ces trois lignes sont suivies du code correspondant aux premières lignes de votre programme BASIC. La fin de ce code est marquée par le label BaseA4. Lors d'un linkage avec la table de symboles, les labels, les noms de procédures et de fonctions sont utilisés sous forme de symboles (souligné plus nom).

Vous pouvez tirer profit de ce chapitre, même si vous ne disposez pas de connaissances en assembleur, en examinant les suggestions dérivées des listings présentés. Il vous sera cependant difficile, dans ce cas, de comprendre en quoi telle formulation d'un programme est plus efficace que telle autre.

Nous avons volontairement veillé à ce que les connaissances de base en assembleur nécessaires à la compréhension des listings désassemblés soient aussi réduites que possible. Ce chapitre s'adresse donc tout particulièrement aux programmeurs en GFA-BASIC qui veulent se familiariser avec l'assembleur pour pouvoir, par exemple, écrire en assembleur des sous-routines pour lesquelles la vitesse d'exécution joue un rôle crucial.

Il ne nous est naturellement pas possible d'évoquer toutes les instructions que le compilateur peut traiter. Nous prendrons donc simplement quelques exemples qui présentent un intérêt particulier sous l'angle de l'optimisation.

5.1. Additions simples

Le processeur de l'Atari ST, le Motorola 68000, dispose d'instructions pour traiter les nombres entiers. Les lignes BASIC constituées uniquement de calculs entre variables entières peuvent être traduites en très peu d'instructions assembleur, tant que ces lignes ne sont pas trop complexes.

Il faut beaucoup plus d'instructions assembleur pour traiter des variables à virgule flottante, et la vitesse d'exécution ne peut que s'en ressentir. L'addition de deux nombres entiers s'effectue en cinq ou six fois moins de temps que l'addition de deux nombres à virgule flottante.

La première question qu'il nous faut évoquer est : quand le compilateur peut-il recourir à l'arithmétique entière, plus rapide, et quand a-t-il recours à l'arithmétique à virgule flottante ?

Sous l'interpréteur, les instructions a%=b%+c% et a%=ADD(b%,c%) ne sont pas traitées avec la même rapidité. Dans cette situation simple, le compilateur produit le même code à partir des deux instructions, à savoir :

```
move.l   -$7ff8(a5),d0
add.l    -$7ffc(a5),d0
move.l   d0,-$8000(a5)
```

Les adresses -$xxxx(a5) dépendent des variables utilisées dans le programme. Les variables sont situées à partir de -$8000(a5).

Si vous avez utilisé des variables sur deux octets, par exemple a&=b&+c&, le compilateur produira les instructions


```
move.w   -$7ffc(a5),d0
add.w    -$7ffe(a5),d0
move.w   d0,-$8000(a5)
```

Avec des variables sur un octet (a!=b!+c!), on obtiendra

```
moveq.l  #$0,d0
move.b   -$7ffe(a5),d0
move.b   -$7fff(a5),d1
add.l    d1,d0
move.b   d0,-$8000(a5)
```

Une ligne de programme comme par exemple a=b+c utilisera cependant des variables à virgule flottante. Pour pouvoir effectuer une addition à virgule flottante, le programme compilé contiendra une routine appropriée. Dans le listing désassemblé d'un programme avec table de symboles, vous trouverez

```
lea.l    -$7ff0(a5),a0
lea.l    -$7ff8(a5),a1
bsr      VVFADD
lea.l    -$8000(a5),a0
move.l   d0,(a0)+
move.w   d1,(a0)+
move.w   d2,(a0)+
```

Vous trouverez dans ce listing la ligne BSR VVFADD. Cette ligne saute à la routine d'addition à virgule flottante dont l'exécution dure beaucoup plus longtemps que l'exécution des quelques instructions générées par exemple par a%=b%+c%.
C'est le code produit à partir de l'addition sur deux octets qui peut être traité le plus rapidement par l'ordinateur. Vient ensuite le code de l'addition sur un octet, puis de l'addition sur quatre octets, et enfin, loin derrière, le code de l'addition à virgule flottante.

La table suivante vous présente un exemple pour les différences de temps d'exécution avec les différents types de variables pour une addition simple sous l'interpréteur et sous le compilateur. Nous avons utilisé pour ce banc d'essai une boucle parcourue 50000 fois, le temps nécessaire au traitement des instructions de boucle ayant été soustrait.

