| | | |
|---|---|---|
| **Université Pierre et Marie Curie** | option Informatique 2003−2003 feuille de TP numéro 6 | **Langage *Objective Caml*** |

**Problème**

Le but de ce problème est de programmer le coeur d'un mini-tableur, qui permette l'évaluation de formules arithmétiques simples, avec références absolues et relatives. On ne souhaite pas ici programmer les aspects graphiques du tableur, mais uniquement son fonctionnement interne. L'interface utilisateur sera donc réduit à un affichage en mode texte. Les commandes et les formules de l'utilisateur seront directement des valeurs *Objective Caml* (en "syntaxe abstraite").

Une partie des fonctions du tableur est déjà programmée. Copier les directements dans votre répertoire courant par la commande suivante (à effectuer dans un shell) :

```
cp /home/Ens/boulme/CAML/TP4/tableur*.ml .
```

Cela recopie les fichiers `tableur.ml` et `tableur-exemples.ml` dans votre répertoire. Ensuite, ouvrez le fichier `tableur.ml` et complétez-le en suivant l'énoncé du TP. Vous n'avez en principe pas besoin de lire le code des fonctions déjà écrites, leur fonctionnement est expliqué dans l'énoncé. Vous devrez néanmoins charger ces fonctions dans le toplevel, pour pouvoir exécuter vos propres fonctions. Il faudra taper la commande :

```
#use "tableur.ml";;
```

Les questions sont relativement faciles. Il ne faut pas vous laisser décourager par la longueur de l'énoncé !

# 1 Description générale

Un tableur est un ensemble de cellules. D'un point de vue abstrait, un tableur est une fonction qui applique une "action" à une plage rectangulaire de cellules (selectionnée par l'utilisateur à la souris). De façon interne, une plage sera désignée par les coordonnées d'un de ses points et sa taille. Comme usuellement sur les tableurs, les lignes (respectivement les colonnes) sont numérotées par ordre croissant du haut vers le bas (resp. de la gauche vers la droite). La numérotation commence ici à 0. Concrètement, dans cet exercice, un tableur sera une fonction de type :

```
action -> int -> int -> int -> int -> unit
```

où `action` est le type des actions qui sera défini ci-dessous, les deux premiers entiers représentent respectivement le numéro de ligne et de colonne du coin supérieur gauche de la plage, et les deux derniers entiers représentent respectivement le nombre de lignes et le nombre de colonnes de la plages. Ces deux derniers entiers doivent donc être supérieurs ou égaux à 1. Le but de ce problème est donc de faire la fonction `cree_mini_tableur` de type :

```
cree_mini_tableur: int -> int -> action -> int -> int -> int -> int -> unit
```

Cette fonction prend deux entiers `n` et `m` supérieurs à 1, et crée un tableur à `n` lignes et `m` colonnes.

Le type `action` est le type suivant (où `expr` est le type des formules qui sera défini plus loin):

```
type action =
 | Impr | Larg of int | Efface_form | Change_form of expr
```

- L'action `Impr` affiche la valeur des cellules sélectionnées.
- L'action `(Larg i)` oblige les cellules sélectionnées à afficher exactement `i` caractères, suivi d'un espace. Lorsque le texte à afficher par la cellule est supérieur à `i` caractères, elle affichera `i` fois le caractère `@`. Sinon, elle complètera le texte à afficher par des espaces, insérés à gauche : dans une même colonne, les cellules seront donc alignés à droite.
- L'action `Efface_form` efface les formules dans les cellules sélectionnées.
- L'action `(Change_form e)` met la formule `e` dans les cellules sélectionnées.

Les expressions de formules sont données par la syntaxe abstraite suivante:

```
type binop = Plus | Mult | Moins

type expr =
  | Valeur of int
  | Binop of binop * expr * expr
  | Absolue of int*int
  | Relative of int*int
  | AbsRel of int*int
  | RelAbs of int*int
```

Les opérateurs arithmétiques autorisées sont les opérateurs binaires suivants : l'addition (constructeur `Plus`), la multiplication (constructeur `Mult`), et la soustraction (constructeur `Moins`). Une formule est donc soit un entier (constructeur `Valeur`), soit une opération binaire (constructeur `Binop`), soit une référence, c'est-à-dire une valeur calculée par une autre cellule. Une référence peut-être absolue : `(Absolue (i,j))` désigne la cellule de numéro de ligne `i` et de colonne `j`. Elle peut être relative : si `(l,c)` désigne le cellule où se trouve la formule, alors `(Relative (i,j))` désigne la cellule `(l+i,c+j)`. Les références peuvent aussi être mixtes : `(AbsRel (i,j))` désigne la cellule `(i,c+j)` et `(RelAbs (i,j))` désigne la cellule `(l+i,j)`.

Le tableur doit gérer 2 sortes d'erreurs : les cycles, et les valeurs non définies. En effet avec le langage de formule précédent, il est possible de faire des cycles (une cellule $A$ fait référence à la valeur d'une cellule $B$, qui elle-même fait référence à la valeur de $A$). Lorsqu'un utilisateur proposera (par l'action `Change_form`) une formule qui fait un cycle, celle-ci sera refusée par le tableur. Le tableur affichera alors un message d'erreur indiquant la cellule provoquant le cycle.

Une cellule $A$ peut aussi faire référence à une cellule $B$ qui n'a pas été définie. Le tableur accepte ce genre de formules, mais à l'affichage il marque une erreur dans la cellule $A$ (les coordonnées absolues de $B$ suivies d'un point d'interrogation, ex: "(10,20)?". L'affichage de la cellule $B$ sera le caractère _, marquant que la cellule n'a pas été définie.

