# Partie 2

# *Le langage Stata 8.2*

# Table des matières

# 1
# Les éléments de base

## Thèmes traités

- Concepts de base de l'informatique
- C
- A
-

# 1. Introduction

Nous considérons que le logiciel est déjà installé en ayant respecté la procédure suggérée par le fabriquant. De plus, nous nous limitons à décrire l'utilisation de Stata sous l'environnement Windows. Tous nos exemples ont été réalisés sous Windows XP en utilisant la version Stata/SE 8.2.

# 2. Environnement de travail proposé

Il existe plusieurs façons de fonctionner sous Stata. Nous présentons ici la démarche que nous utilisons et que nous proposons à tout utilisateur qui veut exploiter efficacement le logiciel. Au lancement de Stata, nous obtenons l'ensemble de fenêtres suivant :

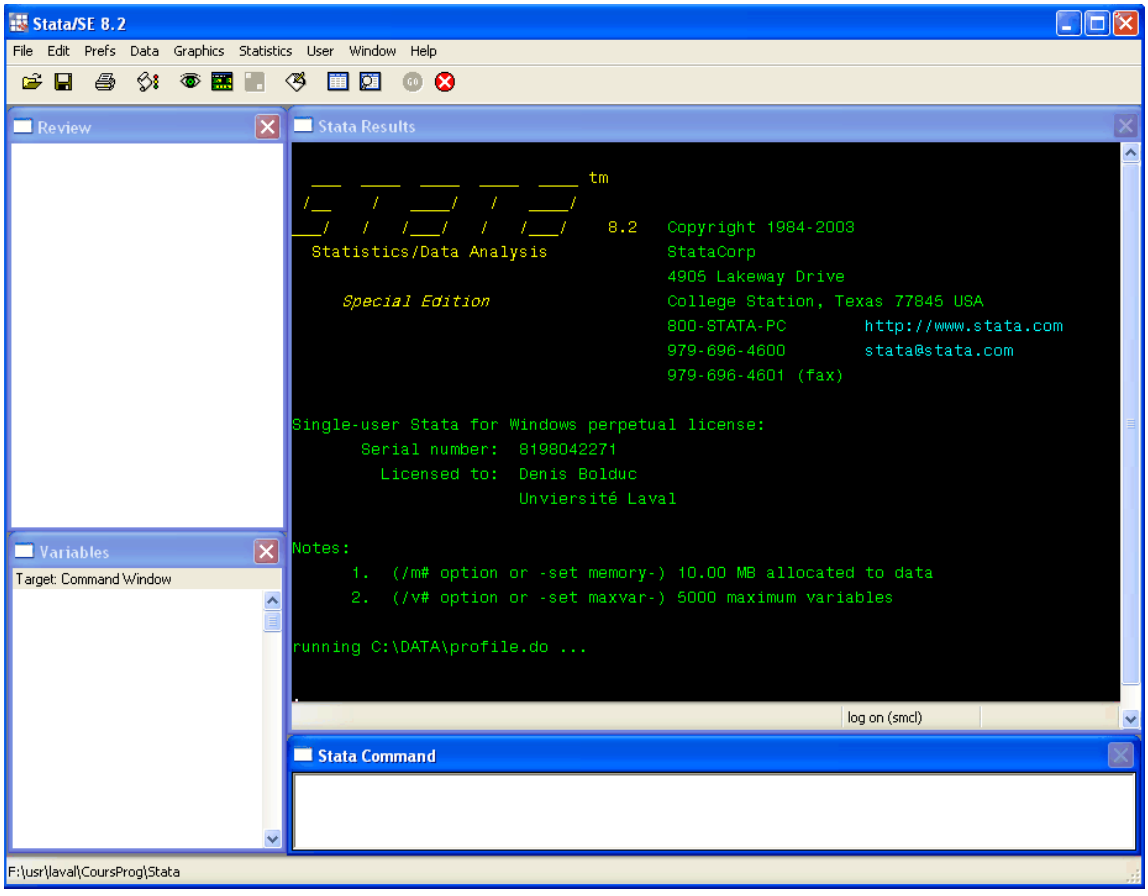



Figure 2.1 L'environnement initial de Stata 8.2.

Il comporte essentiellement quatre fenêtres, la ligne des menus et la barre d'outils. Nous nous limitons à ne décrire que certains aspects spécifiques. Pour une introduction générale à l'utilisation de l'environnement, consulter le guide de l'usager. Au lancement, toutes les fenêtres sont vides sauf celle à fond noir qui constitue un journal de tout ce qui est transigé entre l'utilisateur et Stata, soit les commandes lancées ou encore les résultats obtenus. La fenêtre **`Stata command`** est la fenêtre d'input des commandes de l'utilisateur. La fenêtre nommée **`Review`** comportera éventuellement la liste des commandes tapées par l'utilisateur alors que la fenêtre **`Variables`** donne la liste des variables stockées en mémoire. Pour l'instant aucune variable n'a encore été chargée. Nous pouvons voir à la dernière ligne de la fenêtre **`Stata Results`** l'énoncé :

```
running C:\DATA\profile.do ...
```

Au lancement de Stata, s'il existe un fichier dénommé **`profile.do`** dans le répertoire DATA sur le disque où a été installé Stata (ici le disque **`C:`** ), alors Stata l'exécute automatiquement à chaque lancement du logiciel. Ce fichier permet de personnaliser le fonctionnement de Stata. Le contenu d'un fichier type est présenté à la figure 2.2.

```
1  * Valeurs de défaut de Stata
2  set memory 420m
3  set matsize 800
4  log using "c:\data\log $S_DATE", append
5  cmdlog using "c:\data\cmdlog $S_DATE", append
6  macro define F4 `
7  macro define F5 '
8  adopath + "c:\Denis\do"
9  adopath + "c:\Denis\ado"
10 cd F:\usr\laval\CoursProg\Stata
```

Figure 2.2 Le fichier profile.do.

Le fichier **`profile.do`** est un fichier *ascii* que Stata sait interpréter. Techniquement, un fichier à extension **`.do`** est un fichier de programme Stata. Si le fichier **`profile.do`** se trouve dans le répertoire **`DATA`** par défaut, il est alors automatiquement exécuté lors du lancement de Stata. Pour éditer les fichiers programmes Stata (fichiers à extension **`.do`**), nous utilisons TextPad, tout comme pour le Java. La ligne 1, comme elle débute par une **`*`**, est une ligne de commentaire. La ligne 2 signale à Stata que l'on lui réserve 420 Mg de mémoire vive. La ligne 3 détermine la taille maximale permise pour une matrice. Comme nous le verrons, Stata permet la manipulation de tableaux en deux dimensions (matrices) qui résident en mémoire. Pour l'instant, limitons nous à connaître ce fait. La ligne 4 initialise un fichier de log dont le nom est :

```
c:\data\log LaDateActuelle
```

où **$S_DATE** est une variable système de Stata comportant la date. Ce fichier de **log** comporte exactement tout ce qui est écrit dans la fenêtre **Stata Results**, ce qui nous permet de pouvoir ensuite consulter, quand on le désire, le journal de ce qui a été fait avec Stata, cette journée-là. Le mot **append** permet d'assurer que le contenu d'une nouvelle session lancée le même jour que la précédente s'ajoute à la suite du fichier existant. La ligne 5 est semblable sauf qu'ici, seulement les commandes tapées par l'utilisateur sont stockées. Nous reviendrons plus tard sur les lignes 6 à 9. Mais pour ceux que cela intéresse, les lignes 6 et 7 définissent l'utilisation des touches **F4** et **F5** alors que les lignes 8 et 9 signalent à Stata l'endroit où sont stockés nos fichiers personnels d'utilisation générale de type **.do** et **.ado**. Pour sa part, la ligne 10 détermine le répertoire par défaut que l'on désire utiliser pour la session.

Ceux qui utilisent Stata au laboratoire informatique local 2122 / 2128 du pavillon J.A. deSève devraient utiliser le disque logique z : qui pointe automatiquement sur leur répertoire de défaut qui est monté lors qu'ils lancent une session sur l'appareil.

# *3. Syntaxe des commandes Stata*

## 3.1 Survol de l'utilisation des commandes Stata

Rappelons-nous de la base de données sur les automobiles que nous avons exploitée au chapitre 5 du cours sur le Java.

Cette base est en fait une des bases de démonstration fournie avec le logiciel Stata. Nous allons maintenant copier dans notre répertoire de travail le fichier de données Stata nommé **auto.dta** que seul Stata peut lire. Une copie du fichier est disponible sur la page web du cours. C'est en fait un fichier *binaire* qui comporte des caractères illisibles, si vous tentez de l'éditer avec un éditeur de texte *ascii* comme TextPad. Nous verrons techniquement comment générer des fichiers de données Stata ultérieurement. Pour l'instant contentons nous d'utiliser le fichier auto en tapant la commande **use auto** dans la fenêtre de commande.

Suite à cette commande, nous obtenons un fenêtrage comme à la figure 2.3 de la page précédente. Une fois la commande **use auto** tappée dans la fenêtre de commande, la touche **return** lance alors la commande. Le fenêtre **Review** est alors mise à jour en incluant la commande tapée, la fenêtre **Variables** montre alors les 12 variables du fichier de données qui sont maintenant en mémoire. Par la suite, la commande **summarize** produit la sortie présente dans la fenêtre de résultats. C'est ce que nous présentons à la figure 2.4.

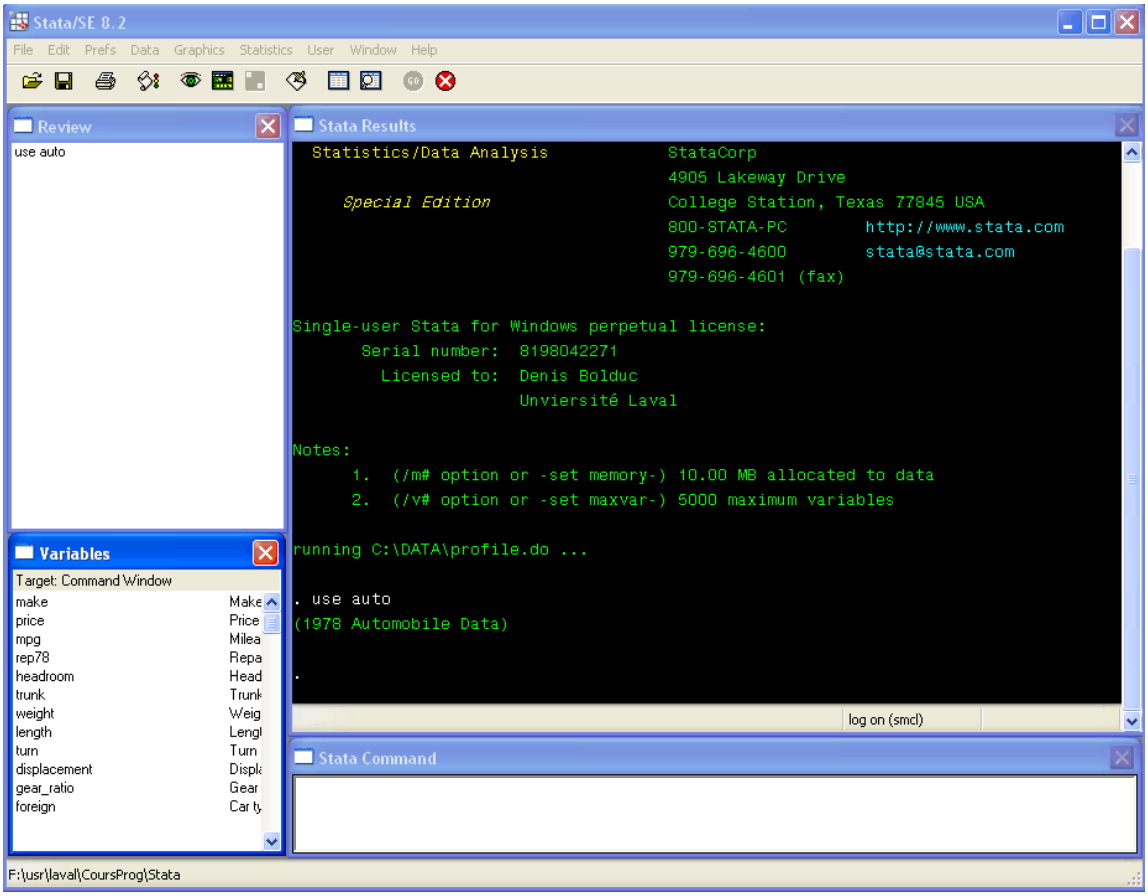



Figure 2.3 L'environnement une fois chargé le fichier Stata de données auto.

La majorité des résultats obtenus à la Figure 2.4 sont cohérents avec ceux produits avec le module BOStat développé dans le cadre du cours sur le Java. La fenêtre **`Review`** comporte maintenant deux lignes. Pour exécuter à nouveau une des deux commandes, double cliquer sur celle désirée.

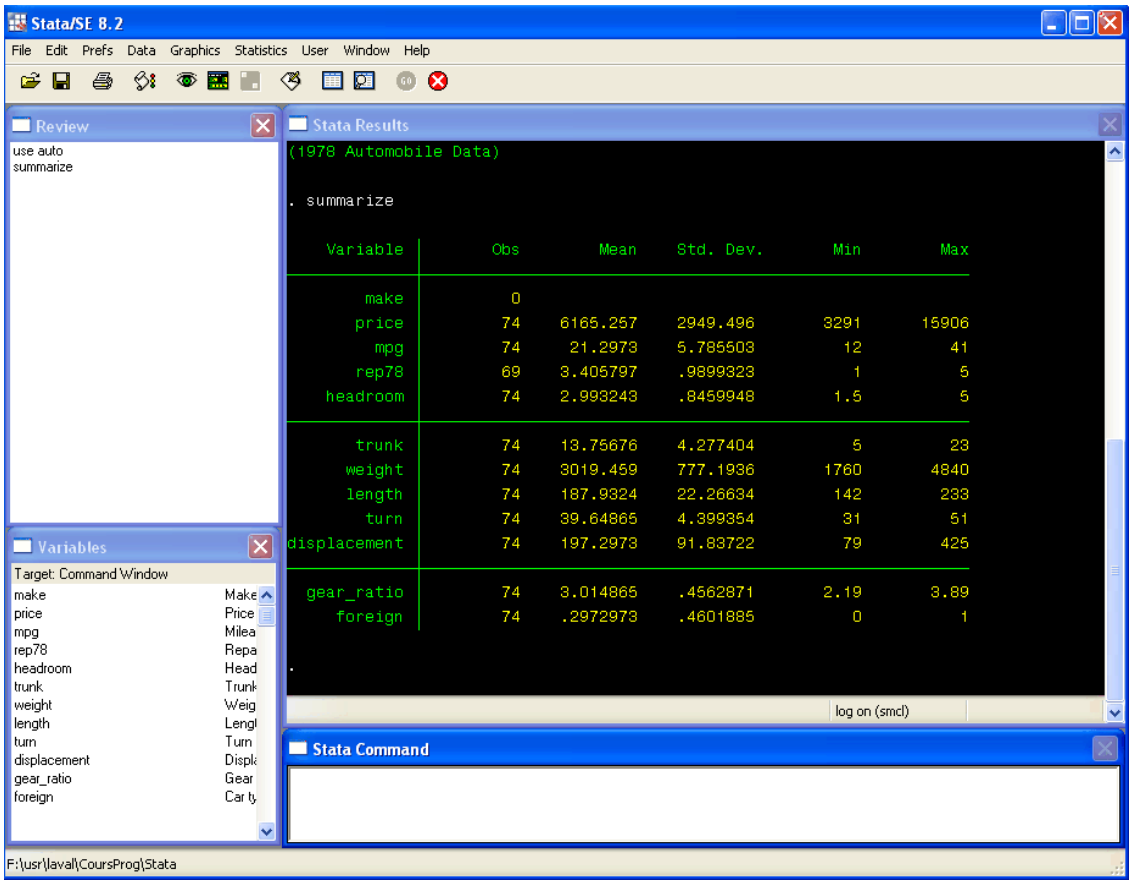


Figure 2.4 L'environnement une fois lancée la commande **`summarize`**.