Type	Interpréteur	Compilateur	Instruction
Byte	11,715	0,31	a|=b|+c|
Word	11,715	0,255	a&=b&+c&
Long	11,355	0,335	a%=b%+c%
Float	6,23	1,925	a=b+c

La situation est tout à fait semblable en ce qui concerne la soustraction.

5.2 .Multiplication

L'optimisation effectué par le compilateur pour la multiplication présente une particularité. Le processeur de l'Atari ST dispose en effet d'instructions de multiplication, mais leur exécution nécessite de très nombreux cycles d'horloge et ces instructions n'existent que pour les valeurs sur deux octets. Le compilateur GFA-BASIC 3.0 doit donc coder sans recourir aux instructions assembleur mulu et muls les multiplications d'une constante avec une variable sur quatre octets.

La ligne

```
x%=y%*4
```

peut par exemple être convertie très simplement, vers le code

```
move.l  -$8000(a5),d0
lsl.l   #$2,d0
move.l  d0,-$7ffc(a5)
```

puisque 4 est une puissance de deux. Le codage de l'instruction

```
x%=y%*21
```

sera déjà un peu plus complexe car 21 correspondant à 10101 en binaire. Pour mieux comprendre le code produit par le compilateur, supposons, dans les commentaires sur le code, que y%=100.

Le code généré se présenterait ainsi :

```
move.l  -$8000(a5),d0    ;y%, c'est-à-dire 100,
                         dans le registre d0
move.l  d0,d2            ;une copie de y% dans le
                         registre d2
lsl.l   #$2,d0           ;d0 fois 4, d0 contient
                         maintenant 400, d2=100
add.l   d0,d2            ;additionner 400 à 100,
                         donc d0=400, d2=500
lsl.l   #$2,d0           ;d0 fois 4, donc d0=1600,
                         d2=500
add.l   d2,d0            ;additionner d2 à d0:
                         d0=2100, d2=500
move.l  d0,-$7ffc(a5)    ;écrire le résultat 2100
                         dans x%
```

On occupe ainsi 6 octets de plus en mémoire qu'avec un muls #$15,d0 (qui n'existe pas, toutefois, pour les valeurs sur quatre octets), mais avec un temps d'exécution de 52 cycles d'horloge (sans les première et dernière instructions move), cette formulation est plus rapide.

L'instruction

```
x&=y&*21
```

serait par contre codée en employant l'instruction muls. Pour 1000 multiplications par 21, l'instruction avec les variables sur quatre octets sera donc un peu plus rapide :

Instruction	Temps d'exécution
x%=y%*21	0,975
x&=y&*21	1,08

Si cependant on ne multiplie pas par 21, mais par des valeurs dont la décomposition en instructions assembleur add, sub, lsl et asl requiert un code relativement long, la variante sur quatre octets sera beaucoup plus longue mais aussi beaucoup plus lente que l'instruction sur deux octets qui travaille avec muls. Si on utilise par exemple 21845 (101010101010101 en binaire) au lieu de 21, les temps d'exécution pour 1000 répétitions seront les suivants

Instruction	Temps d'exécution
x%=y%*21845	2,26
x&=y&*21845	1,34

Nous nous sommes intéressés jusqu'ici à la multiplication d'une variable par une constante. Lorsqu'on multiplie deux variables entières, il est cependant capital de prendre en compte le fait que l'instruction mul du processeur ne peut s'appliquer qu'à des valeurs sur deux octets.

La liste suivante présente le code produit par les différentes multiplications entières et leur temps d'exécution pour 2500 multiplications.

```
move.l   -$7ff4(a5),d0      ;x%=y%*z%, 7.685
move.l   -$7ff8(a5),d1
bsr      LMUL
move.l   d0,-$7ffc(a5)
move.w   -$7ff0(a5),d0      ;x%=y%*z&, 6.165
ext.l    d0
move.l   -$7ff8(a5),d1
bsr      LMUL
move.l   d0,-$7ffc(a5)
move.w   -$7ff0(a5),d0      ;x%=y&*z&, 2.44
muls     -$7fee(a5),d0
move.l   d0,-$7ffc(a5)
move.w   -$7ff0(a5),d0      ;x&=y&*z&, 2.315
muls     -$7fee(a5),d0
move.w   d0,-$7fec(a5)
```

La grande différence de temps entre les premières et les deux dernières est due au fait que, dans les deux derniers cas, muls est utilisé au lieu de la routine LMUL. Les temps absolus dépendent toutefois aussi des valeurs figurant dans y%, z%, y& et z&. Les rapports entre les temps d'exécution des différentes instructions restent cependant à peu près comparables.

