Compilation : premier TP

Maîtrise, 1^er^ semestre 2001/2002

Avant-propos

Le projet qui vous sera demandé devra être écrit en Objective Caml. Ce TP présente des outils qui vous seront utiles pour le mener à bien.

– Vous devez rendre à votre chargé de TP votre réponse à la partie 4. Vous devez la rendre de façon à ce qu'il ait le temps de vous la corriger avant la séance de TP suivante (lui laisser deux jours ouvrables *pleins* ; il arrive aux chargés de TP de prendre des week-ends, surtout le samedi et le dimanche). Ne négligez pas de rendre ce TP, c'est grâce à lui que vos enseignants apprendront à vous connaître et pourront adapter leur pédagogie.
– Il est difficile de terminer ce TP en quatre heures mais vous avez des salles en libre-service !
– Avoir une assez bonne maîtrise de ce qui est demandé dans ce TP est *essentiel* pour la suite ; cela vous évitera bien des ennuis et vous permettra de travailler plus vite.

1 Utiliser les outils fournis

1.1 Au bâtiment 308

Tous les logiciels décrits ici sont normalement installés. Pour les utiliser il vous faudra modifier certaines de vos variables d'environnements. Pour cela, le plus simple est de charger quelques lignes de scripts écrites par vos enseignants en ajoutant la ligne

```
source ~courant/local/etc/tpcompil.csh
```

dans votre `.cshrc` si vous utilisez un shell de la famille `csh` (`csh`, `tcsh`, ...) et la ligne

```
source ~courant/local/etc/tpcompil.sh
```

dans votre `.profile` si vous utilisez un shell de la famille `sh` (`sh`, `bash`, ...).

Vous devrez également rajouter la première de ces deux lignes dans votre `.xsession` si celui-ci est un script `csh`, la deuxième si c'est un script `sh`.

1.2 Chez vous

Les logiciels nécessaires sont `make`, `cvs`, `emacs` ou `xemacs`, le mode Tuareg pour (x)`emacs` et `Objective Caml`, tous disponibles à la fois sur Unix et sous Windows.

2 Objective Caml

2.1 À lire

Sur le web :

– Le site web de Caml (`http://caml.inria.fr`). Passez-y, repassez-y. C'est une mine d'informations.
– L'ouvrage "Développement d'applications avec Objective Caml" (Éditions O'Reilly, Paris, avril 2000, ISBN 2-84177-121-0), disponible en ligne à l'adresse `http://www.pps.jussieu.fr/Livres/ora/DA-OCAML/index.html` .

- Le manuel de Caml en ligne (`http://caml.inria.fr/ocaml`). La partie "The Objective Caml library", et en particulier les chapitres "The core library" et "The standard library", documentent les bibliothèques d'Objective Caml. C'est ici que vous *devez* regarder pour savoir quelles fonctions, structures de données, modules sont fournis par OCaml. Si vous avez besoin de quoi que ce soit, commencez par regarder si la bibliothèque standard ne le contient pas déjà.
- Des conseils de programmation Caml : `http://caml.inria.fr/FAQ/pgl-fra.html`.
- "The Caml Hump" (`http://caml.inria.fr/hump.html`) et "The Ocaml Link Database" (`http://www.npc.de/ocaml/linkdb/`) : plein de liens vers des exemples de code .

2.2 Si vous débutez en Caml ou voulez vous rafraîchir la mémoire

Lisez la partie "The core language" du manuel. Regardez attentivement en particulier le codage du tri donné.

- Quel est l'algorithme de tri utilisé ?
- Écrivez une fonction qui, prenant une liste, la "découpe" en deux listes l_1 et l_2 de taille la moitié de la liste initiale (par "découper", il faut entendre que tout élément apparaissant dans l apparaîtra dans l_1 ou dans l_2 et réciproquement ; de plus si un élément apparaît n fois dans l, il devra apparaître x fois dans l_1 et y fois dans l_2, avec $x + y = n$). Utilisez cette fonction pour écrire une fonction de tri efficace d'une liste par un algorithme "Divide & Conquer" (au fait quel est le nom de cet algorithme de tri ?)