## 3.2 La syntaxe générale des commandes Stata

De façon générale, toutes les commandes Stata sont appelées selon une syntaxe uniforme. Une fois que l'on connaît la forme générale, on sait alors comment appeler toute commande. La forme générale de la syntaxe est la suivante :

[**by** *varlist* **:**] *command* [*varlist*] [=*expr*] [**if** *expr*] [**in** *range*] [*weight*] [, options]

Par convention, la paire de crochets [ ] dénote le caractère optionnel de la commande qu'elle comporte. Les mots clefs sont en gras alors que les autres composantes sont décrites ci-dessous.

*varlist* est une liste comportant des noms de variables ;
*command* dénote une commande Stata ;
*expr* représente une expression algébrique ;
*range* dénote un domaine d'observations actives ;
*weight* représente une variable ou une expression détaillant la pondération à utiliser ;
*options* présent à la droite de la virgule comporte une liste d'options à appliquer.

Que ce soit pour les commandes existantes ou pour des commandes créées par des utilisateurs chevronnés, l'utilisateur peut toujours deviner la forme de la syntaxe car les développeurs de commandes se conforment toujours à ces conventions. Ceci représente une caractéristique très intéressante qui simplifie l'utilisation de Stata. Dans la ligne de syntaxe, à part la commande *command* qui est obligatoire, tout le reste est optionnel. Par exemple, dans le cas de la commande **`summarize`** utilisée précédemment, elle fut utilisée sous sa forme la plus simple. Pour des fins de démonstration, montrons ce que donne la version de commande :

```
by foreign : summarize price mpg if (rep78 > 2) , separator(0)
```

La sortie de cette commande est produite à la figure 2.5 de la page suivante. Dans cette version, nous voulons produire une analyse descriptive des variables **`price`** et **`mpg`** selon le type de modèle (**`by foreign`**) pour les cas où le nombre de réparations effectuées sur le véhicule en 1978 (**`rep78`**) est supérieure à 2. Comme option, nous demandons d'éliminer les lignes de séparations présentes à toutes les 5 variables, dans les figures précédentes. Pour connaître la syntaxe exacte et toutes les possibilités d'options d'une commande donnée, l'utilisateur peut consulter l'aide en ligne en sélectionnant l'élément **`help`** du menu et en choisissant l'item **`Stata command`**.... Taper ensuite dans la boîte de dialogue le nom de la commande qui vous intéresse.

Nous revenons maintenant brièvement sur certaines des composantes de la syntaxe générale d'une commande Stata.

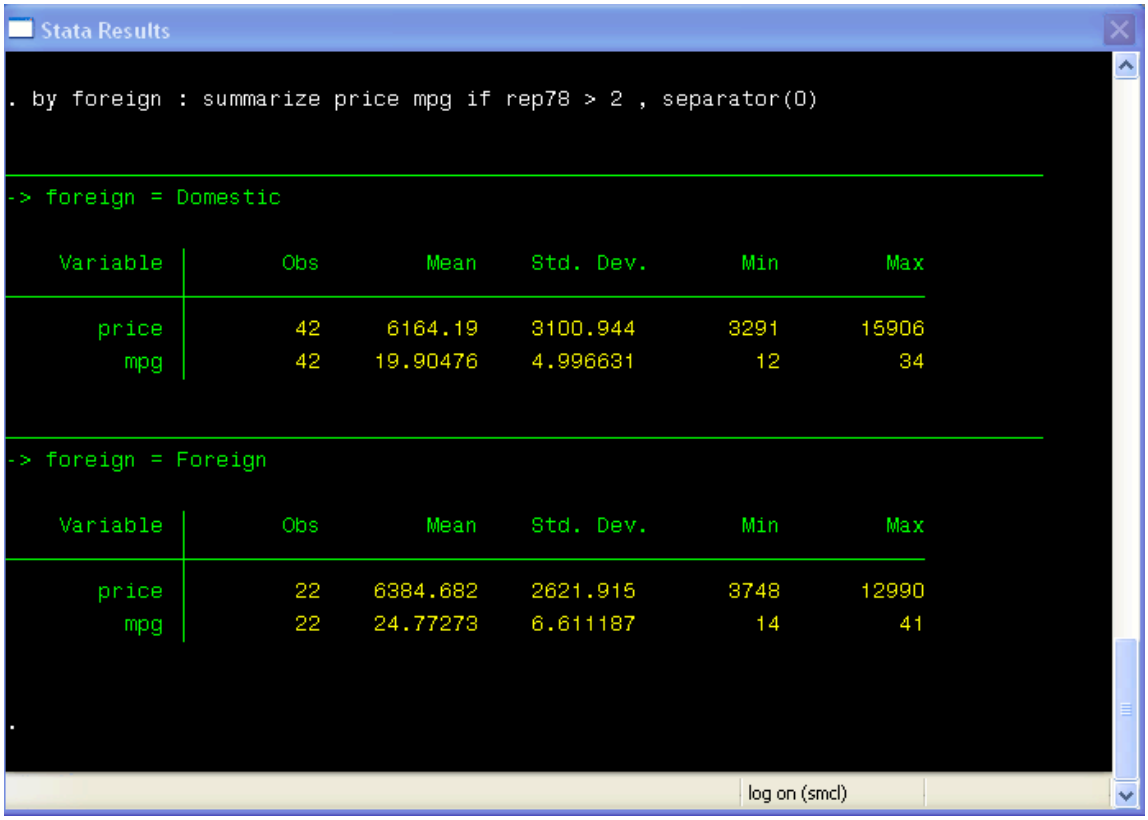



Figure 2.5 Une version spécialisée de la commande **`summarize`**.

### varlist

Lorsqu'une *varlist* est omise d'une commande Stata, implicitement toutes les variables en mémoires sont traitées, ce qui est l'équivalent d'avoir écrit le mot réservé **`_all`**.

**Remarques :**

1) *Par défaut, dès que la commande concernée vise la destruction d'une ou de plusieurs variables, il faut obligatoirement taper au complet le nom de la commande ainsi que la liste des noms sans abréviation des variables à détruire.*

2) *Pour vous simplifier la vie lors de l'entrée de noms de variables, exploiter la propriété de complétion de nome de variables de Stata. Après avoir tapé les premières lettres du nom d'une variable présente dans la base de données, la touche de tabulation permet de compléter le nom. En cas d'ambiguïté, Stata ne complète pas le nom. Dans ce cas, l'utilisateur devrait entrer quelques lettres supplémentaires de façon à lever l'ambiguïté.*

### if

Parlons maintenant de la condition **`if`** présente dans l'énoncé. En Stata, nous rencontrons deux types de **`if`**. Le premier, comme celui de l'énoncé, fait partie de la ligne de commande alors que l'autre s'apparente énormément avec les conditions **`if`** rencontrées dans les programmes en Java. Nous verrons d'ailleurs ce dernier au prochain chapitre. Attardons-nous à la clause **`if`** actuelle. Elle signale à Stata que la commande doit être effectuée sur l'ensemble des observations qui satisfont la condition. Elle doit obligatoirement se trouver à droite de la commande. La forme même du langage permet de faire des opérations sélectives sur des observations sans avoir à faire une boucle qui tourne sur les observations individuelles. La clause **`if`** conventionnelle (comme vue en Java) et les boucles en général seront réservées pour faire des opérations de programmation sur des listes de variables de la base.

### range

L'option **`in range`**, quant à elle, restreint la portée de la commande à une portion spécifique des observations. Sa spécification prend la forme $\#_1$ [/$\#_2$] avec $\#_1$ et $\#_2$ désignant des entiers positifs ou négatifs. Les entiers négatifs signifient que l'on compte à partir de la fin des observations. Par exemple, -1 réfère à la dernière observation. La première observation doit nécessairement être inférieure ou égale à la dernière. La première et la dernière observation de la base de données peuvent être dénotées par **f** et **l**, comme dans **f**irst et **l**ast. L'option **`in range`** ne peut être utilisée lorsque la commande est précédé par **`by varlist`**.

### weight

L'option weight indique le poids à attribuer à chaque observation. La syntaxe est :

[weightword = exp]

où les [ ] doivent être tapés et weightword correspond au type de poids désiré (consulter le user's guide pour une liste des poids disponibles).

## 3.3 Abréviations et complétion

Par défaut, les noms de variables et les noms de commandes peuvent être abréviées. Nous laissons alors à Stata la tâche de deviner le nom complet de l'objet qui nous

intéresse. Pour notre part, nous éviterons d'exploiter cette pratique qui apporte parfois des confusions lors de la lecture des programmes et lors de l'exécution des programmes. Au lieu d'utiliser les abréviations, nous proposons d'exploiter la fonctionnalité permettant à Stata de compléter le nom d'une variable en appliquant un **tab** lors de l'entrée du nom.

## 3.4 Conventions de nomenclatures

Pour nommer vos variables, utiliser les mêmes conventions qu'en Java. Stata est aussi sensible à la casse. Les noms suivants sont réservés :

| | | | |
|---|---|---|---|
| _all | **double** | **long** | _rc |
| _b | **float** | _n | _se |
| **byte** | if | _N | _skip |
| _coef | in | _pi | using |
| _cons | **int** | _pred | with |

Tableau 3.1 Les noms réservés.

## 3.5 varlist

Une *varlist* est une liste (ou chaîne) de noms de variables. La liste réfère soit exclusivement à des variables existantes soit exclusivement à des variables non encore existantes.

### Liste de variables existantes

L'utilisation de l'étoile * permet de représenter une liste de variables ayant une composante du nom en commun. Par exemple, dans une ensemble comportant les variables **pop1 pop2 population allo**, alors **pop*** serait un substitut pour **pop1 pop2** et **population**. De la même façon, le **?** peut remplacer un seul caractère. Donc, **pop?** est un substitut pour **pop1** et **pop2**.

L'utilisation du tiret - est aussi très utile. Comme substitut à la liste **pop1 pop2 population allo** nous pouvons utiliser **pop1-allo** si ces variables sont dans cet ordre sur l'ensemble de données. Pour découvrir l'ordre des variables, consulter la fenêtre **Variables** de Stata ou encore émettre la commande **describe**.

**Remarque :**
*On peut placer les variables de l'ensemble de données dans l'ordre alphabétique à l'aide de la commande* **`aorder`**.

### Liste de nouvelles variables

Lors de la création de nouvelles variables, le tiret permet de compléter la suite en ordre ascendant. Par exemple, **`input v2-v6`** serait automatiquement interprété par Stata comme étant **`input v2 v3 v4 v5 v6`**. Nous parlerons plus loin de la commande **`input`**.

## 4. Variables

Une variable en Stata, représente une colonne d'un tableau rectangulaire de données où chaque ligne constitue une observation. Les observations sont numérotées de 1 à **`_N`**. Les variables en Stata comportent soit des valeurs numériques soit des chaînes de caractères. Dans la version intercooled, ces chaînes sont limitées à 80 caractères alors que dans la version SE, elles peuvent comporter jusqu'à 244 caractères. Pour les nombres, on retrouve les **`int`**, **`long`**, **`float`** et **`double`**, avec lesquels nous sommes déjà familiers. Encore une fois, nous préconisons l'utilisation des **`int`** et des **`double`**.

### Valeurs manquantes

Par souci de flexibilité, Stata permet systématiquement la présence de valeurs manquantes. Par la suite, les analyses sont effectuées uniquement sur les valeurs non manquantes. Par exemple, la figure 2.4 nous signalait que **`rep78`** ne comportait que 69 valeurs non manquantes par rapport aux autres variables qui comportaient 74 observations. Les valeurs manquantes sont représentées par '**`.`**', **`.a`**, …, **`.z`**, et donc il existe 27 types de valeurs manquantes. En pratique, nous nous limiterons au premier type par défaut '**`.`**'. Les autres types sont présents pour donner plus de flexibilité à l'utilisateur. En Stata, les valeurs manquantes sont représentées par les valeurs maximales que peut prendre la variable d'un type donné. Donc, tout nombre est strictement inférieur à la valeur manquante **`.`**. Il est donc important d'être vigilant lorsque l'on teste la valeur de variables. Par exemple, **`taille > 200`** sera vrai si taille est supérieure à 200 ou si taille est manquante. Une façon d'éviter ce problème est de s'assurer explicitement que l'on n'est pas en présence d'une valeur manquante. Cela peut être fait de la façon suivante:

```
taille > 200 & taille<.
```

## 4.1 Variables numériques

Bien que les types permis soient **byte**, **int** et **long** pour les entiers et **float** et **double** pour les nombres à décimales, nous exploiterons surtout les **int** et les **double**. Par défaut, en Stata, les variables à décimales sont stockées en **float**. Par contre, toutes les opérations des calculs sont effectuées en double précision. Il sera de bonne pratique de spécifier explicitement une variable comme étant de type **double** lors de sa création.

## 4.2 Variables de type chaînes

Une chaîne **string** comporte une séquence de caractères délimitée par des guillemets. Comme exemples de chaîne valides, considérons :

```
"Salut, comment allez-vous ?"
"1.23456"
"L'étape est franchie."
"Le seigneur des anneaux, partie III."
```

La deuxième chaîne est une représentation en caractère d'un nombre. Nous verrons plus loin comment convertir une chaîne comportant un nombre en une valeur numérique utilisable.

## 4.3 Formats d'impression

Pour contrôler l'apparence à la sortie des valeurs numériques et des chaînes, Stata nous donne accès à la commande **format**. Avant de décrire formellement la commande, voyons le résultat de la commande **describe** qui permet de résumer le contenu du fichier de données chargé en mémoire. C'est ce que présente la figure 4.1.

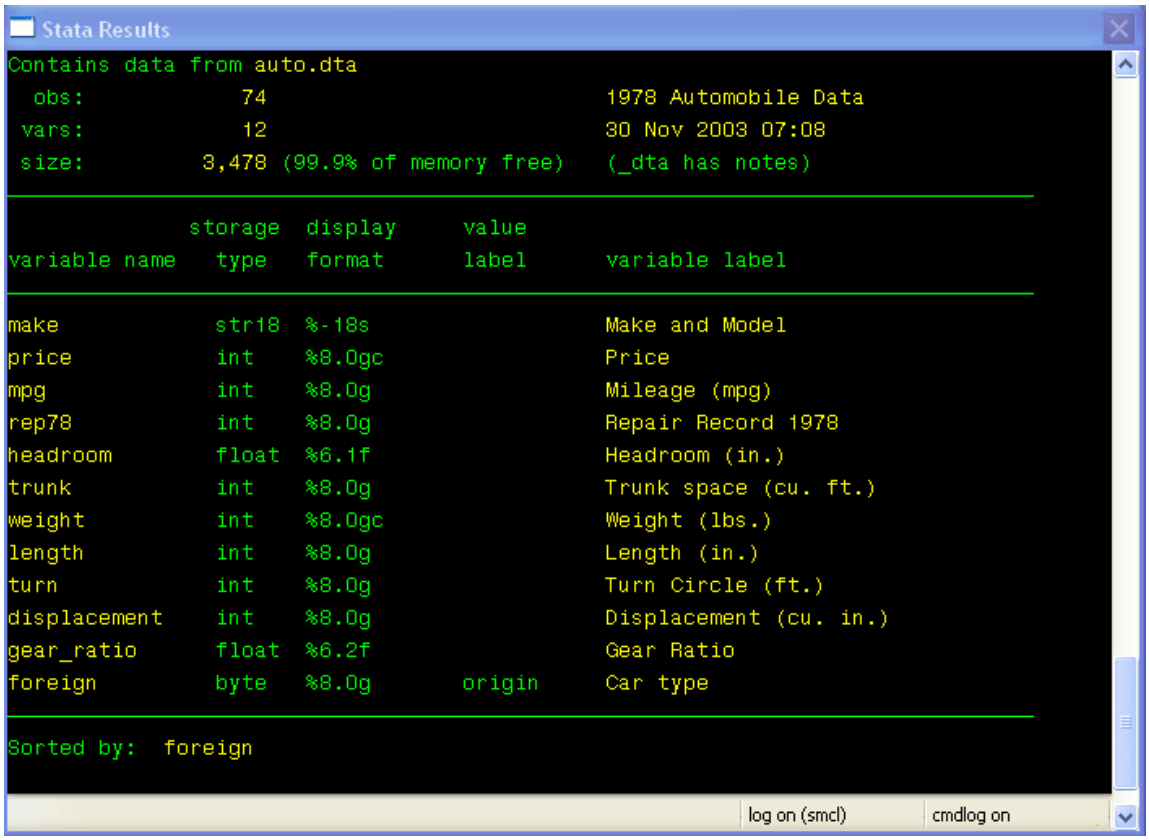


Figure 4.1 Le résultat de la commande **`describe`**.