Dans ces deux cas où une variable sur quatre octets est impliquée, on peut également imposer l'emploi de muls avec l'option *& du compilateur. Il convient cependant de veiller dans ce cas à ce que deux octets seulement des valeurs sur quatre octets impliquées participent effectivement au calcul. Le chapitre sur les options du compilateur (multiplication entière) vous propose une description plus détaillée.

Les petites différences entre les deux cas avec LMUL et les deux cas avec muls reposent uniquement sur l'opposition entre move.w d'une part et move.l et l'instruction ext.l d'autre part. Pour la multiplication des variables entières, il est donc profitable d'utiliser des variables sur deux octets au lieu de variables sur quatre octets.

5.3.Divisions

Les divisions de variables entières peuvent déboucher sur des chiffres après la virgule, par exemple x%=5, y%=2, x%/y%=2.5. Contrairement à la multiplication, la division exigerait donc en principe que le compilateur effectue systématiquement les opérations avec virgule flottante. C'est d'ailleurs ce qu'il fait, à moins que le programmeur ne lui ait pas explicitement imposé d'effectuer la division de deux variables entières sous forme de division entière avec résultat entier.

C'est l'option S%3 du compilateur qui permet d'obtenir qu'il en soit ainsi. L'option inverse S%0, qui entraîne que toute division soit effectuée sous forme d'une division avec virgule flottante, est prédéfinie. L'analyse du code produit par la ligne

```
x=y%/z%
```

montre la différence. Avec l'option %0, prédéfinie, on obtient :

```
move.l   -$7ff8(a5),d0
bsr      FITOF
move.l   d0,-(a7)
move.w   d2,-(a7)
move.w   d1,-(a7)
move.l   -$7ff4(a5),d0
bsr      FITOF
move.w   (a7)+,d4
move.w   (a7)+,d5
move.l   (a7)+,d3
bsr      FXDIV
lea.l    -$8000(a5),a0
move.l   d0,(a0)+
move.w   d1,(a0)+
move.w   d2,(a0)+
```

Vous voyez que les deux variables à diviser sont transformées en variables à virgule flottante avec la routine FITOF (integer to float). Ensuite est appelée la routine de division de deux valeurs à virgule flottante (FXDIV) et le résultat est affecté à la variable x, à partir de -$8000(a5).

Si la même ligne est compilée avec l'option %3, le code suivant sera généré :

```
move.l   -$7ffc(a5),d0
move.l   -$7ff8(a5),d1
bsr      LDIV
lea.l    -$8000(a5),a0
bsr      ISTOF
```

La routine ISTOF se charge de l'affectation à x. La routine LDIV est ici appelée pour diviser deux variables entières en format de long mot. Si les valeurs à diviser font que le résultat arithmétique doit comporter des chiffres après la virgule, ceux-ci ne sont naturellement pas renvoyés par LDIV.

Si la ligne de programme était x%=y%/z%, au lieu de x=y%/z% comme ici, ce serait sans conséquence puisque le résultat doit être placé dans une variable entière. Dans ce cas, le compilateur GFA-BASIC utilise d'ailleurs systématiquement LDIV pour la division, même si $%0 est activée.

S'il s'agit de diviser des variables sur deux octets, on peut employer l'instruction div du Motorola 68000, qui est bien sûr encore plus rapide que la routine LDIV. Le code est alors particulièrement simple et rapide. La ligne

```
x&=y&/z&
```

produira alors :

```
move.w   -$7ffe(a5),d0
ext.l    d0
divs     -$7ffc(a5),d0
move.w   d0,-$8000(a5)
```

On peut donc en conclure qu'il faut, autant que possible, utiliser des variables sur deux octets pour la division et qu'il ne faut activer l'option %3 que lorsque vous êtes sûr que les variables entières divisées ne produisent pas de chiffres après la virgule significatifs.

Pour la division par des constantes, des optimisations semblables à celles pour la multiplication sont effectuées. C'est ainsi que la division par des puissances de deux est convertie en instructions de décalage de bits.