2.3 Pour les curieux : représentation interne des objets Caml

Si vous êtes curieux, vous vous demandez peut-être comment Caml représente les objets qu'il manipule. Pour voir la représentation d'un objet Caml, on utilise le module `Display` qui fournit deux fonctions `display` et `display_list.` Lisez les fichiers `~courant/local/doc/display.README` et `~courant/local/lib/ocaml/display.mli` pour comprendre comment utiliser ces fonctions. Observez comment sont codées les listes. Vous pourrez, après avoir fait l'exercice 4, regarder comment est représenté en mémoire `(fractale 4)`. Observez le partage des objets et discutez son optimalité en terme d'occupation de la mémoire.

3 Gestion d'un projet Caml

3.1 Structuration et compilation

3.1.1 Modules

La partie du manuel intitulée "The module system", quoiqu'un peu ardue, peut être intéressante à lire en complément de cette section et de la suivante.

Lorsqu'un projet commence à avoir une taille conséquente, il faut le découper en modules. Un module Caml est composé :

- d'une interface donnant le nom des types exportés par le module, ainsi que les noms des fonctions exportées avec leurs types ;
- d'une implantation dans laquelle sont définis les fonctions et les types du module.

Les fichiers d'interface ont pour suffixe `.mli` et les fichiers d'implantation, `.ml`.

Pour utiliser une fonction `f` d'un module `Toto` à l'extérieur du module `Toto`, il faut que cette fonction ait été déclarée dans `toto.mli`. Pour y faire référence, on a ensuite le choix entre

- faire référence à cette fonction sous le nom `Toto.f` (attention, la majuscule est obligatoire sur le nom du *module* ; la minuscule en début du nom du *fichier* l'est également) ;
- faire un `open Toto` au début du fichier où `f` sera utilisée, pour indiquer qu'on "importe" toutes les fonctions exportées par `Toto`, puis faire référence à `f` en l'appelant tout simplement `f`.

NB : Il vaut mieux faire des modules avec des noms assez courts et utiliser la première solution plutôt que commencer un fichier avec de multiples commandes `open`. Cela vous permettra d'avoir un code plus lisible tant pour votre propre usage que pour vos correcteurs et vous facilitera la maintenance et le déverminage.

3.1.2 Utilisation du compilateur Caml

Objective Caml est livré avec deux compilateurs : un compilateur vers une machine abstraite (`ocamlc`, dit compilateur des bytecode) qui permet de compiler rapidement un projet et un compilateur vers les machines concrètes (`ocamlopt`, dit compilateur natif), plus lent mais produisant un code optimisé plus rapide à l'exécution. Nous nous intéresserons d'abord à la compilation avec `ocamlc`, puis nous verrons comment utiliser `ocamlopt`.

Pour compiler un module `Toto`, on doit suivre les règles suivantes :
- on doit d'abord compiler son interface `toto.mli`, avec la commande `ocamlc -c toto.mli` ; pour cela, il faut avoir précédemment compilé, avec la même commande, les interfaces de tous les modules mentionnés dans `toto.mli`, s'il y en a.
- on peut ensuite compiler le fichier `toto.ml`, avec la commande `ocamlc -c toto.ml` ; là encore, il faut avoir précédemment compilé les interfaces de tous les modules mentionnés dans `toto.ml`.

Pour compiler un projet, il faut :
- compiler tous les modules du projet, en suivant les règles ci-dessus ;
- réaliser l'édition de liens en exécutant la commande

  `ocamlc -o proj mod1.cmo mod2.cmo ... modn.cmo`.

  Les fichiers `mod1`, ..., `modn` doivent être passés dans un ordre compatible avec les dépendances entre modules. `proj` est le nom de l'exécutable qui sera produit.

Notez qu'on peut compiler un fichier `toto.ml` faisant référence à un module `Titi` même si l'implantation de `Titi`, `titi.ml`, n'existe pas. Il suffit en effet que l'interface de `Titi` (qui doit s'appeler `titi.mli`) existe et ait été compilée. Ensuite, si `titi.ml` est créé, il ne pourra être compilé que s'il implante l'interface décrite dans `titi.mli`. Le développeur de `toto.ml` peut donc travailler et compiler son code *en parallèle* avec le développeur de `titi.ml`, à partir du moment où les deux développeurs se sont mis d'accord sur l'interface du module `Titi`, c'est-à-dire sur le contenu du fichier `titi.mli`.