Cette commande nous fournit plusieurs informations intéressantes. Commençons par le type de stockage des variables présentes dans la base de données. La variable **`make`** est de type **`string`** limitée à 18 caractères. La variable **`foreign`** est de type **`byte`** ce qui veut dire que cette variable est appelée à prendre que des petites valeurs entières. Les variables **`gear_ratio`** et **`headroom`** sont des variables à décimales stockées en **`float`** alors que les autres sont des variables entières (**`int`**). La séquence active de contrôle du format d'impression de chacune des variables se trouve à la colonne 3. La séquence **`%-18s`** associée à **`make`** signifie que la variable de type **`string`** sera imprimée dans un champ de 18 caractères justifié à gauche (à cause de la présence du tiret **`-`**). Pour sa part, **`price`** dont la séquence de format est **`%8.0gc`** comporte une valeur numérique avec aucune valeur à droite de la décimale qui sera imprimée dans un champ de 8 caractères justifié à droite (puisque le tiret est absent). La lettre **`g`** vient spécifier le fait que la notation à utiliser est soit décimale fixe ou scientifique, selon la valeur du nombre. Pour sa part le **`c`** détermine que l'on veut mettre une virgule pour séparer les milliers. D'avoir utilisé **`%8,0gc`** comme format aurait permis d'obtenir une virgule plutôt qu'un point pour délimiter les décimales. Utiliser la lettre **`f`** plutôt que **`g`** pour forcer la notation décimale fixe alors que **`e`** forcerait la notation scientifique. Le tableau suivant produit l'ensemble des contrôles sur le format des variables.

| Formats pour l'impression | | |
|---|---|---|
| **Tâche** | **Composante** | **Explication** |
| **Numérique** | | |
| taper d'abord | % | Pour indiquer le début du format |
| optionnellement | - | Pour justification à gauche |
| optionnellement | 0 | Pour conserver les zéros présents à gauche |
| taper un nombre | w | Déterminer la largeur du champ d'impression |
| taper | . ou , | Décimale . ou , selon le choix, le défaut est . |
| taper un nombre | d | Déterminer le nombre de décimales à mettre |
| taper ensuite | e | Pour format scientifique 1.00e+02 |
| ou | f | Pour format fixe 100.00 |
| ou | g | Pour format général (**e** ou **f**, selon la valeur) |
| optionnellement | c | Pour ajouter une virgule pour les milliers |
| **Chaîne** | | |
| taper d'abord | % | Pour indiquer le début du format |
| optionnellement | - | Pour justification à gauche |
| taper un nombre | w | Déterminer la largeur du champ d'impression |
| taper ensuite | s | Pour désigner un **string** |

Tableau 4.1 La commande **format**.

Amusez vous à changer l'apparence d'une variable à l'aide de la commande **format**. Par exemple, la commande :

```
format headroom %8,3f
```

permet maintenant de voir la variable avec un séparateur de décimal de type virgule. Pour visualiser le résultat, taper :

```
browse
```

pour lancer le **Browser** de Stata. Une autre possibilité est d'utiliser le bouton prévu à cet effet. Si vous sauvez la base suite à ces changements, ils deviennent permanents.

## 4.4 Les étiquettes de variables et de valeurs

Cette section concerne ce que nous voyons aux colonnes 4 et 5 de la figure 4.1 obtenue comme résultat de la commande **describe**. La commande **label variable** permet d'affecter à une variable une étiquette. Par exemple, on pourrait vouloir modifier l'étiquette de la variable **make** comme suit :

```
label variable make "Le modèle et la marque"
```

pour franciser l'étiquette. Pour leur part, les variables numériques représentant un nombre limité de catégories sont souvent plus facile à interpréter si l'on place une étiquette pour chaque valeur. Dans la base de données auto, la colonne 4 nous signale que la variable **foreign** est de ce type. En réalité cette variable numérique comporte deux valeurs qui sont : 0 pour *Domestic* et 1 pour *Foreign*. Pour s'en convaincre, utilisons la commande **tabulate** avec et sans l'option **nolabel** de façon à produire un tableau de fréquence. Nous obtenons alors la sortie présente à la figure 4.2.

La colonne 4 de la figure 4.1 nous disait que l'étiquette des valeurs de **foreign** porte le nom de **origin**. Ceci signifie que l'étiquette a été créée de la façon suivante :

```
label define origin 0 "Domestic" 1 "Foreign"
```

alors qu'elle a été affectée à la variable **foreign** comme suit:

```
label values foreign origin
```

Les commandes suivantes vont nous permettre de convertir en français les étiquettes de valeurs. Comme l'étiquette **origin** existe déjà, nous pouvons soit la détruire avant de la redéfinir, soit en définir une nouvelle. Les commandes suivantes font le travail :

```
label define origine 0 "Domestique" 1 "Importées"
label values foreign origine
```

Consultez le résultat avec la commande **browse**. Pour voir les accents correctement, vous pouvez toujours redéfinir la police de chacune des fenêtres de Stata en cliquant le coin supérieur gauche d'une fenêtre en sélectionnant **Font**.... Choisir ensuite la police Courrier New de taille 10.

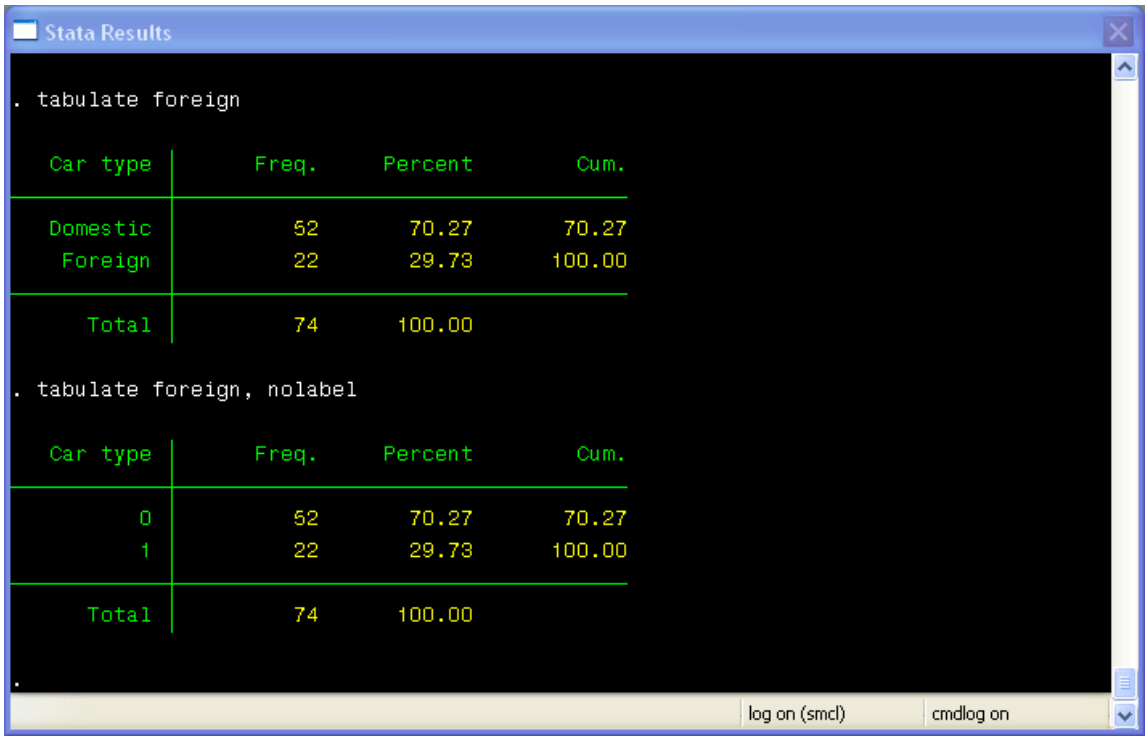



Figure 4.2 Le résultat de la commande **`tabulate`**.

## 5. Affectation

Jusqu'à maintenant, nous avons présenté des commandes permettant de modifier et d'analyser des variables déjà présentes dans l'ensemble des données. Dans cette section, nous décrivons comment créer de nouvelles variables. La commande principale d'affectation à une nouvelle variable est la commande **`generate`**. La syntaxe générale de cette commande est :

generate [type] *newvar* = *expr* [**if** *expr*] [**in** *range*]

Toute création de nouvelle variable doit être précédée de la commande **`generate`**. Par exemple :

```
generate double price2 = price * price
```

ajouterait à la base de données une variable **`price2`** stockée en double précision qui comporterait le carré de chacun des observations de la variable **`price`**. D'avoir omis le type double aurait créé par défaut une variable de type **`float`**. Pour élever une variable à une puissance, Stata réserve l'opérateur **`^`**. Donc la commande suivante :

```
generate double price2 = price ^ 2
```

aurait donné le même résultat.

**Remarque :**
*Une fois une variable créée, la seule façon de la modifier consiste à utiliser la commande* **`replace`**.

### Commande replace

Un exemple d'utilisation de la commande **`replace`** va comme suit :

```
replace price = 15 if price > 15
```

Cette opération remplacerait toutes les occurrences d'un prix supérieur à 15 par un prix égal à 15. La commande **`replace`** ne permet pas de spécifier ou de modifier le type de la variable.

# *6. Opérateurs*

Stata possède quatre classes d'opérateurs : arithmétique, chaîne, relationnel et logique. Nous décrivons maintenant chacun des types.

## 6.1 Opérateurs arithmétiques

Les opérateurs en Stata sont : **+**(addition), **-**(soustraction), *****(multiplication), /(division), **^**(puissance) et le préfixe **!**(négation). Tout opération sur une valeur manquante produit une valeur manquante.

## 6.2 Opérateurs de chaînes

Tout comme en Java, le + joue le rôle d'opérateur de concaténation de chaîne. Par exemple :

```
generate str30 strUneChaine = "Allo," + "comment allez vous ?"
```

Dans cet exemple, on ajouterait à la base de données une colonne (une nouvelle variable de type string limitée à 30 caractères) qui vaudrait **`"Allo, comment allez vous ?"`**

pour toutes les observations de la base. Si on omet le type, Stata détermine automatiquement la taille à affecter à la nouvelle variable. Il est important de réaliser que toute commande est toujours effectuée pour l'ensemble des observations de la base active de données.

## 6.3 Opérateurs relationnels

Les opérateurs relationnels sont : > (plus grand que), < (plus petit que), >= (plus grand ou égal à), <= (plus petit ou égal à), = = (égal à) et != = (non égal à). Par exemple,

```
browse if rep78 >= 2
```

permet de visualiser le sous ensemble des observations pour lesquelles au moins 2 réparations on été effectuées sur les véhicules. Les opérateurs relationnels retournent 1 pour chaque observation pour laquelle l'expression est vraie et 0 si elle est fausse.

## 6.4 Opérateurs logiques

Les opérateurs logiques sont : & (et), | (ou), et ! (non). Les opérateurs logiques interprètent toute valeur différente de 0 (incluant les valeurs manquantes) comme étant **`true`** et 0 comme **`false`**. Les opérateurs logiques permettent de combiner des expressions logiques. La clause suivante :

```
browse if rep78 >= 2 & foreign == 1
```

permet de consulter les observations concernant les véhicules importés qui ont été réparés plus d'une fois.

## 6.5 Préséance

L'ordre de préséance des opérateurs est comme suit : **`!, ^, -`**(négation), **`/,*,-, +,!=,>,<,<=,>= ,==, &`** et **`|`**. Tout comme en Java, pour des opérateurs de préséance égale, l'expression est évaluée de gauche à droite.

# *7. Fonctions et commande egen*

Stata possède plusieurs fonctions prédéfinies permettant d'effectuer des opérations mathématiques diverses à partir de variables existantes. Par exemple, les commandes :

```
generate prixExpo = exp(price)
generate prixRacineCarree = sqrt(price)
```

ajouteraient deux variables à la base de données correspondant respectivement à l'exponentielle et à la racine carrée du prix du véhicule. Le tableau 7.1 suivant présente une sélection de fonctions mathématiques utiles. Consulter l'aide en ligne concernant la commande **`function`** pour une liste complète des fonctions disponibles en Stata.

| Fonction | Explication |
|---|---|
| abs(x) | retourne la valeur absolue de x. |
| ceil(x) | retourne l'entier n tel que n-1 < x <= n. ceil(x) retourne x (non ".") si x est manquant, ce qui veut dire que ceil(.a) = .a. Voir aussi floor(x), int(x), round(x). |
| comb(n,k) | retourne la fonction combinatoire -- "k de n". Les arguments doivent être des entiers non négatifs avec n >= k >= 0. |
| exp(x) | retourne l'exponentielle de x; cette fonction est l'inverse de ln(x). |
| floor(x) | retourne l'entier n tel que n <= x < n+1. floor(x) retourne x (non ".") si x est manquant, ce qui veut dire que floor(.a) = .a. Voir aussi ceil(x), int(x), round(x). |
| int(x) | retourne la valeur entière de x en tronquant vers 0. Alors, int(5.2) = 5, et int(-5.2) = -5. int(x) retourne x (non ".") si x est manquant, ce qui veut dire que int(.a) = .a. Voir aussi round(x), ceil(x), floor(x). |
| ln(x) | retourne le logarithme naturel de x si x>0; c'est la réciproque de exp(x). |
| log(x) | retourne le logarithme naturel de x si x>0; c'est la réciproque de exp(x). |
| max(x1,x2,...,xn) | retourne le maximum de x1, x2, ..., xn. Les valeurs manquantes sont ignorées. Si tous les arguments sont manquants (.), une valeur manquante est retournée. |
| min(x1,x2,...,xn) | retourne le minimum de x1, x2, ..., xn. Les valeurs manquantes sont ignorées. Si tous les arguments sont manquants (.), une valeur manquante est retournée. |
| mod(x,y) | retourne le modulo de x par rapport à y. Si y est 0 alors une valeur manquante (.) est retournée. mod(x,y) = x - y*int(x/y). |
| round(x) | retourne x arrondi à l'entier le plus proche. round(x) = round(x,1). round(x) retourne x (non ".") si x est manquant, ce qui veut dire que round(.a) = .a. Voir aussi int(x), ceil(x), floor(x). |
| sign(x) | retourne -1 if x<0, 0 si x==0, 1 si x>0, et une valeur manquante (.) si x est manquant. |
| sqrt(x) | retourne la racine carrée de x si x >= 0. |
| sum(x) | retourne la somme cumulative de x, traitant les valeurs manquantes (.) comme des zéros. Voir l'aide sur egen pour une fonction sum alternative. |
| tanh(x) | retourne la tangente hyperbolique de x. tanh(x) = (exp(x) - exp(-x))/(exp(x) + exp(-x)) |

Tableau 7.1 Les principales fonctions mathématiques de Stata.

**Remarque** :
*Pour conserver une copie sur le disque de la version actuelle du fichier de données, utiliser la commande de sauvegarde de fichiers de données Stata suivante :*

```
save nouveaunom, replace
```

Cette commande donnerait le nom **nouveaunom.dta** au fichier sauvegardé. L'option **replace** est utilisée afin de forcer le remplacement du fichier sur le disque, au cas où un fichier du même nom serait déjà présent.

## 7.1 Fonctions classées par type

En consultant l'aide de Stata concernant la commande **function**, nous pouvons constater qu'il existe une panoplie de fonctions déjà programmées applicables dans diverses situations. Elles sont classées par type au tableau suivant :

| Type de fonction | Consulter |
|---|---|
| Mathématiques | mathfun |
| Distributions de probabilité et densités | probfun |
| Nombre aléatoires | random |
| chaînes | strfun |
| programmation | progfun |
| dates | datefun |
| séries chronologiques | tsfun |
| matrices | matfcns |

Tableau 7.2 Les principales fonctions de Stata par type.