5.4.Calculs complexes

Nous n'avons jusqu'ici évoqué que les calculs avec deux variables. Nous allons maintenant nous intéresser à ce qui se passe lorsque plus de deux variables sont combinées dans un calcul. Si ce type de lignes d'instructions atteignent un certain niveau de complexité, le compilateur ne peut plus les traduire en code qui ne comporte pas de sauts à des sous-routines. L'exemple suivant nous permettra de mettre ce principe en évidence.

Pour traiter la ligne de programme

```
x&=x&+(y&*y&-y&)/y&+z&
```

50000 fois, un programme compilé (sans les instructions de boucle) aura besoin de 2,56 secondes environ. Or cette ligne pourrait aussi être décomposée en

```
x&=x&+(y&*y&-y&)/y&
x&=x&+z&
```

Sur un plan fonctionnel, ces deux lignes aboutissent exactement au même résultat que l'instruction précédente, mais le programme pourra exécuter ces lignes 50000 fois en 1,875 secondes seulement. On peut donc en déduire la règle suivante : on a intérêt à décomposer en plusieurs lignes les lignes comportant des calculs entiers complexes.

Pour bien comprendre pourquoi la variante sur deux lignes est traitée plus rapidement que celle à une ligne, analysons la section correspondante du programme désassemblé (symbolique). La ligne

```
x&=x&+(y&*y&-y&)/y&+z&
```

sera convertie en

```
move.w   -$7ffe(a5),d0
muls     -$7ffe(a5),d0        ;y&*y&
movea.w  -$7ffe(a5),a0
sub.l    a0,d0                ;-y&
move.w   -$7ffe(a5),d1
t.l    d1
bsr      LDIV                 ;/y&
movea.w  -$8000(a5),a0
add.l    a0,d0                ;+x&
add.w    -$7ffc(a5),d0        ;+z&
move.w   d0,-$8000(a5)
```

Les adresses des variables dépendent ici aussi de l'ordre dans lequel elles ont été introduites dans le programme. En l'occurrence, la variable x& a été la première introduite dans le programme, y& la deuxième et z& la troisième. Leurs adresses respectives sont donc

```
x&   -$8000(a5)
y&   -$7ffe(a5)
z&   -$7ffc(a5)
```

Vous trouvez dans le listing la ligne bsr LDIV, c'est-à-dire l'appel d'une sous-routine de division de deux valeurs entières. Si vous comparez maintenant ce listing avec la version sur deux lignes, vous vous apercevez que cette dernière ne comporte pas d'appel de sous-routine. Les instructions

```
x&=x&+(y&*y&-y&)/y&
x&=x&+z&
```

produisent le code

```
move.w   -$7ffe(a5),d0
muls     -$7ffe(a5),d0        ;y&*y&
movea.w  -$7ffe(a5),a0
sub.l    a0,d0                ;-y&
move.w   -$7ffe(a5),d1
divs     d1,d0                ;/y&
add.w    d0,-$8000(a5)        ;Fin de la première ligne
```



```
move.w   -$7ffc(a5),d0
add.w    d0,-$8000(a5)        ;+z&
```

Comme vous le voyez, ce listing comprend un certain nombre d'instructions de moins que le précédent et il ne contient aucun appel de sous-routine avec bsr (branch subroutine). Notez d'ailleurs que le code généré sera encore plus efficace si vous utilisez la notation polonaise avec ADD x&,... au lieu de x&=x&+....

Sous l'interpréteur, la variante à une ligne est plus rapide que celle à deux lignes. Mais si vous voulez adapter vos programmes au compilateur, vous avez intérêt à ne lui proposer que des instructions de calculs entiers assez courtes, qu'il pourra coder en instructions assembleur avec une efficacité optimale.

Il arrive souvent, dans des calculs, qu'on place des parenthèses à des endroits où elles sont en fait superflues et servent seulement à rendre l'expression plus lisible. On pourrait par exemple utiliser la ligne

```
x&=x&+((y&*y&)-y&)/y&
```

au lieu de

```
x&=x&+(y&*y&-y&)/y&
```

Le problème n'est pas ici de discuter si cet exemple est véritablement plus clair avec une paire de parenthèses supplémentaire. Vous constaterez en tout cas que la version avec une paire de parenthèses supplémentaire est plus lente sous l'interpréteur. Sous le compilateur, par contre, les deux versions sont aussi rapide car c'est exactement le même code qui est produit à partir des deux.