Compiler avec `ocamlopt` est très similaire, aux différences suivantes près :
- les fichiers objets sont produits avec la commande `ocamlopt -c toto.ml` et ont le suffixe `.cmx`.
- l'édition de lien est réalisée par `ocamlopt -o proj mod1.cmx mod2.cmx ... modn.cmx`.

3.1.3 Utilisation du gestionnaire de projet `make`

`make` est un outil Unix permettant de gérer la compilation d'un projet. Il peut calculer pour vous l'ordre dans lequel les fichiers de votre projet doivent être compilés et lancer la compilation. Lorsque vous le lancez après avoir modifié certains fichiers sources de votre projet, il saura être *économe*, c'est-à-dire qu'il ne recompilera que les fichiers qui ont besoin d'être recompilés, d'où un gain de temps appréciable lors du développement.

Vous vous devez de maîtriser cet outil, c'est pourquoi *nous vous imposons d'utiliser* `make` *pour gérer la compilation de votre projet*.

L'outil `make` est guidé par un fichier `Makefile` qui donne un ensemble de règles pour compiler le projet donné. La documentation de `make` est disponible à l'adresse `http://www.gnu.org/manual/make/index.html`. Nous vous conseillons cependant d'apprendre la syntaxe des Makefile en lisant et en modifiant des Makefile existants.

À cet effet, nous vous fournissons un `Makefile` minimal pour les besoins de l'exercice 4. Copiez-le chez vous avec la commande :

```
cp ~filliatr/lib/Makefile.frequence Makefile
```

Nous vous fournirons un `Makefile` plus conséquent pour votre projet.

3.2 Édition du code

3.2.1 Emacs et Tuareg

Nous vous demandons d'utiliser l'éditeur de fichiers `xemacs` (ou `emacs` pour ceux qui connaissent déjà et qui préfèrent). Cet éditeur surpuissant est en effet un outil incontournable du monde Unix à l'heure actuelle.

Si vous n'êtes pas déjà utilisateur de `xemacs`, copiez le fichier `~courant/local/etc/tuareg-308.el` dans un fichier appelé `.emacs` à la racine de votre répertoire comme suit :

```
cp ~courant/local/etc/tuareg-308.el ~/.emacs
```

Sinon, insérez le contenu de ce fichier dans votre `.emacs`.

Les fonctionnalités apportées par Emacs[1] pour l'édition de texte Caml sont les suivantes :

- mots clés mis automatiquement en couleurs ;
- indentation semi-automatique ;
- possibilité de lancer la compilation directement à partir d'Emacs (menu Tuareg/Compile, ou `F5` et `F6`) ;
- positionnement sur l'erreur dans le source en cas d'échec de la compilation (taper `F7`) ;
- passage de l'interface au code et réciproquement avec `C-c C-a` (Control-`c` puis Control-`a`).

Voici quelques notions nécessaires pour commencer à utiliser (X)Emacs :

- Pour créer un nouveau fichier, il faut ouvrir un fichier (Open File) avec le nom du fichier que vous voulez créer.
- Pour sélectionner un nom de fichier avec la souris, il faut utiliser le bouton du milieu.
- Les raccourcis utilisent les abréviations suivantes :
  - `C-..` désigne la touche `Control`
  - `M-..` désigne la touche `Meta` (i.e. `Alt` sur les PC ou la touche avec un losange noir sur les `SUN`) ou `Esc`
- Si vous n'arrivez plus à rien faire, tapez `C-g` deux fois de suite. Cela annule les commandes en cours.

3.2.2 Gestion de versions : CVS

CVS (`http://www.cvshome.org`) est un outil qui vous permettra de gérer l'édition de votre code dans le temps ("je veux récupérer tel fichier tel qu'il était à telle date, telle heure") ainsi que l'édition partagée entre plusieurs utilisateurs (vous et votre binôme) et entre plusieurs lieux (université et maison). Si vous l'utilisez correctement, plus de problème du style "j'ai deux versions de tel fichier, laquelle est la bonne ? Il y a du bon dans chacun des deux, mais quoi ?".

Allez lire `http://echo-linux.alienor.fr/articles/cvs/cvs.html` (omettez les paragraphes 2.5 et 2.6 pour l'instant).