Le tableau 7.1 résumait les fonctions mathématiques. Nous aurions pu faire un résumé similaire pour chacun des types. Ceci aurait toutefois été inutile. Nous considérons qu'il est plus important de montrer les possibilités et particularités du langage que de traduire la totalité de fonctionnalités. Une fois que l'on connaît les bases du langage, on peut toujours consulter l'aide en ligne pour découvrir des fonctions spécifiques. En consultant **probfun** avec l'aide de Stata en ligne, nous apprenons par exemple qu'il existe une fonction **norm(*variable*)** qui produit la fonction de distribution cumulative d'une normale standardisée évaluée à la valeur de la variable *variable*. La commande,

```
generate double cumNormPrix = norm(price)
```

crée une variable où chaque observation *i* de **cumNormPrix** = $\Phi$(de l'obs *i* de *price*).

## 7.2 Fonctions pour les chaînes

Dans cette section, nous présentons les principales fonctions pouvant être utilisées sur les chaînes de caractères. Dans notre base de données, la marque du véhicule est une variable de type **string** sur laquelle il serait intéressant de faire des opérations.

Présentons d'abord les fonctions applicables aux chaînes et faisons ensuite quelques tests sur notre variable **`make`**.

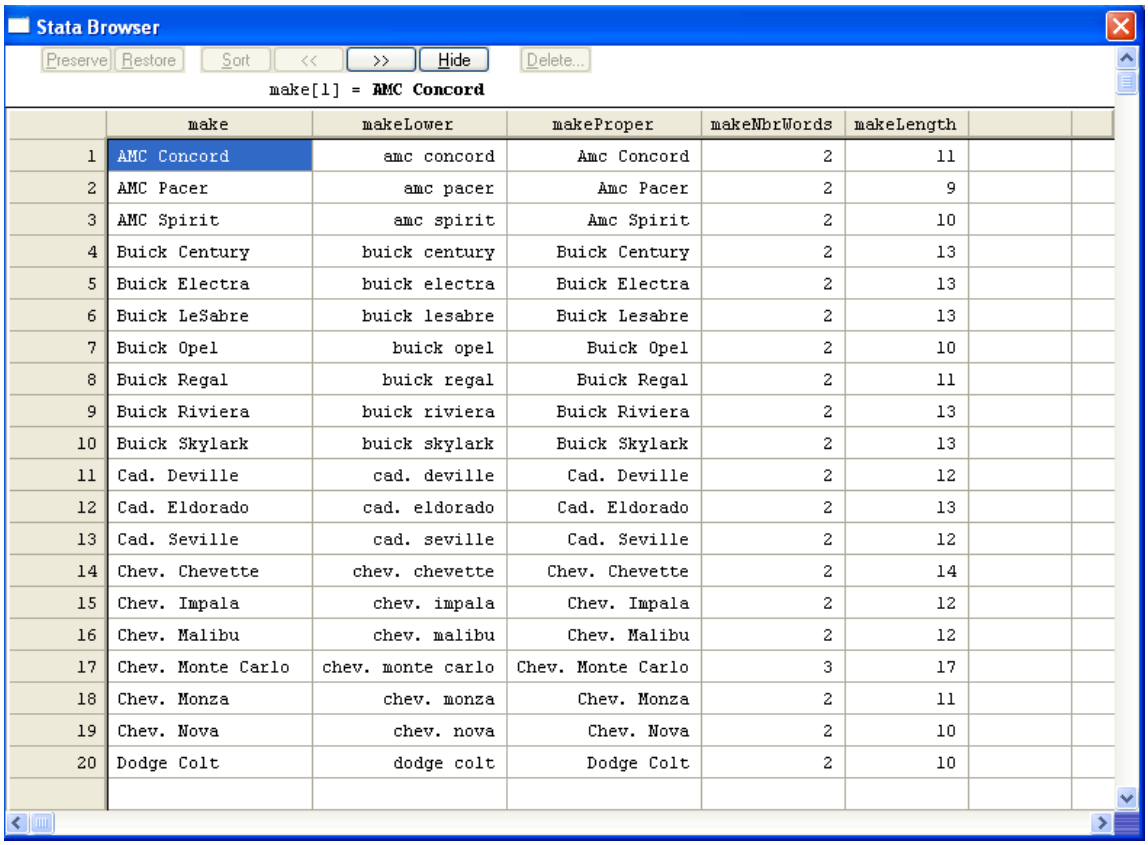
| Fonction | Explication |
|---|---|
| abbrev(s,n) | retourne le nom s, abrégé à n caractères. s peut contenir un nom, le nom d'un opérateur de série chronologique, ou toute chaîne. Si un caractère de s est un point ".", et n < 8, alors n prend la valeur 8 par défaut. Sinon, si n < 5 alors n prend la valeur 5 par défaut. abbrev() est normalement utilisé avec des noms de variables et des noms de variables avec des opérateurs de série chronologique. |
| char(n) | retourne le caractère correspondant au code ascii n si 1 <= n <= 255. Si n < 1 ou si n > 255, une valeur manquante ("") est retournée. |
| index(s1,s2) | retourne la position de s1 à laquelle s2 est trouvé en premier ou 0 si s1 ne contient pas s2. |
| indexnot(s1,s2) | retourne la position dans s1 du premier caractère de s1 qui ne se trouve pas dans s2 ou 0 si tous les caractères de s1 se trouvent dans s2. |
| length(s) | retourne la longueur de la chaîne s. |
| lower(s) | retourne la version en minuscule de la chaîne s. |
| ltrim(s) | retourne la chaîne s en tronquant les espaces en début de chaîne. |
| match(s1,s2) | Retourne la valeur 1 si s1 est semblable à s2; et 0 autrement. |
| plural(n,s) | retourne le pluriel de s en ajoutant "s" si n est différent de 1 ou -1, sinon cette fonction retourne s. |
| proper(s) | retourne une chaîne avec la première lettre en majuscules et toutes les lettres qui suivent immédiatement des caractères qui ne sont pas des lettres également en majuscules, les autres lettres sont converties en minuscules. |
| real(s) | converti s en sa valeur numérique ou retourne une valeur manquante. |
| reverse(s) | retourne s avec les lettres listées en ordre inversé. |
| rtrim(s) | retourne la chaîne s en tronquant les espaces en fin de chaîne. |
| string(n) | convertit n en chaîne. |
| string(n,"%fmt") | converti n en chaîne avec le format %fmt. |
| subinstr(s1,s2,s3,n) | retourne s1 dans laquelle les n premières occurrences dans s1 de s2 ont été remplacées par s3. Si n est manquant (.), toutes les occurrences sont remplacées. |
| subinword(s1,s2,s3,n) | retourne s1 dans laquelle les n premières occurrences dans s1 de s2 ont été remplacées par s3. |
| substr(s,n1,n2) | retourne la sous chaîne de s commençant à n1 et ayant une longueur de n2. Si n1<0, la position de départ est calculée à partir de la fin de la chaîne. Si n2 est manquant (.), la portion restante de la chaîne est retournée. |
| trim(s) | retourne la chaîne s en tronquant les espaces en début de chaîne. Ceci est le résultat de ltrim(rtrim(s)). |
| upper(s) | retourne la version en majuscules de s. |
| word(s,n) | retourne le $n^{ième}$ mot de s. Les nombres positifs indiquent que les mots sont comptés du début de s (1 = premier mot), les nombres négatifs indiquent que les mots sont comptés à partir de la fin (-1 = dernier mot). Retourne manquant ("") si n est une valeur manquante. |
| wordcount(s) | retourne le nombre de mots contenus dans s. |

Tableau 7.3 Les principales fonctions pour les chaînes.

Nous retrouvons quelques fonctions similaires à celles que nous avons utilisé en Java. Notez encore une fois qu'en Stata, une commande appliquée à une variable donnée est effectuée sur toutes les observations de la variable. Utilisons un exemple de certaines de ces fonctions sur la variable **make** de la base de données auto. Les commandes suivantes :

```
generate makeLower = lower(make)
generate makeProper = proper(makeLower)
generate makeNbrWords = wordcount(make)
generate makeLength = length(make)
```

produisent de nouvelles variables que nous visualisons en nous limitant aux 20 premières observations à l'aide de la commande : **browse make* in 1/20**. Le résultat des opérations est produit à la figure 7.1.




| | make | makeLower | makeProper | makeNbrWords | makeLength |
|---|---|---|---|---|---|
| 1 | AMC Concord | amc concord | Amc Concord | 2 | 11 |
| 2 | AMC Pacer | amc pacer | Amc Pacer | 2 | 9 |
| 3 | AMC Spirit | amc spirit | Amc Spirit | 2 | 10 |
| 4 | Buick Century | buick century | Buick Century | 2 | 13 |
| 5 | Buick Electra | buick electra | Buick Electra | 2 | 13 |
| 6 | Buick LeSabre | buick lesabre | Buick Lesabre | 2 | 13 |
| 7 | Buick Opel | buick opel | Buick Opel | 2 | 10 |
| 8 | Buick Regal | buick regal | Buick Regal | 2 | 11 |
| 9 | Buick Riviera | buick riviera | Buick Riviera | 2 | 13 |
| 10 | Buick Skylark | buick skylark | Buick Skylark | 2 | 13 |
| 11 | Cad. Deville | cad. deville | Cad. Deville | 2 | 12 |
| 12 | Cad. Eldorado | cad. eldorado | Cad. Eldorado | 2 | 13 |
| 13 | Cad. Seville | cad. seville | Cad. Seville | 2 | 12 |
| 14 | Chev. Chevette | chev. chevette | Chev. Chevette | 2 | 14 |
| 15 | Chev. Impala | chev. impala | Chev. Impala | 2 | 12 |
| 16 | Chev. Malibu | chev. malibu | Chev. Malibu | 2 | 12 |
| 17 | Chev. Monte Carlo | chev. monte carlo | Chev. Monte Carlo | 3 | 17 |
| 18 | Chev. Monza | chev. monza | Chev. Monza | 2 | 11 |
| 19 | Chev. Nova | chev. nova | Chev. Nova | 2 | 10 |
| 20 | Dodge Colt | dodge colt | Dodge Colt | 2 | 10 |

Figure 7.1 Le résultat des opérations sur la variable **make**.

## 7.3 La commande egen

En plus des fonctions, Stata donne accès à des fonctionnalités étendues disponibles via la commande nommées **`egen`**, pour « extended generate ». De façon générale, la syntaxe d'**`egen`** est la suivante :

egen [type] *newvar* = fcn(*arguments*) [**if** *expr*] [**in** *range*] [, options]

Nous recensons au tableau 7.3 les fonctions **`fcn`** les plus utiles lors de l'utilisation de la commande **`egen`**.

| Fonction | Explication |
|---|---|
| `count(exp)` | (permet by varlist) crée une constante (within varlist) contenant le nombre d'observations non-manquantes de exp. Voir aussi robs() et rmiss() ci-dessous. |
| `concat(varlist) [, format(%fmt) decode maxlength(#) punct(pchars)]` | ne peut être combiné à by. La fonction concatène varlist pour produire une variable string. Les valeurs des variables string sont inchangées. Les valeurs des variables numériques sont converties tel quel en string ou converties en utilisant la fonction format(%fmt), ou decodées avec l'option decode, dans quel cas maxlength() peut aussi être utilisé pour contrôler la longueur maximale d'étiquette utilisée. Par défaut, les variables sont ajoutées bout-à-bout: punct(pchars) peut être utilisé pour spécifier une ponctuation, comme un espace, punct(" "), ou une virgule, punct(,). |
| `diff(varlist)` | ne peut être combiné à by. Cette fonction crée une variable indicatrice valant 1 si les variables de varlist sont différentes, et 0 sinon. |
| `group(varlist) [, missing label lname(name) truncate(num)]` | ne peut être combiné à by. Cette fonction crée une variable prenant les valeurs 1, 2, ... pour le groupe formé par varlist. varlist peut contenir une chaîne, une variable numérique ou les deux. L'ordre des groupes correspond à l'ordre du sort de varlist. missing indique que les valeurs manquantes de varlist (soit . or "") doivent être traitées comme toute autre valeur lors de l'assignation des groupes, au lieu d'assigner les valeurs manquantes au groupe missing. L'option label retourne des entiers valant 1 et plus correspondant aux différents groupes de varlist in sorted order. Les integers seront étiquettés avec les valeurs de varlist, ou les valeurs d'étiquette si elles existent. lname() spécifie le nom à donner to the value label created to hold the labels, lname() implies label. L'option truncated will truncate the values contributed to the label from each variable in varlist to the length specified by the integer argument num. L'option truncated ne peut être utilisée sans l'option label. L'option truncated ne modifie pas les groupes qui ont été formés; elle modifie seulement les étiquettes. |
| `max(exp)` | (Permet by varlist:) crée une constante (within varlist) contenat la valeur maximale de exp exp. Voir aussi min(). |
| `mdev(exp)` | (allows by varlist:) crée une constante (within varlist) contenant the mean absolute deviation from the mean of exp. |
| `mean(exp)` | (permet by varlist:) crée une constante(within varlist) contenant la moyenne de exp. Voir aussi sd(). |
| `median(exp)` | (permet by varlist:) crée une constante (within varlist) contenant le médiane de exp. Voir aussi pctile(). |
| `min(exp)` | (permet by varlist:) crée une constante (within varlist) contenant la valeur minimale de exp. Voir aussi max(). |
| `rmax(varlist)` | ne peut être combiné à by. Cette fonction donne la valeur maximale (ignorant les valeurs manquantes) de varlist pour chaque observation (rangée). Si toutes les valeurs de varlist sont manquantes pour une observation, newvar vaudra missing. |
| `rmean(varlist)` | ne peut être combiné à by. La fonction crée la moyenne (rangée) des variables de varlist, ignorant les valeurs manquantes.Par exemple, si 3 variables sont spécifiées et, dans certaines observations, une variable est manquante, dans ces observations newvar will contiendra la moyenne de 2 variables variables. Les autres observations contiendront la moyenne des 3 variables. Aux observations où aucune variable n'existe, newvar vaudra missing. |
| `rmin(varlist)` | ne peut être combiné à by. La fonction donne la valeur minimale dans varlist pour chaque observation (rangée). Si toutes les valeurs de varlist sont manquantes pour une observation, newvar vaudra missing. |

… suite page suivante

| Fonction | Explication |
|---|---|
| `rsd(varlist)` | ne peut être combiné à by. La fonction crée l'écart-type (rangée) des variables de varlist, en ignorant les valeurs manquantes. Voir aussi rmean(). |
| `rsum(varlist)` | ne peut être combiné à by. La fonction crée la somme (rangée) des variables de varlist en traitant les valeurs manquantes comme 0. |
| `sd(exp)` | (permet by varlist:) La fonction crée une constante (dans varlist) contenant l'écart-type de exp. Voir aussi mean(). |
| `std(exp)` `[, mean(#)` `std(#)]` | ne peut être combiné à by by. La fonction crée la valeur standardisée de exp. Les options permettent de spécifier la moyenne et l'écart-type désiré. Les options par défaut sont mean(0) and std(1), qui produisent une variable de moyenne 0 et d'écart-type 1. |
| `sum(exp)` | (permet by varlist:) La fonction crée une constante (dans varlist) contenant la somme de exp. Voir aussi mean(). |
| `tag(varlist)` `[, missing]` | ne peut être combiné à by. It tags just one observation in each distinct group defined by varlist. When all observations in a group have the same value for a summary variable calculated for the group, it will be sufficient to use just one value for many purposes. The result will be 1 or 0, according to whether the observation is tagged, and never missing. Values for any observations excluded by either [if exp] or [in range] are set to 0 (not missing). Hence, if tag is the variable produced by egen tag = tag(varlist), the idiom if tag is always safe. missing specifies that missing values of varlist may be included. |

Tableau 7.4 Les principales fonctions de la commande egen de Stata.