Si, par conséquent, vous êtes tenté, par souci de clarté, d'ajouter des parenthèses n'ayant aucun effet sur le résultat, sachez que vous pouvez le faire sans que l'exécution du programme compilé s'en trouve ralentie.

5.5.Les boucles

Information la plus notable concernant l'optimisation des programmes sous GFA-BASIC 2.0 était que la boucle FOR-NEXT était la plus rapide sous l'interpréteur 2.0, alors que c'était la boucle REPEAT-UNTIL qui était plus rapide sous le compilateur 2.0. Il en va autrement sous le compilateur GFA-BASIC 3.0.

Si on fait exécuter une boucle vide 50000 fois avec les trois types de boucle FOR-NEXT, REPEAT-UNTIL et WHILE-WEND, on obtient les temps suivants pour une variable de comptage sur quatre octets :

	FOR-NEXT	REPEAT-UNTIL	WHILE-WEND
Interpréteur	3.76	14.765	16.42
Compilateur	0.77	0.77	0.77

Nous n'évoquerons pas ici la boucle DO-LOOP avec EXIT IF, ni les dérivés de la boucle DO-LOOP, DO-LOOP UNTIL par exemple. Indiquons simplement que la boucle DO-LOOP avec EXIT IF est plus lente sous le compilateur (0.875).

Comme le montre cette table, les trois types de boucle tournent aussi vite dans un programme compilé. Sous le compilateur 3.0, il n'est donc plus utile "d'adapter" un programme au compilateur en convertissant toutes les boucles en boucles REPEAT-UNTIL.

L'équivalence des temps d'exécution amène naturellement à supposer que le code produit par les trois boucles doit être assez semblable. Cette supposition est parfaitement fondée, comme les listings désassemblés des trois types de boucle vont maintenant nous le montrer. Nous avons encore ajouté l'instruction b%=10 dans chaque boucle, pour permettre d'identifier la position du corps de la boucle dans le code.

La boucle FOR

```
FOR i%=1 TO 10000
  b%=10
NEXT i%
```

produira le listing suivant (les adresses appelées et les adresses de variables nous serviront encore une fois uniquement d'exemple) :


```
        moveq.l #$1,d0          ;La boucle démarre à 1.
        move.l  d0,-$7ff8(a5)   ;Affecter la valeur
                                 initiale à i%.
L1:     moveq.l #$a,d0          ;Affecter la valeur 10
        move.l  d0,-$8000(a5)   ;à la variable b%.
        addq.l  #$1,-$7ff8(a5);Augmenter i% de 1.
        cmpi.l  #$2710,-$7ff8(a5) ;Valeur limite de la
                                 boucle déjà atteinte ?
        ble.s   L1              ;Si ce n'est pas encore
                                 le cas, revenir au début
```

Les deux premières instructions fixent la valeur de départ de la boucle i%. Vient ensuite le corps de la boucle, ici à partir du label L1, qui consiste à affecter la valeur 10 à la variable b%.

La variable de comptage i% est ensuite augmentée de 1, après quoi on examine si la valeur-limite de la boucle a déjà été atteinte. La valeur-limite était 10000, soit $2710 en hexadécimal. Si la variable de comptage est encore inférieure ou égale à cette valeur, la boucle doit être parcourue une nouvelle fois.

Une boucle dont la valeur-limite n'est pas définie par une constante, mais par exemple par une variable, présentera naturellement une structure différente, de même qu'une boucle FOR NEXT utilisant STEP.

Le corps de la boucle étant suivi de addq et la condition d'arrêt n'étant testée qu'après, i% vaudra donc 10001 après tous les parcours de la boucle. En BASIC, la convention veut effectivement que la variable de comptage soit supérieure à la valeur-limite d'une boucle après abandon de cette boucle. Le critère d'arrêt est en effet rempli lorsque le compteur est supérieur à la valeur-limite.

Important : Lorsqu'on veut traduire des programmes dans d'autres dialectes BASIC ou dans d'autres langages, il ne faut pas oublier que cette convention BASIC n'a pas de portée universelle.