Initialisation d'une archive CVS. Choisissez un binôme avec qui essayer de faire fonctionner CVS.

Nous prenons l'exemple du binôme Alice (login `alice`) et Bob (login `bob`). Voici les étapes qu'ils doivent effectuer pour initialiser CVS :

- Alice et Bob choisissent que ce sera Alice qui détiendra l'archive dans son compte ;
- Alice et Bob éditent leurs fichiers de configuration et positionnent la variable `CVSROOT` à `:pserver:alice@cvs.ie2.u-psud.fr:/homes/alice/.REP_CVS_COMPIL`, ils font en sorte que ce changement soit effectif dans les terminaux qu'ils ont déjà ouverts (ce qu'ils vérifient avec un `echo $CVSROOT`).
- Alice lance le script `initialisation_cvs` qui lui demande un mot de passe qui sera utilisé pour protéger[2] l'accès au répertoire CVS. Ce mot de passe *doit* être compliqué (exemple `A#9/-_bL`), il n'a de toute façon pas à être retenu ;
- Une fois le script terminé, Alice envoie par mail à Bob le contenu de son fichier `~alice/.cvspass`.
- Bob récupère ce fichier et le sauve dans `~bob/.cvspass`. Il vérifie que les droits de ce fichier font qu'il n'est lisible que par lui seul.

Utiliser CVS. Le script d'initialisation a créé un nouveau projet appelé `Projet_Compilation`. Pour le récupérer, il suffit à Bob ou à Alice de faire

```
cvs checkout Projet_Compilation
```

[1]En fait, ces fonctionnalités sont apportées par le *mode* tuareg pour Emacs, plus les quelques réglages mineurs ajoutés par les chargés de TP.

[2]La protection en question n'est pas des plus efficaces (niveau comparable à rlogin, bien inférieur à celui de ssh), elle devrait néanmoins l'être suffisamment pour ce TP.

ce qui créera un sous-répertoire `Projet_Compilation` dans le répertoire courant. Ce répertoire contient un sous-répertoire `CVS` utilisé par `cvs`, auquel vous ne devez pas toucher !

Pour ajouter des fichiers dans le projet, il vous suffit de les créer dans ce répertoire `Projet_Compilation`, et de faire `cvs add` *fichier* (la même chose fonctionne avec un répertoire).

Vous pouvez modifier les fichiers qui sont dans votre répertoire `Projet_Compilation` comme vous voulez. Une fois que vous voulez mettre ces modifications dans l'archive, faites un `cvs commit` sur le répertoire ou sur les fichiers sur lesquels vous voulez entériner les modifications.

Pour récupérer les modifications faites par votre binôme, faites un `cvs update -d`.

Enfin, `cvs remove` *fichier* permet d'enlever un fichier de l'archive.

Il ne vous reste plus qu'à essayer ! Nous vous suggérons de procéder de la façon suivante :
- Alice et Bob récupèrent chacun une version de l'archive avec `cvs checkout`.
- L'un et l'autre créent des nouveaux fichiers, les ajoutent avec `cvs add` et entérinent ces ajouts avec `cvs commit`.
- Chacun essaie de se mettre à jour avec l'archive (`cvs update`), modifie les fichiers introduits par l'autre, et entérine les modifications (`cvs commit`).
- Que se passe-t-il si deux utilisateurs modifient simultanément le même fichier ?
- Que faire si vous perdez par erreur un fichier (en l'effaçant avec un `rm` malheureux) ?
- Vous avez fait des modifications sur un fichier que vous ne voulez pas entériner ; comment faire pour se resynchroniser avec l'archive ?

Vous avez une doc en ligne pour CVS, au format info. Vous avez également une importante documentation sur le Web à partir de `http://www.cvshome.org`

3.2.3 Documenter votre code : Ocamlweb

Pour vous aider à documenter votre code, et à obtenir une documentation lisible et utilisable, vous pouvez utiliser `ocamlweb`[3].