## 8. Comment accéder aux résultats des commandes

Les commandes présentées jusqu'à maintenant offrent à l'utilisateur des fonctionnalités supplémentaire via les arguments de retour. On peut consulter la liste des arguments de retour d'une commande en tapant **`return list`** immédiatement après l'avoir utilisée. Par exemple, les commandes suivantes :

```
summarize price
return list
```

Produisent les résultats présentés à la figure 8.1. Cette fonctionnalité permet de pouvoir utiliser explicitement certains des calculs produits par la commande. Il faut cependant les utiliser immédiatement après la commande car sinon, il se peut que la commande suivante remplace les valeurs de retour. Par exemple, nous pourrions utiliser **`r(mean)`** pour créer une variable de prix exprimée en déviation par rapport à sa moyenne en tapant la commande suivante :

```
generate prixDeviation = price - r(mean)
```

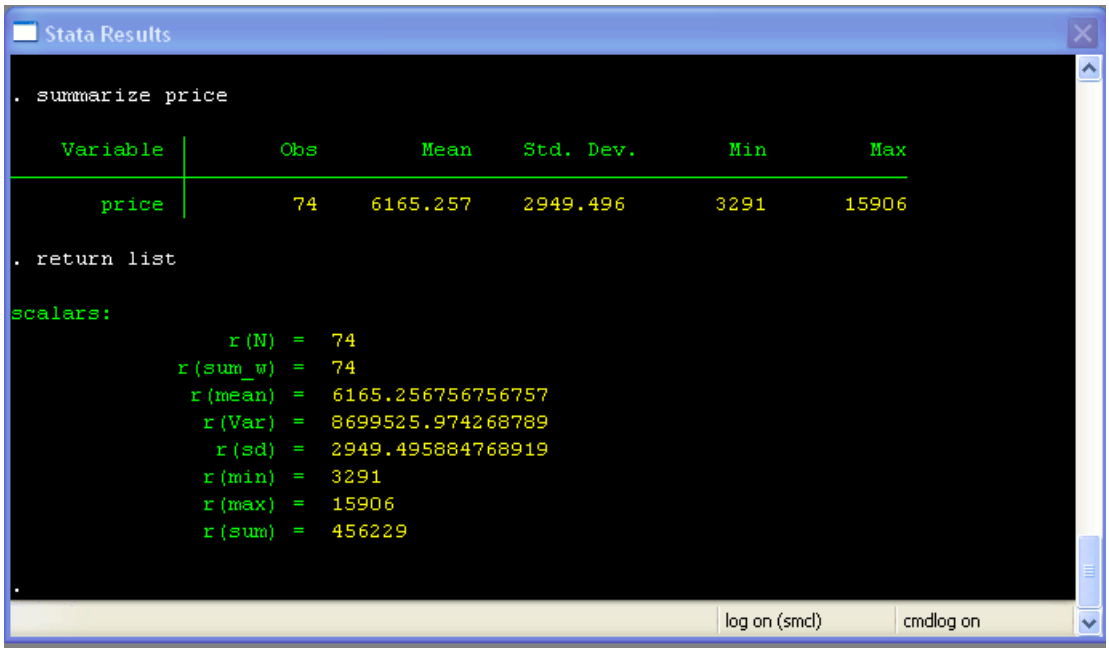



Figure 8.1 Les retours d'une commande.

## 9. Comment indicer explicitement les observations

En Stata, les mots clefs **_n** et **_N** sont importants car ils désignent respectivement l'observation active et le nombre d'observations de l'ensemble. Ce concept se généralise au sous-groupe actif dans le cas où une clause **by :** est exploitée. Dans ce cas, **_N** donnerait le nombre d'observations dans le sous-groupe actif d'observations.

Formellement, la commande :

```
generate prix2 = price ^2
```

est interprétée par Stata comme signifiant :

```
generate prix2 = price[_n] ^2
```

ce qui s'interprète comme suit, la première observation de la variable **prix2** prend la valeur de la première observation de la variable **price** mise au carré, la deuxième observation de la variable **prix2** prend la valeur de la deuxième observation de la variable **price** mise au carré, et ainsi de suite. L'utilisateur n'a le droit d'indicer explicitement l'observation d'une variable que du coté droit du signe d'affectation. Une

fois ce principe compris, il devient simple de générer des variables retardées. Par exemple la variable prix retardée d'un période s'écrirait.

```
generate price_1 = price[_n-1]
```

Nous présentons maintenant un exemple qui exploite le nom réservé **_N** et le fait que cette valeur s'ajuste selon le groupe concerné. Disons que l'on veut ajouter une variable qui donne le prix du véhicule le plus cher selon que le type de véhicule est domestique ou importé. Pour ce faire, il est important de se familiariser avec la commande **sort** qui est très utile, car elle permet de trier les observations. La liste des commandes pour effectuer cette tâche est :

```
sort foreign price
by foreign : generate prixMaxdeClasse = price[_N]
list foreign price prixMaxdeClasse in 50/54, clean abbreviate(20)
list foreign price prixMaxdeClasse in 68/74, clean abbreviate(20)
```

La sortie produite est présentée à la figure 9.1.

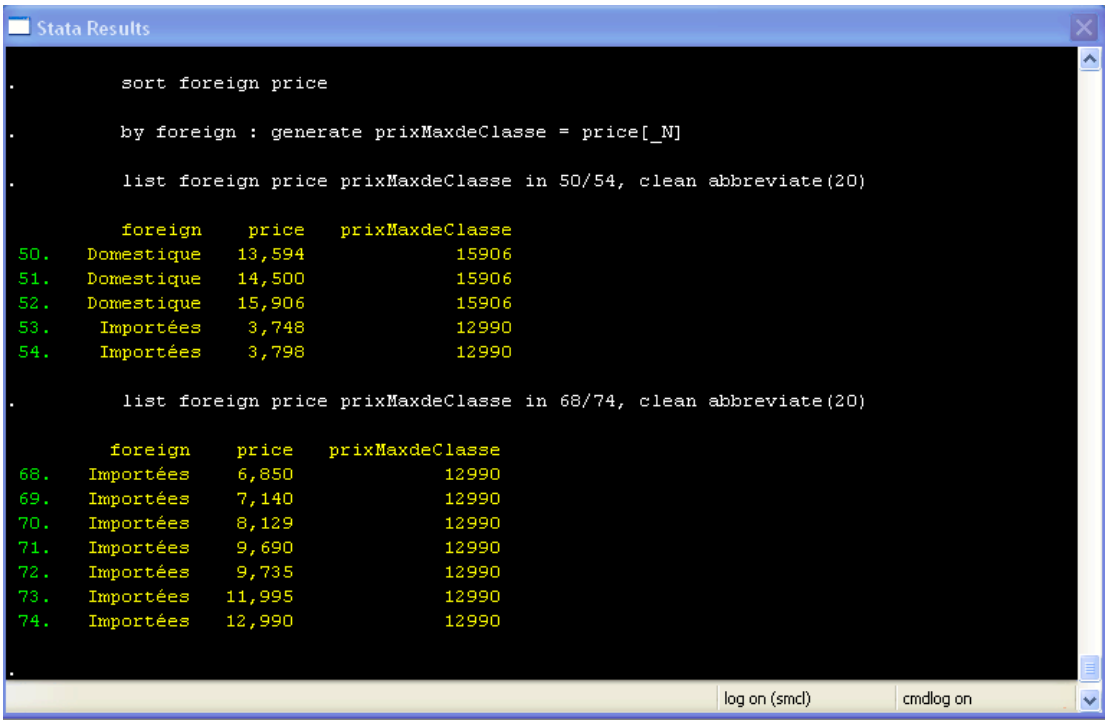


Figure 9.1 Résultats des quatre commandes précédentes.

La première ligne permet de trier les observations en ordre croissant selon le type de véhicule et le prix, la dernière observation du groupe d'un type donné correspondant au véhicule le plus cher. La seconde commande génère cette variable de prix maximal pour

le type concerné. Les deux prochaines lignes utilisent la commande **list** plutôt que la commande **browse** pour afficher les observations. On se limite au sous groupe de variables **foreign**, **price** et **prixMaxdeClasse**. Seules les observations 50 à 54 et 68 à 74 sont imprimées. Les options retenues de la commande **list** visent (**clean**) à enlever les lignes automatiquement créées par Stata et (**abbreviate(20)**) permettent d'imprimer le nom des variables jusqu'à un maximum de 20 caractères. La ligne d'observation 52 de l'ensemble nouvellement trié concerne le véhicule domestique le plus cher alors que la ligne 53 correspond au véhicule importé le moins cher.

## 10. Les fichiers do

Jusqu'à maintenant, nous avons émis les commandes une à une via la fenêtre de commandes. À la suite d'un certain nombre de commandes entrées de façon interactive, il vient un temps où l'on se demande s'il ne serait pas intéressant de regrouper les commandes dans un fichier unique qui nous permettrait d'exécuter mécaniquement une suite de commandes. Les fichiers **.do** servent exactement à ces fins. Construisons un fichier que nous appelons **faire.do** en sélectionnant dans le fichier **cmdlog** actif un sous ensemble des commandes utilisées jusqu'à maintenant.

Le fichier construit est présenté à la figure 10.1 alors que le résultat de son exécution est présenté à la figure 10.2. Pour exécuter le contenu d'un fichier **do**, taper la commande **do** suivie du nom du fichier. Dans le cas de notre exemple, nous avons tapé :

```
do faire.do
```

Un des intérêts d'utiliser un fichier **do** est qu'il est simple de faire une correction à une des commandes et de relancer ensuite la séquence des tâches. Pour les fins de notre exemple, nous repartons d'une toute nouvelle session Stata.

```
1  use auto
2  label variable make "Le modèle et la marque"
3  label define origine 0 "Domestique" 1 "Importées"
4  label values foreign origine
5  generate makeLower = lower(make)
6  generate makeProper = proper(makeLower)
7  generate makeNbrWords = wordcount(make)
8  generate makeLength = length(make)
9  save monFichierAuto, replace
10
```

Figure 10.1 Un exemple de fichier **do**.

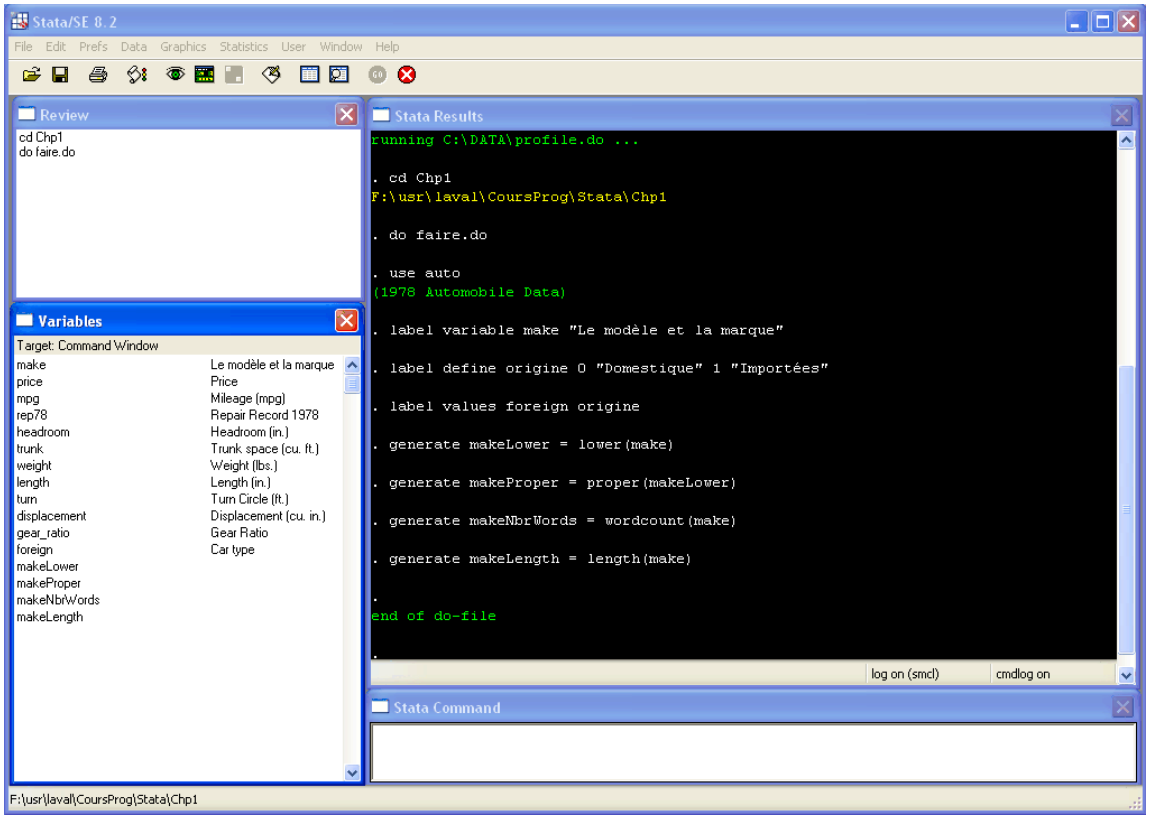



Figure 10.2 La sortie générée par le fichier faire.do.

Comme prévu, l'exécution du programme (fichier **do**) Stata crée quatre nouvelles variables qui s'ajoutent à la fin de la liste des variables existantes. Il est important de remarquer ici le fait que le fichier **faire.do** a été lancé lors d'une nouvelle session Stata. Si nous tentons immédiatement de relancer le même programme, nous pouvons remarquer à la figure 10.3 que cela plante. Tirons de ceci des enseignements importants pour nos futurs programmes.

Le message obtenu signale que Stata ne veut pas charger dans la mémoire la base de donnée auto car il y a déjà une base en mémoire. Rappelons que Stata ne travaille en principe que sur une base à la fois. Afin de forcer le chargement de la base de donnée, il existe deux solutions. La première consiste à ajouter une ligne de commande **clear** au tout début du programme, ce qui remet Stata à son état initial lors de l'ouverture. La seconde façon ajoute l'option **clear** à la commande **use** auto.

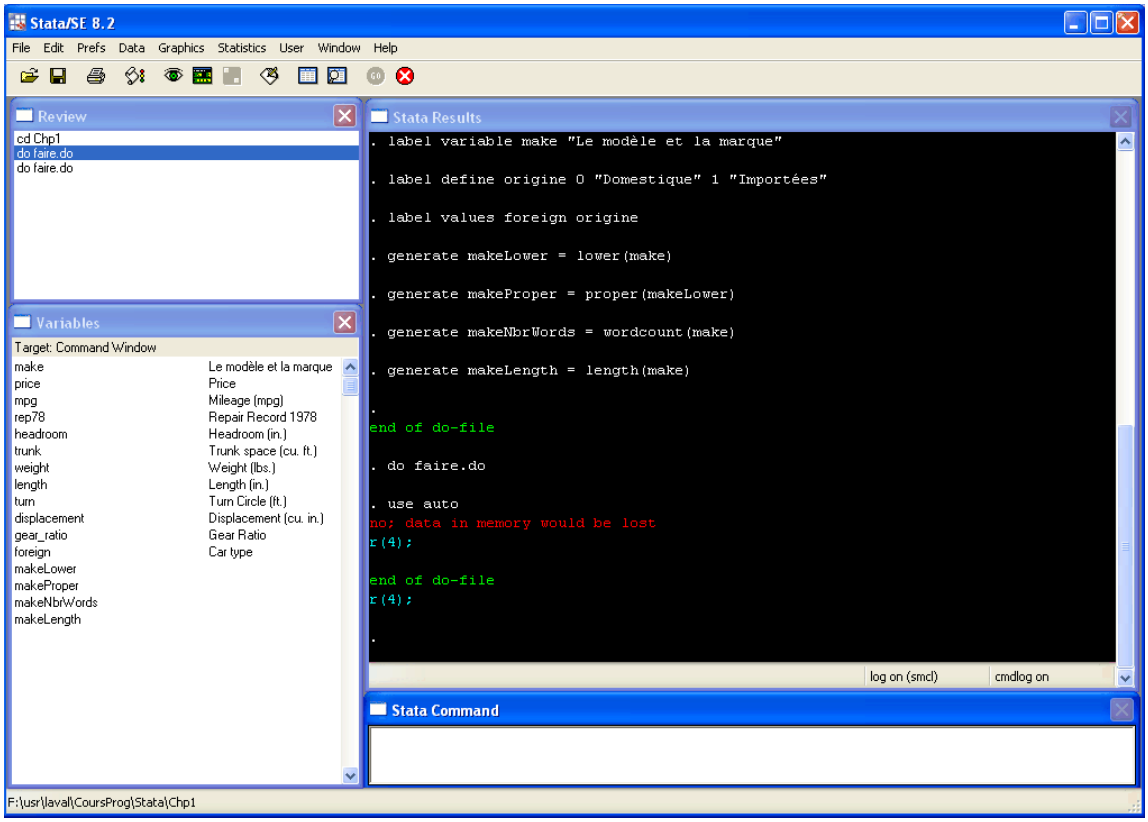



Figure 10.3 La sortie générée par le fichier faire.do exécuté une seconde fois.