Venons-en maintenant à la boucle REPEAT-UNTIL. Les lignes BASIC

```
i%=0
REPEAT
  INC i%
  b%=10
UNTIL i%=10000
```

aboutiront aux instructions assembleur suivantes :

```
        clr.l   -$7ff8(a5)          ;i%=0.
L1:     addq.l  #$1,-$7ff8(a5)      ;Augmenter i% de 1.
        moveq.l #$a,d0              ;Fixer b% sur
        move.l  d0,-$8000(a5)       ;la valeur 10.
        cmpi.l  #$2710,-$7ff8(a5)   ;Comparer 10000
                                    avec i%.
        bne.s   L1                  ;Si i% différent de
                                    10000.
```

Dans ce listing, l'instruction clr.l n'est exécutée qu'une fois, alors que toutes les autres instructions sont exécutées de nombreuses fois. Dans le listing pour la boucle FOR-NEXT, sont parcourues à de nombreuses reprises les mêmes instructions que dans la boucle REPEAT-UNTIL : un addq, un moveq, un move, un cmpi et une instruction branch avec une condition logique.

On comprend ainsi que les boucles FOR-NEXT et REPEAT-UNTIL obtiennent les mêmes temps d'exécution. Avec la boucle WHILE-WEND, ce sont également les mêmes instructions qui sont traitées à de nombreuses reprises. Les lignes

```
i%=0
WHILE i%
  INC i%
  b%=10
WEND
```

débouchent sur le texte assembleur

```
        clr.l   -$7ff8(a5)          ;i%=0.
        bra.s   L2                  ;Sauter à cmpi.
L1:     addq.l  #$1,-$7ff8(a5)      ;INC i%.
        moveq.l #$a,d0              ;Fixer b% sur
        move.l  d0,-$8000(a5)       ;la valeur 10.
L2:     cmpi.l  #$2710,-$7ff8(a5)   ;Comparer 10000
                                    et i%.
        blt.s   L1                  ;Si i% inférieur,
                                    revenir au début.
```


La conversion de la boucle WHILE-WEND en assembleur se distingue sur deux points de la conversion de la boucle REPEAT-UNTIL.

❶ La condition logique est formulée différemment. Il s'agit d'une condition d'arrêt dans la boucle REPEAT-UNTIL alors qu'il s'agit d'une "condition de continuation" dans la boucle WHILE-WEND.

❷ La boucle WHILE-WEND est une boucle a priori, c'est-à-dire que la condition est testée avant même que le corps de la boucle ne soit exécuté une première fois. C'est pourquoi, dans le programme assembleur, une instruction bra saute tout d'abord au test de la condition de boucle.

Sur tous les autres points, la réalisation des deux types de boucle est rigoureusement identique. Pour vous en tant que programmeur, l'équivalence des vitesses d'exécution signifie que vous pouvez sélectionner les types de boucle utilisés uniquement du point de vue de la structure du programme. Il n'y a donc pas lieu, pour obtenir une exécution plus rapide, d'opter pour un type de boucle se prêtant moins bien à la solution du problème qui vous est posé.

5.6.Chaînes de caractères

Cette section ne concerne pas l'optimisation de la vitesse d'exécution, mais uniquement les moyens de raccourcir le programme développé. Lorsque, dans un programme GFA-BASIC, on affecte une chaîne de caractères donnée à une variable alphanumérique, par exemple a$="Test", le compilateur place cette chaîne de caractères dans le segment DATA du programme généré.

Le compilateur essaye de comprimer autant que possible la zone des chaînes de caractères initialisées. Lorsqu'une chaîne de caractères donnée est affectée à une variable alphanumérique, le compilateur recherche tout d'abord si la nouvelle chaîne ne figure pas déjà dans la zone des chaînes de caractères définies. Si c'est le cas, il note l'adresse de la chaîne.

Exemple : supposons que vous vouliez traduire un programme en plusieurs langues et que vous placiez donc toutes les sorties de texte dans une procédure du programme BASIC pour faciliter cette traduction.

Cette procédure comprendra donc les lignes

```
a$="touche Escape"
b$="terminer le programme"
x$="Vous pouvez terminer le programme avec la
touche Escape."
```

Lors du linkage symbolique, le symbole DATASTAR sera inséré et les chaînes seront disposées à partir de ce symbole. C'est donc là qui figureront dans la mémoire nos trois chaînes de caractères, immédiatement l'une à la suite de l'autre.

Supposons toutefois que vous modifiiez l'ordre de ces trois lignes, et qu'on ait

```
x$="Vous pouvez mettre terminer le programme avec
la touche Escape."
a$="touche Escape"
b$="terminer le programme"
```

Vous ne trouverez plus maintenant, à partir de DATASTAR, que la chaîne de caractères "Vous pouvez mettre fin au programme avec la touche Escape.". La raison en est la suivante : le compilateur a tout d'abord placé cette longue ligne de texte dans le segment DATA. Il lui a ensuite fallu initialiser une chaîne avec le texte "touche Escape".