Le principe d'`ocamlweb` est simple : on documente le source (en LaTeX, dans des commentaires), et `ocamlweb` produit ensuite à partir du source un document où le code est embelli (mots clés en gras, identificateurs en italique, symboles mathématiques, etc.) et où la documentation présente dans le source est reproduite telle-quelle. Voici un exemple de fichier source documenté :

```
(* La fonction [fib] calcule $F_n$. *)

let rec fib n =
  if n <= 1 then 1 else fib (n - 1) + fib (n - 2)
```

et voici l'allure du document produit par `ocamlweb` :

La fonction `fib` calcule F_n.

let rec *fib* $n =$
  if n ≤ 0 then 1 else *fib* $(n-1)$ + *fib* $(n-2)$

Comme vous pouvez le constater, du code peut être cité dans les commentaires, entre crochets (ici l'identificateur `fib`).

3.3 Dernier recours : le debugger

La façon la plus efficace de déverminer un code est

1. de le commenter ;
2. de le *relire* ;
3. de le relire de nouveau ;

[3] `http://www.lri.fr/~filliatr/ocamlweb/index.fr.html`

4. de le relire encore.

Cependant, dans certains cas où rien ne va plus, il peut être utile d'exécuter votre programme dans un environnement permettant d'inspecter la valeur des différentes variables de votre programme, de suivre pas à pas son fonctionnement, de revenir en arrière après une erreur (pour voir ce qui l'a provoquée). Pour cela on utilisera le *debugger* d'Objective Caml. Rappelez-vous bien cependant que le debugger doit être considéré comme une bouée de sauvetage avec laquelle il est déraisonnable de traverser l'océan.

Pour être utilisable avec le debugger, votre programme doit être compilé en bytecode et l'option `-g` doit être passée lors de la compilation des modules et lors de l'édition de lien. Le debugger se lance à partir d'Emacs avec `M-x camldebug`. Les commandes les plus utiles sont :

- dans le source Caml : `C-x C-a C-b` pour poser un point d'arrêt ;
- dans le debugger : `step` (un pas d'exécution unitaire), `next` (un grand pas d'exécution — ne rentre pas dans le code des fonctions appelées), `run` (exécution jusqu'au prochain point d'arrêt), `print` (inspection de la valeur d'une expression), `back` (retour en arrière d'un pas).

Voir la documentation en ligne pour plus de détails.

Essayez par exemple d'utiliser le debugger sur votre programme de tri pour trier une liste de 10 éléments. Vous exécuterez votre programme jusqu'au bout, puis reviendrez en arrière jusqu'à l'endroit où s'est fait la dernière opération de fusion de deux listes triées. Suivez cette opération pas à pas.

4 Exercice : Fréquence des mots dans un texte

Dans cet exercice, on se propose d'écrire un programme qui détermine les 10 mots apparaissant le plus souvent dans un texte, avec leur nombre d'occurrences.

Le texte sera donné sur l'entrée standard sous la forme d'un mot par ligne, en lettres minuscules (éventuellement accentuées). Chaque mot pourra donc être lu avec la fonction Caml `read_line : unit → string`. Le résultat devra être affiché sous la forme suivante :

```
2897 de
1905 le
1623 la
1602 et
1426 il
1056 les
 918 en
 897 un
 770 du
 686 que
```

Vous pouvez tester votre programme en entrant manuellement des mots sur son entrée standard. Pour tester plus avant votre programme, la commande shell suivante met en minuscules les mots d'un fichier et les sépare par des retour-chariots :

```
cat fichier | tr A-ZÂÇÉÈÊËÎÏÔÖÙÜ a-zàçéèêëîïôöùü | tr -cs 'a-zàçéèêëîïôöùü' '[\n*]'
```

De manière équivalente, vous pouvez utiliser le petit script `decoupe` écrit pour les besoins de cet exercice :

```
decoupe fichier
```

Vous pouvez alors envoyer le résultat de cette commande sur l'entrée standard de votre programme avec un tuyau :

```
decoupe fichier | votre_programme
```

Trois œuvres du domaine public vous sont fournies dans le répertoire `~filliatr/lib` : *Cyrano de Bergerac* (`cyrano.txt`), *Le tour du monde en 80 jours* (`tdm80.txt`) et *Du côté de chez Swann* (`swann.txt`).

Modifier votre programme afin que l'on puisse fixer une limite inférieure au nombre de lettres des mots à considérer. Cette limite aura par défaut la valeur 2, à moins qu'elle ne soit spécifiée sur la ligne de commande de votre programme (comme unique argument).