La nouvelle version du programme devient alors:

```
1  use auto, clear
2  label variable make "Le modèle et la marque"
3  label define origine 0 "Domestique" 1 "Importées"
4  label values foreign origine
5  generate makeLower = lower(make)
6  generate makeProper = proper(makeLower)
7  generate makeNbrWords = wordcount(make)
8  generate makeLength = length(make)
9  save monFichierAuto, replace
10
```

Figure 10.4 Le fichier faire.do modifié.

## 10.1 Commentaires et lignes de continuation

Afin de simplifier et de rendre plus claire la lecture des programmes Stata, Il est possible d'utiliser des caractères de commentaires très semblables à ceux du Java. Nous avons déjà rencontré le caractère * qui, lorsque mis au début de la ligne met en commentaire toute la ligne. On peut aussi utiliser les caractères `//` pour ajouter des

commentaires sur une ligne de code. Tout ce qui est à droite de **//** n'est pas interprété par Stata. Pour les commentaires multi lignes, utiliser la combinaison **/*** et ***/**.

Comme nous l'avons remarqué, Stata considère qu'il y a une commande par ligne. Débutant avec la version 8.0 de Stata, il est maintenant possible d'entrer une commande sur plusieurs ligne en utilisant la séquence de caractère **///**.

### Delimit

Une autre façon d'entrer du code sur plusieurs lignes est d'indiquer la fin d'une commande grâce à un délimiteur comme un point virgule, par exemple. Pour activer un tel délimiteur, utiliser la commande **#delimit ;** . Terminer ensuite chaque commande avec un **;.** Pour retourner à l'état initial où un changement de ligne détermine une fin de commande, utiliser la commande **#delimit cr**, pour « carriage return ». Le fichier précédent pourrait être modifié comme suit :

```
#delimit ;
use auto, clear ;
label variable make "Le modèle et la marque" ;
label define origine 0 "Domestique"
                     1 "Importées" ;
label values foreign origine ;
#delimit cr

   // À partir de ce point nous termininons les commandes avec un "cr".
generate makeLower = lower(make)
generate makeProper = proper(makeLower)
generate makeNbrWords = wordcount(make)
generate makeLength = length(make)
save monFichierAuto, replace

```

Figure 10.5 Utilisation de #delimit.

## 11. Les matrices

Jusqu'à maintenant, nous avons travaillé exclusivement avec des variables. En Stata, il est aussi possible d'utiliser des objets complètement indépendants que l'on appelle matrices. Bien que nous parlerons ultérieurement plus formellement de l'utilisation des matrices en Stata, nous nous devons ici d'introduire brièvement le concept de matrice. Toute opération impliquant des matrices est précédée de la commande **matrix**. Un cas particulier des matrices est le scalaire (**scalar**) qui est en fait une matrice de dimension $(1\times1)$. Les commandes suivantes produisent la sortie de la figure 11.1.

```
matrix Identite = I(5)
matrix dir
matrix list Identite
```

La première commande génère une matrice identité de taille 5. La seconde commande fait la liste de tous les objets **`matrix`** présentement en mémoire. Dans le cas présent, seule la matrice **`Identite`** est présente. La dernière commande produit une sortie imprimée de son contenu.

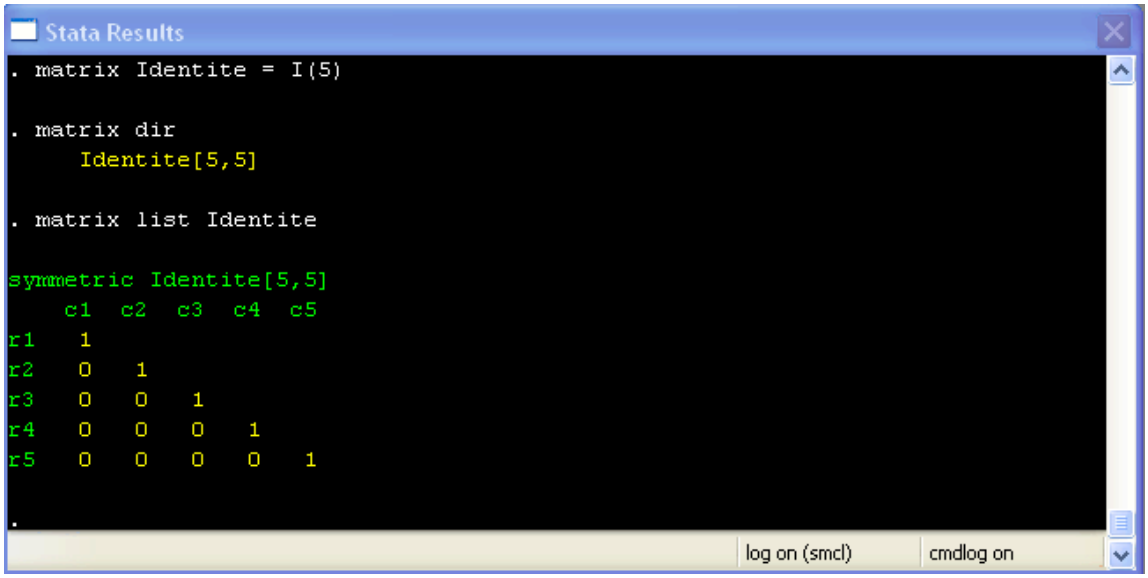



Figure 11.1 Les matrices en Stata.

L'exemple qui suit exploite la fonction matricielle **`trace(`*`matrice`*`)`** qui retourne dans un scalaire la valeur de la trace de la matrice nommée ***`matrice`*** passée en argument.

```
scalar traceI = trace(Identite)
scalar dir
```

La sortie qui découle de ces deux commandes est produite à la figure suivante :

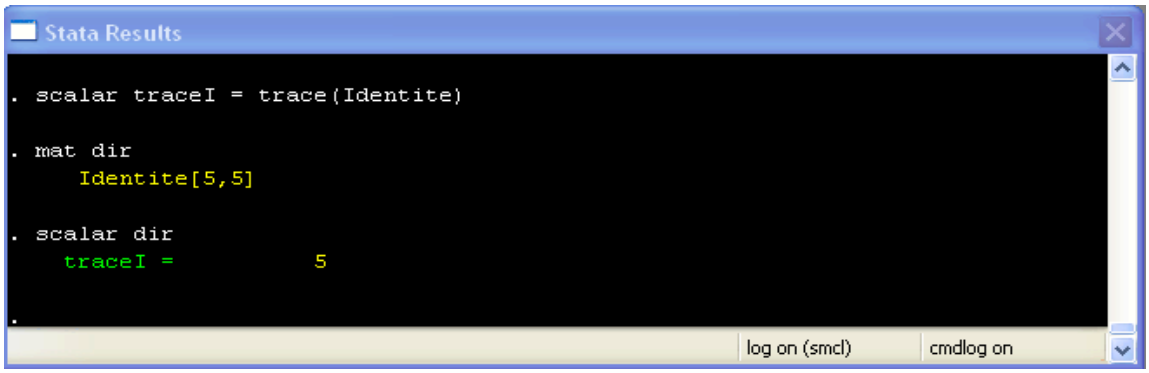



Figure 11.2 Les scalaires en Stata.

Stata offre plusieurs possibilités d'interaction entre les objets matriciels et les variables de la base de données active. Nous avons vu précédemment comment accéder aux retours des commandes Stata. En fait, certaines commandes, en plus de retourner des éléments scalaires, peuvent aussi retourner des vecteurs et matrices. Par exemple, la commande **`regress`** retourne le vecteur des paramètres du modèle ainsi que la matrice des variances et covariances. Pour ceux qui ne connaissent pas le concept de régression, cette dernière phrase peut paraître nébuleuse, mais pour les autres, elle signifie beaucoup. Comme nous le verrons ultérieurement, les **`returns`** qui incorporent des vecteurs ou matrices sont dit **`ereturns`**, pour retours d'estimation. On les consulte en tapant **`ereturn list`**.

# *12. Les bases de données*

Dans ce chapitre, nous nous sommes limités jusqu'à maintenant à utiliser une base unique de données laquelle était déjà une base Stata. Dans cette section, nous présentons tout d'abord les diverses commandes permettant de lire des bases de données de toute source afin de pouvoir ensuite les utiliser en Stata. Nous discutons ensuite des commandes Stata permettant de transférer une base de données de type Stata vers d'autres logiciels. Nous terminons par des commandes permettant combiner des bases de données.

**Remarque :**
*La façon la plus directe d'échanger les bases de données entre Stata et les autres logiciels statistiques existants consiste à utiliser le logiciel stat/transfer disponible au lien* http://www.stattransfer.com/.

## 12.1 Importer des données en Stata

Nous traitons spécifiquement de deux commandes pour lire des données en Stata, les commandes **`infile`** et **`insheet`**.

### Infile

On utilise la commande **`infile`** lorsque le fichier source est en *ascii* et que les champs sont séparés par soit un ou des espaces ou encore des virgules. Si le fichier ne contient que des valeurs numériques, rien n'est plus simple. La syntaxe à utiliser est décrite ci-dessous. Dans le cas où certaines colonnes comportent des caractères autres que numérique, (ex : pour des variables de type chaînes), il faut alors s'assurer que les chaînes sont entre guillemets à moins qu'elle ne soient composées que de lettres contiguës. Une bonne pratique consiste à toujours les délimiter avec des guillemets.

La version de syntaxe de la commande **`infile`** que nous décrivons est :

infile *varlist* **using** *filename* [, clear ]

La syntaxe est plus générale que cela, sauf que nous croyons qu'il est plus simple de lire la totalité de la base pour ensuite faire les traitements. Le mot-clef *filename* représente le nom du fichier *ascii* comportant les observations de la base à importer. Le *varlist* correspond à la liste des noms à donner à chacune des variables à associer aux colonnes du fichier. Précéder les noms de variables de type chaîne par le type **`str??`**, en spécifiant le nombres de caractères de la chaîne. Englober le nom de la variable ou liste de variables entre parenthèses. Il est important de s'assurer que *varlist* contient autant d'éléments qu'il y a de colonnes (champs) sur la base de données. L'option clear vise à signaler à Stata qu'il est correct de libérer la mémoire afin de charger une nouvelle base en mémoire.

Exemple :

Soit le fichier *ascii* suivant qui représente un sous-ensemble de la base **`auto.dta`** qui concerne les 14 premières observations des quatre premières variables. Nous avons fait ressortir les caractères invisibles d'espacement et de retour de ligne pour les fins de l'exposé.

```
 1  "AMC Concord"     4099  22  3↓
 2  "AMC Pacer"       4749  17  3↓
 3  "AMC Spirit"      3799  22  0↓
 4  "Buick Century"   4816  20  3↓
 5  "Buick Electra"   7827  15  4↓
 6  "Buick LeSabre"   5788  18  3↓
 7  "Buick Opel"      4453  26  0↓
 8  "Buick Regal"     5189  20  3↓
 9  "Buick Riviera"  10372  16  3↓
10  "Buick Skylark"   4082  19  3↓
11  "Cad. Deville"   11385  14  3↓
12  "Cad. Eldorado"  14500  14  2↓
13  "Cad. Seville"   15906  21  3↓
14  "Chev. Chevette"  3299  29  3↓
```

Figure 12.1 Le fichier ascii autoPourInfile.txt.

La commande suivante permet de charger en mémoire cet ensemble de données.

```
infile str20 ( make )  price mpg rep78  using autoPourInfile.txt,clear
```

Assurez-vous de bien indiquer à Stata quelles variables sont de type chaîne en enrobant le(s) nom(s) du type **`str??`**. Une fois en mémoire, nous pouvons sauver la version Stata de la base à l'aide de la commande habituelle **`save nomfichier, replace`**.

### Insheet

La commande **insheet** est plus simple à utiliser qu'**infile**. La commande **insheet** est spécifiquement utilisée pour importer des données qui proviennent d'un tableur. Il est alors permis de mettre sur la première ligne le nom de chaque variable. Le plus simple est de séparer les champs par un code de tabulation. L'exemple suivant montre la façon naturelle d'exporter des données d'excel vers des fichiers *ascii* avec tabulations.

Exemple :

Le fichier **autoPourInsheet.txt** a été créé en y copiant le contenu de la zone de cellules extraite de la feuille excel auto.xls contenant la totalité de la base. Le fichier obtenu est présenté à la figure suivante :

```
 1  make>             price>          mpg>            rep78
 2  AMC Concord>      4099>           22>             3
 3  AMC Pacer>        4749>           17>             3
 4  AMC Spirit>       3799>           22>             0
 5  Buick Century>    4816>           20>             3
 6  Buick Electra>    7827>           15>             4
 7  Buick LeSabre>    5788>           18>             3
 8  Buick Opel>       4453>           26>             0
 9  Buick Regal>      5189>           20>             3
10  Buick Riviera>    10372>          16>             3
11  Buick Skylark>    4082>           19>             3
12  Cad. Deville>     11385>          14>             3
13  Cad. Eldorado>    14500>          14>             2
14  Cad. Seville>     15906>          21>             3
15  Chev. Chevette>   3299>           29>             3
```

Figure 12.2 Le fichier ascii autoPourInsheet.txt.

Les codes de tabulation son représentés par un signe >. Une fois un tel fichier créé, la commande suivante

```
insheet using autoPourInsheet.txt, clear
```

permet de générer la base Stata. Dans le cas présent, la tâche de l'utilisateur est simplifiée, car Stata détecte automatiquement le type des différentes variables. Les noms des variables sont lus à la ligne 1 du fichier.

## 12.2 Exporter des données hors de Stata

La façon la plus simple d'exporter des données d'une base Stata, à part via l'utilisation de Stat/Transfer, consiste à utiliser la commande **`outsheet`**.

La syntaxe typique de cette commande est :

outsheet [*varlist*] **using** *filename* [**if** *expr*] [**in** *range*] [, replace ]

Dans le cas de l'exemple utilisé dans la section précédente, la commande :

```
outsheet using autoVersAscii.asc, replace
```

créerait un fichier identique en tout point au fichier présent à la figure 12.2.

## 12.3 Combiner des bases de données Stata

Dans cette section nous montrons comment ajouter des observations à une base de données existante à l'aide de la commande **`append`**. Nous montrons ensuite comment ajouter des variables d'une autre base à une base active à l'aide de la commande **`merge`**. Tout comme pour les commandes **`infile`** et **`insheet`**, nous démontrons l'utilisation de ces commandes en poursuivant l'exemple entamé dans la section précédente.

### append

La syntaxe de cette commande est :

append **using** *filename* [, nolabel keep(*varlist*) ]

Exemple :

Ajoutons aux données du fichier présent à la figure 12.2 l'ensemble de données du fichier **`autoPourAppend.txt`** dont le contenu est présenté à la figure 12.3. Ce fichier créé que pour fins de démonstration de la commande contient les observations 15 à 23 des quatre premières variables de la base de données complète.

```
make              price          mpg            rep78
Chev. Impala      5705           16             4
Chev. Malibu      4504           22             3
Chev. Monte_Carlo 5104           22             2
Chev. Monza       3667           24             2
Chev. Nova        3955           19             3
Dodge Colt        3984           30             5
Dodge Diplomat    4010           18             2
Dodge Magnum      5886           16             2
Dodge St. Regis   6342           17             2
```

Figure 12.3 Le fichier ascii autoPourAppend.txt.

Les commandes de la figure 12.4 suivante produisent le fichier désiré, lequel comporte les 23 observations.

```
insheet using autoPourInsheet.txt, clear
save auto1, replace

insheet using autoPourAppend.txt, clear
save auto2, replace

use auto1, clear
append using auto2

save monFichierAuto, replace
```

Figure 12.4 Le fichier de commande pour le **append** .