Il a alors examiné le segment DATA et a trouvé que le texte "touche Escape" faisait partie de la longue ligne de texte. Il a donc pu se contenter de noter la position de "touche Escape" dans le segment DATA, sans avoir à le réécrire. La même méthode a été suivie pour le texte "terminer le programme".

Par conséquent, si vous présentez la longue ligne après les deux autres, comme dans la première version, le compilateur ne trouvera dans le segment DATA, au moment d'initialiser cette longue ligne, que des fragments de cette ligne. Dans ce cas, il réécrira la chaîne de caractères entièrement.

Vous pouvez donc en déduire comme règle d'optimisation, du point de vue de la place mémoire, que si une chaîne de caractères contient entièrement le texte d'autres chaînes de caractères, c'est le texte de la ligne la plus longue qui doit être affecté à une variable en premier lieu.

Notez par ailleurs que les chaînes de caractères dans les lignes DATA sont placées en mémoire à partir de DATASTAR, avant toutes les chaînes qui seront affectées à des variables. Le segment DATA lui-même commence par une table des variables locales (VARTAB), suivie d'une TYPETAB de 48 octets, sous réserve toutefois que TYPE ou une affectation à travers des pointeurs (par exemple *x=3 ou SWAP *x,x%()) ait effectivement été employée.

Après VARTAB, et éventuellement TYPETAB, viennent les zones INLINE. Ce n'est qu'ensuite que figure le symbole DATASTAR, à partir duquel sont placés le contenu des lignes DATA et les chaînes initialisées.

5.7.Les variables locales et globales

Sous l'interpréteur 3.0, les opérations avec des variables locales et globales sont exécutées aussi rapidement. Il en va autrement dans un programme compilé. Une boucle FOR-NEXT parcourue 50000 fois avec une variable de comptage sur quatre octets nécessitera environ 1,315 secondes sous l'interpréteur, que la variable de comptage soit une variable locale ou une variable globale.

En format compilé, la boucle avec la variable locale nécessitera environ 0,595 secondes, alors que la boucle avec la variable globale ne nécessitera que 0,385 secondes environ.

Dans beaucoup d'autres langages le rapport entre les variables locales et globales se présente différemment. En C par exemple, les routines à variables locales sont généralement plus rapides que les mêmes routines avec des variables globales.

Compléments

LINKER

Après chaque appel de programme, la valeur de retour du programme correspondant est affichée. Zéro signifie OK, des nombres négatifs pour des erreurs (comme fichier non trouvé), des nombres positifs pour d'autres erreurs. Le compilateur redonne le nombre des instructions non compilées, le linker le nombre des symboles indéfinis et des dépassements offset.

Le linker GL affiche quelques messages d'erreurs succincts :

?	signifie symbole inconnu
+	signifie redéfinition symbole
>	signifie offset 16 octets trop grand

Ces messages ne doivent jamais apparaître dans les programmes GFA basic sans $X.

C: CALL

Vous ne pouvez lire malheureusement qu'entre les lignes du manuel du GFA BASIC que ces routines, comme les routines C sur les ordinateurs 68000 habituellement utilisés, les registres A3 jusqu'à A6 et A7 ne doivent pas changer, bien que ceci n'est aucune conséquence dans l'interpréteur. Pour C:, il n'y a presque jamais de paramètre dans D0, une particularité fortuite de l'interpréteur.

Pour pouvoir utiliser des routines assembleur qui ne suivent pas cette convention, il y a l'option compilateur SC+ et $c. Grâce à $C, les registres A3 A6 sont mis en dépôt avant le C: ou l'appel Call et ensuite retirés du dépôt.

UNPACK.GFA

Ce programme transforme le TEST.O sauvé sous forme compilée du compilateur en un format standard DR TESTX.O.

Le compilateur soutient comme la version de l'interpréteur GFA BASIC 3.07 l'instruction CURVE.

Achevé d'imprimer sur les presses de l'imprimerie IBP à Rungis (Val-de-Marne 94)
(1) 46.86.73.54 - Dépôt légal - Novembre 1990 - N° d'impression: 5414