### merge

Il arrive souvent que l'on doive ajouter des variables à une base de données en voulant respecter un critère de compatibilité. Pensons à un exemple simple où dans un fichier nous possédons de l'information sur l'adresse civique de diverses personnes alors que dans un autre fichier, nous trouvons, pour ces mêmes personnes, leur numéro de téléphone et leur date de naissance. Donc, le fichier A comporte les variables **`nom`** et **`adresse`** alors que le fichier B contient les variables **`nom`**, **`numeroTel`** et **`dateNaissance`**. Notre objectif est donc de créer un nouveau fichier C qui partant du fichier A est créé en y ajoutant le contenu du fichier B en utilisant **`nom`** comme variable commune permettant de combiner les bonnes lignes. Dans ce cas, la variable **`nom`** joue le rôle de clef. Une clef peut être formée par une combinaison de variables. Avant de faire une fusion (un merge), il est important de s'assurer que les deux ensembles de données fichier A et fichier B sont tous deux triés selon la même clef. La syntaxe typique de cette commande est :

merge [*varlist*] **using** *filename* [, keep(*varlist*) nokeep]

Les détails complets concernant cette commande peuvent s'obtenir avec l'aide en ligne. Partant d'un fichier déjà chargé en mémoire (le fichier A connu sous le nom de fichier **master**) et trié selon *varlist*, indiquer dans *filename* le nom du fichier B (connu sous le nom de fichier **using**) à fusionner. L'option keep(*varlist*) permet de fournir la liste des variables que l'on veut ajouter au fichier en mémoire. L'omettre signifie que l'on veut ajouter toutes les variables de l'ensemble **using**. Pour sa part, l'option **nokeep** signifie que toute observation présente dans le **using** qui n'est pas présente dans le fichier **master** n'est pas incorporée dans l'ensemble résultant. Par défaut, la commande **merge** crée une nouvelle variable catégorique nommée _merge qui donne pour chaque observation le statut de la fusion de chaque observation. Cette variable est définie comme suit :

| _merge | Explication |
|---|---|
| 1 | obs. vient du fichier master |
| 2 | obs. vient du fichier using |
| 3 | obs. présentes dans les fichiers master et using |

Tableau 12.1 Signification du code _merge.

Il est considéré être de très bonne pratique de produire une tabulation de cette variable, **tabulate _merge**, de façon à verifier que la fusion a été effectuée correctement. Si toutes les observations du fichier A trouvent une correspondance dans le fichier B, alors seuls des codes **_merge==3** seraient produits. On peut ensuite examiner les cas inattendus à l'aide de la commande **browse if _merge /= 3**. Une fois certains que tout est correct, détruire la variable _merge à l'aide de la commande **drop _merge**.

Exemple :

Ajoutons aux données du fichier présent à la figure 12.2 les variables du fichier **autoPourMerge.txt** dont le contenu est présenté à la figure 12.5.

```
 1  make              headroom    trunk    weight    length
 2  AMC Concord       2.5         11       2930      186
 3  AMC Pacer         3.0         11       3350      173
 4  AMC Spirit        3.0         12       2640      168
 5  Buick Century     4.5         16       3250      196
 6  Buick Electra     4.0         20       4080      222
 7  Buick LeSabre     4.0         21       3670      218
 8  Buick Opel        3.0         10       2230      170
 9  Buick Regal       2.0         16       3280      200
10  Buick Riviera     3.5         17       3880      207
11  Buick Skylark     3.5         13       3400      200
12  Cad. Deville      4.0         20       4330      221
13  Cad. Eldorado     3.5         16       3900      204
14  Cad. Seville      3.0         13       4290      204
15  Chev. Chevette    2.5         9        2110      163
16
```

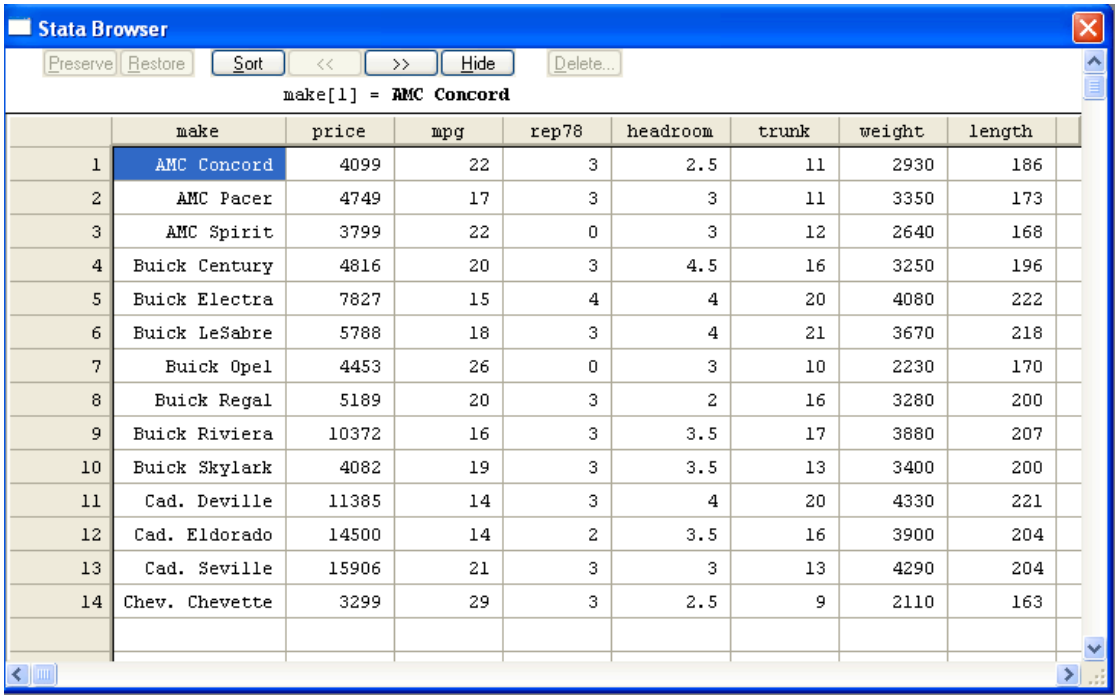
Figure 12.5 Le fichier ascii autoPourMerge.txt.

Les lignes de programme permettant de faire la fusion de ces données se retrouvent à la figure 12.6. Le fichier tel qu'il se trouve une fois la fusion effectuée est présenté à la figure 12.7.

La ligne 2 du programme assure que l'ensemble est trié selon la variable **`make`**. La ligne 6 fait de même pour l'ensemble qui sert de fichier **`using`** dans la commande **`merge`**. La ligne 11 produit une tabulation qui permet de vérifier si tout s'est bien passé (lorsque la variable _merge ne prend que la valeur 3).

```
1  insheet using autoPourInsheet.txt, clear
2  sort make
3  save auto1, replace
4
5  insheet using autoPourMerge.txt, clear
6  sort make
7  save auto2, replace
8
9  use auto1, clear
10 merge make using auto2, nokeep
11 tabulate _merge
12 drop _merge
13 save monFichierAuto, replace
14
```

Figure 12.6 Le fichier de commande pour le **merge** .




| | make | price | mpg | rep78 | headroom | trunk | weight | length |
|---|---|---|---|---|---|---|---|---|
| 1 | AMC Concord | 4099 | 22 | 3 | 2.5 | 11 | 2930 | 186 |
| 2 | AMC Pacer | 4749 | 17 | 3 | 3 | 11 | 3350 | 173 |
| 3 | AMC Spirit | 3799 | 22 | 0 | 3 | 12 | 2640 | 168 |
| 4 | Buick Century | 4816 | 20 | 3 | 4.5 | 16 | 3250 | 196 |
| 5 | Buick Electra | 7827 | 15 | 4 | 4 | 20 | 4080 | 222 |
| 6 | Buick LeSabre | 5788 | 18 | 3 | 4 | 21 | 3670 | 218 |
| 7 | Buick Opel | 4453 | 26 | 0 | 3 | 10 | 2230 | 170 |
| 8 | Buick Regal | 5189 | 20 | 3 | 2 | 16 | 3280 | 200 |
| 9 | Buick Riviera | 10372 | 16 | 3 | 3.5 | 17 | 3880 | 207 |
| 10 | Buick Skylark | 4082 | 19 | 3 | 3.5 | 13 | 3400 | 200 |
| 11 | Cad. Deville | 11385 | 14 | 3 | 4 | 20 | 4330 | 221 |
| 12 | Cad. Eldorado | 14500 | 14 | 2 | 3.5 | 16 | 3900 | 204 |
| 13 | Cad. Seville | 15906 | 21 | 3 | 3 | 13 | 4290 | 204 |
| 14 | Chev. Chevette | 3299 | 29 | 3 | 2.5 | 9 | 2110 | 163 |

Figure 12.7 Le fichier suite à la fusion.

# 2
# Les bases de la programmation

## Thèmes traités

- Concepts de base de l'informatique
- C
- A

# 1. Introduction

Nous avons vu au chapitre précédent comment utiliser des commandes Stata, soit en tapant une commande dans la fenêtre de commande de Stata ou encore en tapant une suite de commandes dans un fichier *ascii* nommé avec une extension **.do** que nous exécutons ensuite en tapant **do *nomfichier*.do**.

Stata est aussi programmable, en ce sens qu'un utilisateur a la possibilité d'étendre le langage Stata en pouvant définir de nouvelles commandes. En fait, sans nous en rendre compte, en utilisant **summarize**, nous avons utilisé une commande interne, c'est-à-dire qui fait partie de la composante de base de Stata alors que **egen**, pour sa part, est une commande étendue, c'est-à-dire qui a été ajoutée ultérieurement au langage initial Stata. Pour se convaincre, utilisons la commande **which** de Stata pour en apprendre plus sur ces deux commandes. Le résultat est produit à la figure 1.1.

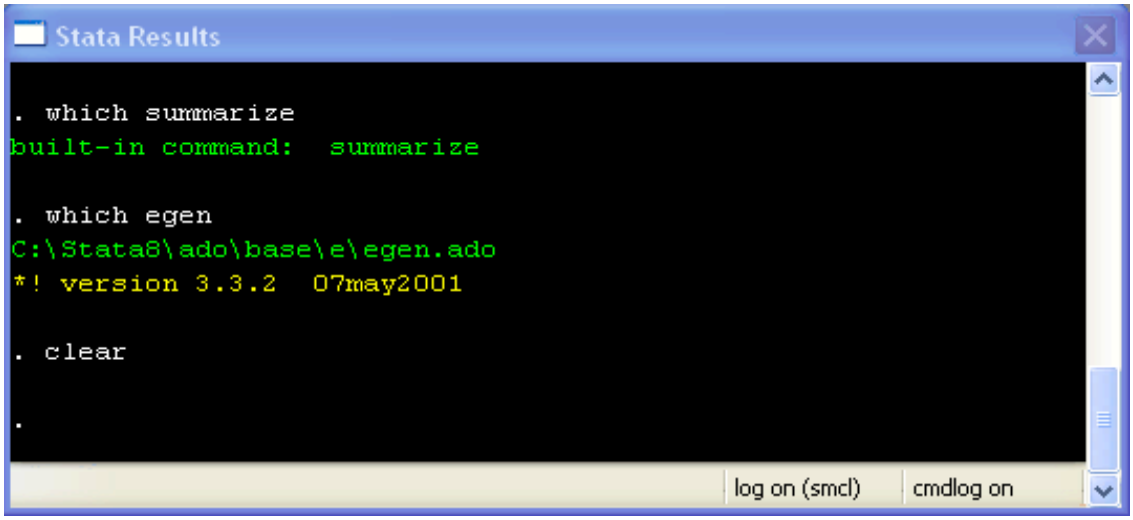



Figure 1.1 La commande **which**.

La commande **summarize** est bel et bien interne alors que la commande **egen** prend comme source le fichier **egen.ado** qui réside dans un sous répertoire bien précis du répertoire d'installation de Stata.

Techniquement, lorsque Stata rencontre la commande **egen** pour la première fois d'une séance, comme la commande n'est pas interne, Stata se met à chercher un fichier du même nom portant l'extension **.ado**, pour **automatic do** file, dans les divers répertoires prévus à cet effet. Pour obtenir la liste de ces répertoires, taper **sysdir**.

```
.  sysdir
   STATA:  C:\Stata8\
 UPDATES:  C:\Stata8\ado\updates\
    BASE:  C:\Stata8\ado\base\
    SITE:  C:\Stata8\ado\site\
    PLUS:  c:\ado\stbplus\
PERSONAL:  c:\ado\personal\
OLDPLACE:  c:\ado\
```

Figure 1.2 La commande **sysdir**.

Une fois le fichier **egen.ado** trouvé, Stata le charge en mémoire et traite ensuite **egen** comme si c'était une commande interne. Un fichier avec extension **.ado** doit être construit de façon particulière. On peut voir les fichiers **.ado** de façon générale en les traitant comme des fichiers comportant chacun une méthode (un programme). Un utilisateur peut faire du Stata à temps plein sans jamais avoir à créer, comme tel, un programme de type fichier **.ado**.

En soit, les fichiers **.do** du chapitre précédent étaient aussi des programmes. Nous allons les distinguer ici en disant qu'un fichier **.ado** comporte un programme alors qu'un fichier **.do** comporte une suite de commandes. On parlera d'un fichier **do** et d'un fichier **ado**. Un programme débute par un énoncé **program** et se termine par une clause **end**. Les fichiers **.do** et **.ado** se distinguent selon divers aspects que nous traitons maintenant.

## Liens en fichiers do et ado

Stata traite les fichiers **do** et **ado** avec plusieurs similitudes. Il existe aussi des différences à noter :

**Différences**

1. On invoque un fichier **do** en tapant **do nomfichier** alors qu'un fichier **ado** est chargé en tapant que le nom **nomfichier.**
2. Lors de l'exécution d'un fichier **do**, chaque commande et le résultat qui lui est associé est imprimé dans la fenêtre de résultats alors que dans le cas d'une invocation d'un fichier **ado**, seul le résultat final est imprimé. Un fichier **do** est donc plus facile à débogguer.

**Similitudes**

1. On peut passer des arguments aux fichiers **do** et **ado** de façon identique. Nous y reviendrons.
2. Les fichiers **do** peuvent appeler d'autres fichiers **do** et fichiers **ado**. Les fichiers **ado** peuvent appeler d'autres fichiers **ado** et des fichiers **do** (plus rare).

## 2. Les fichiers ado

Un fichier **ado** est un fichier qui comporte la définition d'un programme (ou méthode, ou commande externe) Stata. Faisons comme dans la formation Java et commençons par écrire une application que l'on sauve dans un fichier **ado**. Fixons nous comme objectif d'écrire une application qui écrit : You hou ! Mon premier programme Stata. Le code du programme se trouve à la figure 2.1 alors que la sortie est produite à la figure 2.2.

```
1     * Définition du programme (de la méthode).
2  program premierProg
3     display "You Hou ! Mon premier programme Stata."
4     display
5     display "You Hou !" _newline "Mon premier programme Stata."
6  end
7
```

Figure 2.1 Mon premier programme Stata.


Figure 2.2 Sortie du programme.

Comme on peut le constater dans la fenêtre de résultats, de taper **premierProg** a suffit pour lancer l'application qui imprime un message dans la fenêtre de résultats.

Stata a en effet été capable de trouver le fichier **premierProg.ado** car sa recherche a débuté dans le répertoire courant, endroit où nous avons stocké le code du programme. Il a alors stoppé sa recherche. Sinon, il aurait visité les différents répertoires produits par la commande **sysdir**. Une fois le code du programme testé et éprouvé, on pourrait déplacer le fichier source dans notre répertoire personnel **c:\ado\personal** que Stata visiterait lors de ses recherches.

La commande **display** joue le même rôle que la méthode **print** de Java. La ligne 5 du programme écrit le message sur deux lignes. La ligne de commande lancée avant de taper **premierProg** est très importante. Décomposons la commande

```
capture program drop premierProg
```

et expliquons chacun des mots. Le mot **capture** signale à Stata de poursuivre même si la commande qui suit le mot clef avorte. Le mot **capture** est surtout utilisé dans les fichiers de commande **do** pour empêcher les interruptions. Par la suite, la commande **program drop** *nomdeprogramme* enlève le code de *nomdeprogramme* de la mémoire. On doit lancer cette commande à chaque fois que l'on veut modifier ou mettre à jour un programme. Par défaut, Stata ne modifie pas le contenu d'un programme résidant en mémoire.

Dans la phase de développement d'un fichier **ado**, il est souvent plus pratique de copier le code du programme dans un fichier **do** car, comme nous l'avons mentionné précédemment, les fichiers **do** sont plus facile à débogguer. La version **do** du **premierProg** est présentée à la figure 2.3.

```
1
2  capture program drop premierProg
3
4     * Définition du programme (de la méthode).
5  program premierProg
6     display "You Hou ! Mon premier programme Stata."
7     display
8     display "You Hou !" _newline "Mon premier programme Stata."
9  end
10
11    * Début du code exécutable.
12 premierProg
13
```

Figure 2.3 La version du programme **premierProg.do**.

La ligne 2 est critique car elle permet de faire des changements au code du programme défini aux lignes 5 à 9. La ligne 11 lance son exécution. La sortie générée est produite à la figure 2.4 suivante.

Nous pouvons constater que tout s'est bien passé. Le code du programme a été interprété. Chaque ligne lue est listée et, s'il y a lieu, son résultat est imprimé. Une fois le code testé et si l'on veut réutiliser le programme **premierProg** dans nos futurs programmes, nous pouvons alors sauver le codes des lignes 5 à 9 dans une fichier appelé

**premierProg.ado** que nous pouvons conserver dans le répertoire de fichiers personnels **c:\ado\personal** pour que Stata puisse le trouver automatiquement.

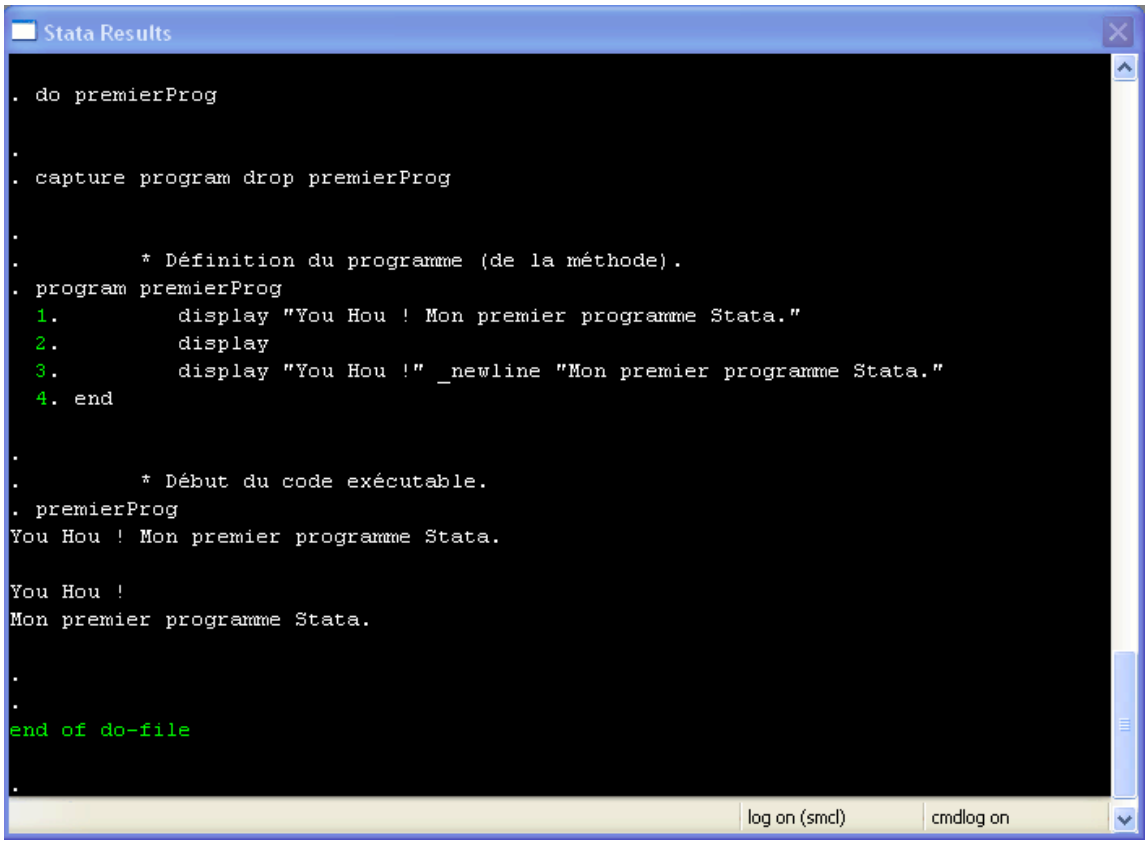



Figure 2.4 Sortie du programme **premierPRog.do**.

## 3. Les variables de programmation

Pour aider à la programmation, Stata permet la définition de variables spécifiquement dédiées à la construction de programmes. Ces variables que nous introduisons ici comme variables de programmation sont techniquement appelées macro. Tout comme en Java, il existait les concepts de variables locales et globales, ici nous avons les macro locales et globales. Une macro locale n'est disponible que lors de l'exécution du code qui la contient alors que le contenu de la macro globale demeure en mémoire et est donc accessible après l'exécution du code via la fenêtre de commande.

Une macro peut jouer le rôle d'une variable de programmation numérique ou encore celui d'une chaîne d'au maximum 67 784 caractères dans la version SE et de 80 caractères dans la version Intercooled. Une macro locale est définie comme suit :

```
local x = 2                  // x est une macro numérique locale et initialisée à 2
local x  “var1 var2 var3”    // x est une macro chaîne locale et initialisée var1 var2 var3
global x = 2                 // x est une macro numérique globale et initialisée à 2
```

Une macro numérique est initialisée grâce à une affectation effectuée par l'intermédiaire d'un signe d'égalité. Pour une macro de type chaîne, il est préférable d'omettre le signe d'égalité, car autrement la longueur de la chaîne permise serait de 80 caractères dans la version Intercooled. Pour accéder au contenu des macros, il faut utiliser des opérateurs spécifiques. Dans le cas d'une macro locale, englober le nom de la macro par les signes ` et ′. Celui de gauche correspond à un signe d'accent grave alors que celui de droite à un apostrophe. Pour ce qui concerne la macro globale, son contenu est accessible en ajoutant le signe de dollar $ devant le nom de la macro. Le petit programme de la figure 3.1 suivante démontre comment accéder au contenu des macros. La sortie de ce programme est produite à la figure 3.2 de la page suivante.

```
1  local x "Bonjour comment allez vous"
2  display "Contenu de x : `x'"
3
4  local x = 4 + 5
5  display "Contenu de x : `x'"
6
7  global x "Bonjour comment allez vous"
8  display "Contenu de x : $x"
9
10 global x = 4 + 5
11 display "Contenu de x : $x"
12
```

Figure 3.1 Utilisation des macros.

L'utilisation des macros permet d'écrire des programmes très compacts et donne accès à un outil de programmation très puissant. Les exemples que nous ferons par la suite convaincront sûrement les lecteurs. Voici un premier exemple que l'on exécute une fois chargé le fichier **auto.dta**. Les commandes

```
global listeVariables  “price headroom length”
summarize $listeVariables
```

La sortie suivante confirme ce qui a été dit précédemment. Le contenu de la macro globale est demeuré disponible même après la fin de l'exécution du code :

```
. global listeVariables  "price headroom length"

. summarize $listeVariables

    Variable |       Obs        Mean    Std. Dev.       Min        Max
-------------+--------------------------------------------------------
       price |        74    6165.257    2949.496       3291      15906
    headroom |        74    2.993243    .8459948        1.5          5
      length |        74    187.9324    22.26634        142        233
```

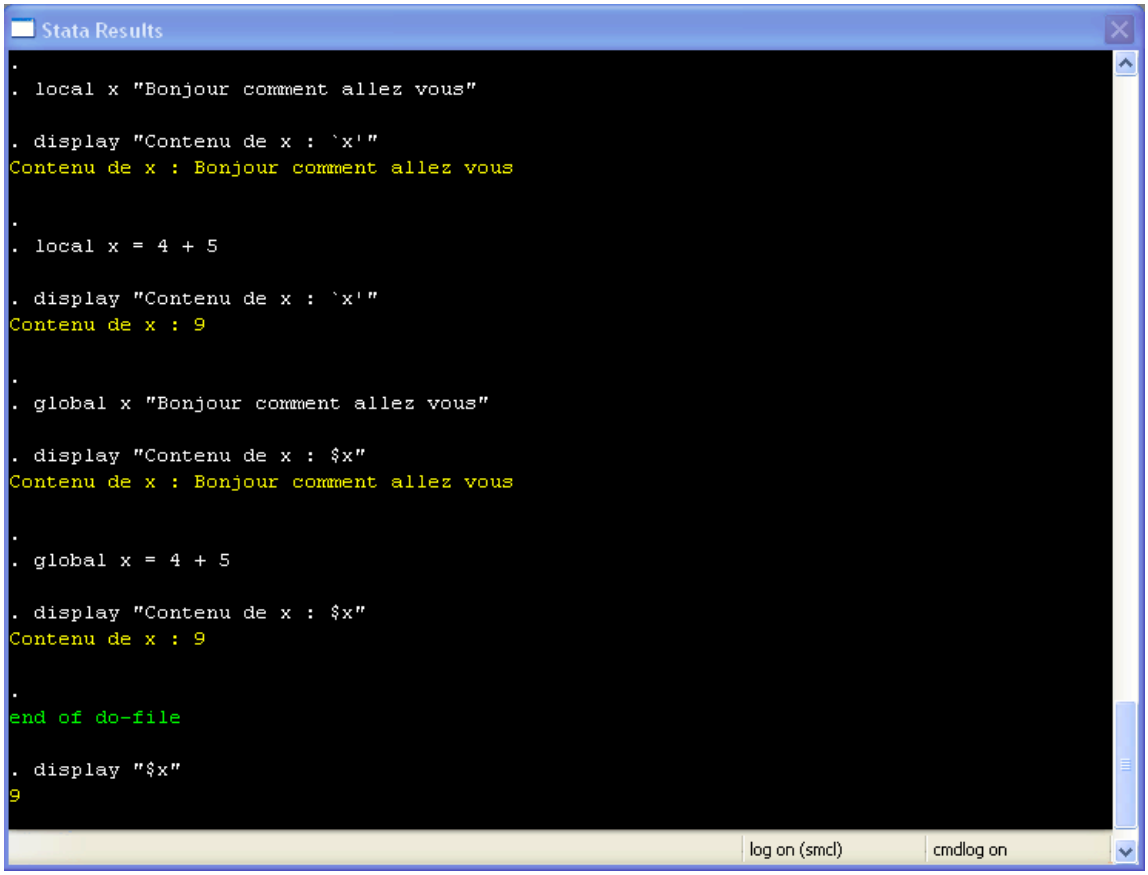



Figure 3.2 Résultats du programme.

Une des utilisations des macros locales concerne la mise en œuvre de boucles où la macro locale joue le rôle de compteur. Stata permet de définir des concepts généraux de compteurs. Pour l'instant, concentrons nous sur les compteurs numériques.

## Incrémentation

La commande suivante permet d'incrémenter le contenu d'une macro nommée **i**.

```
local i = `i' + 1
```

Cette commande s'écrit de façon tout à fait équivalente

```
local ++i
```

Pour ce qui est de la décrémentations, nous aurions **local --i** .

Dans ce chapitre, nous nous limitons aux concepts de base des macros. Ultérieurement, nous présenterons des généralisations des macros connues sous le nom de fonctions de macro étendues.

# 4. Les structures de contrôle

Le langage de programmation a été développé afin de permettre de faire des opérations sur les variables d'un ensemble de données sujettes à l'évaluation de conditions ou encore en procédant à des opérations en bouclant sur des listes de variables. L'utilisation des macros est essentielle pour effectuer ces tâches.

## 4.1 Énoncés conditionnels

La structure d'un **`if`** en programmation Stata est comme suit :

```
if expressionlogique1 {
   commandes
}
else if expressionlogique2 {
   commandes
}
else {
   commandes
}
```

Exemple :

Le code suivant :

```
local x "Salut"

if "`x'" == "Salut" {
   local z "Le message est : Salut."
}
else if "`x'" == "Bonjour" {
   local z "Le message est : Bonjour."
}
else {
   local z "Le message est autre."
}
display "`z'"

```

Figure 4.1 Énoncés conditionnels.

## 4.2 Structures de répétitions

Pour effectuer des répétitions, Stata permet d'utiliser une clause **`while`** et deux variantes de type **`for`** nommées **`forvalues`** et **`foreach`**. La clause **`while`** est identique à celle rencontrée en Java. La clause **`forvalues`** boucle sur des valeurs numériques alors que la clause **`foreach`** boucle sur les éléments d'une liste (une chaîne de mots séparés par des espaces). Voyons chacun de ces clauses.

### while

La clause **`while`** s'écrit :

**while** ***expr*** **{**
    ***commandes***
**}**

Exemple :

```
1  local i = 1
2  while (`i' < 5) {
3     display "x = `i' "
4     local ++i
5  }
6
x = 1
x = 2
x = 3
x = 4
```

Figure 4.2 Énoncé **while**.

### forvalues

Nous produisons la forme la plus commune du **`forvalues`**. Consulter l'aide en ligne pour découvrir les variantes de cette clause.

**forvalues** ***compteur*** **= a / b {**
    **commandes qui utilisent `*compteur*'**
**}**

permet de boucler sur les valeurs de la macro *compteur* dont le contenu `*compteur*' prend les valeurs a à b.

Exemple :

```
forvalues i=1/4{
   display "x = `i' "
}

x = 1
x = 2
x = 3
x = 4
```

Figure 4.3 Énoncés **forvalues**.

Supposons qu'il existe sur la base de données des variables nommées **var1-var5**. La clause suivante permettrait de créer d'une seule opération des variables valant le carré de chacun des variables.

```
forvalues i=1/5 {
      generate var`i'carre = var`i' ^2
}
```

Cette opération créerait en effet les variables **var1carre – var5carre**.

### foreach

Nous produisons la forme la plus commune du **foreach**. Consulter l'aide en ligne pour découvrir les variantes de cette clause.

**foreach** *lname* **of local|global** *nomliste* {
    **commandes qui utilisent** *`lname'*
}

Exemple :

```
local maListe "un deux trois quatre"
foreach var of local maListe {
   display "var = `var' "
}

var = un
var = deux
var = trois
var = quatre
```

Figure 4.4 Énoncés **foreach**.

## 5. Arguments de programmes

L'invocation des programmes (fichiers ado) ou des fichiers do en Stata permet la présence d'arguments. Par exemple, un programme nommé **somme** pourrait retourner la somme numérique de valeurs mises en arguments comme par exemple :

**somme 1 2 3 4 5**

retournerait la valeur 15. Une version **do** du programme exigerait d'écrire :

**do somme 1 2 3 4 5**

Les arguments sont passés aux programmes via des macro locales appelées automatiquement `1', `2',… La macro `*' contient la liste complète `1' `2' … Le programmes suivant servent à démontrer l'exploitation des arguments en Stata.

```
1  program somme
2     local i = 1
3     local sum = 0
4
5        * ``i'' vaut `1', `2', ...
6     while ( "``i''" != "" ) {
7        local sum = `sum' + ``i''
8        local ++i
9     }
10    display "La somme donne : " `sum'
11 end
12
```

Figure 5.1 Utilisation des arguments version **ado**.

```
1  capture program drop somme
2
3  program somme
4     local i = 1
5     local sum = 0
6
7        * ``i'' vaut `1', `2', ...
8     while ( "``i''" != "" ) {
9        local sum = `sum' + ``i''
10       local ++i
11    }
12    display "La somme donne : " `sum'
13 end
14
15 somme `*'
16
```

Figure 5.2 Utilisation des arguments version **do**.

## *6. Exemples de programmes Stata*

Selon

# 3
# La programmation avancée

## Thèmes traités

- Macros étendues
- Matrices
- Retourner des valeurs avec e(), r() et s()

# *1. Introduction*

Selon