## 2 Types prédéfinis

Pour gérer les erreurs, on utilise le type paramétré suivant:

```
type ('a,'b) opt = Normal of 'a | Erreur of 'b
```

Par convention, le constructeur `(Normal v)` représente la valeur `v`, et `(Erreur e)` représente une erreur qui contient une information `e` sur la cause de l'erreur.

Les cellules seront représentées par un enregistrement avec champs mutables, dont le type est suivant:

```
type cellule =
  {
    mutable largeur: int;
    mutable formule: (expr, unit) opt ;
    mutable valeur: (int, int*int) opt ;
    mutable statut: bool;
    dep_base: bool array array;
    dep : bool array array ;
  }
```

Voilà la description de ces champs:

- Le champ `largeur` contient la largeur d'affichage de la cellule (cf. action `Larg`).
- Le champ `formule` contient la formule de la cellule. Lorsqu'il n'y a pas de formule dans la cellule, ce champ est `Erreur ()`.
- Le champ `valeur` contient la valeur de la cellule. Ce champ peut contenir une erreur `(Erreur (i,j))` où `(i,j)` sont les coordonnées (absolues) d'une cellule qui n'est pas définie, et dont dépend la valeur de la cellule courante. Eventuellement `(i,j)` peut désigner la cellule courante, lorsque celle-ci n'a pas de formule.
- Le champ `statut` indique si la valeur de la cellule a bien été actualisée (ie recalculée) depuis les derniers changements dans le tableur.
- Le champ `dep_base` est une matrice de booléens, de la taille du tableur, et qui indique pour chaque cellule `(i,j)` si la formule de `(i,j)` fait directement référence à la cellule courante.
- Le champ `dep` est une matrice de booléen, de la taille du tableur, et qui indique pour chaque cellule `(i,j)` si la valeur de `(i,j)` dépend de celle de la cellule courante. Dans cette matrice, on considère qu'une cellule dépend forcément d'elle-même.

Dans le reste du problème, on distinguera bien les *"dépendances"* entre cellules (champ `dep`) des *"dépendances basiques"* entre cellules (champ `dep_base`). (En mathématiques, on dit que la relation *"dépend de"* entre cellules, est la clôture transitive-réflexive de *"dépend basiquement de"*).

Enfin, l'état interne du tableur sera stocké sous la forme d'un enregistrement de type `feuille` :

```
type feuille =
  {
    n_ligne: int;
    n_col: int;
    tab : cellule array array ;
  }
```

Les champs `n_ligne` et `n_colonne` indiquent respectivement le nombre de lignes et de colonnes dans la table. Le champ `tab` est une matrice de cellules.

## 3 Questions

**Question 1** Ecrire la fonction qui correspond à l'action `Larg` sur une seule cellule. Cette fonction prend une cellule et un entier, et met cet entier dans le champ `largeur` de la cellule.
`fixe_largeur_cellule : cellule -> int -> unit`

**Question 2** Ecrire la fonction `cree_cellule` qui prend quatre entiers `i`, `j`, `n` et `m` et retourne une cellule initialisée, de coordonnées `(i,j)` et dans une table de `n` lignes par `m` colonnes. La cellule devra être initialisée de telle sorte que `statut` soit à `true`.
`cree_cellule : int -> int -> int -> int -> cellule`

**Question 3** Ecrire la fonction qui prend deux entiers `n` et `m` et crée une feuille de `n` lignes par `m` colonnes. Les cellules de cette feuille devront être correctement initialisées. Pensez à l'exercice 8 du TP 4.
`cree_feuille : int -> int -> feuille`

**Question 4** Ecrire la fonction qui prend une feuille, les numéros `l` et `c` d'une cellule et qui retourne la liste des cellules qui dépend de `(l,c)` (au sens des dépendances du champ `dep`).
`dep_sur_lc : feuille -> int -> int -> (int * int) list`

**Question 5** Même question que précédemment, mais avec la liste des cellules dont dépend `(l,c)`.
`dep_de_lc : feuille -> int -> int -> (int * int) list`

**Question 6** Lorsqu'on efface ou on change une formule dans une cellule, il faut mettre à jour les champs `dep_base`, `statut` et `dep` de chaque cellule de la table. Ecrire la fonction `supprime_maj` qui fait cette mise-à-jour dans le cas de la suppression d'une formule. Cette fonction prend une table, les numéros `l` et `c` de la cellule dans laquelle sera effectuée cette suppression. Elle doit donc garantir les points suivants :

- `(l,c)` ne dépend basiquement plus d'aucune cellule.
- Les cellules qui dépendaient de `(l,c)` devront être reactualisées (leur champ `statut` passe à `false`)

Ensuite, il faudra mettre à jour les champs `dep` des cellules. En fait, la fonction prédéfinie `roy_warshall` de type s'occupe de se problème : elle prend comme argument la table en cours, la liste des dépendances de `(l,c)` et la liste des dépendances sur `(l,c)`. Elle a pour type :
`feuille -> (int * int) list -> (int * int) list -> unit`
La fonction que vous devez écrire a la signature suivante :
`supprime_maj : feuille -> int -> int -> unit`

**Question 7** Ecrire la fonction qui correspond à l'action `Efface_form` sur une seule cellule. Cette fonction prend une feuille, et le numéro `l` et `c` de la cellule. Elle commence par mettre à jour le reste de la table (cf. fonction précédente), puis les champs `formule`, `valeur` et `statut` de la cellule en question.
`efface_formule : feuille -> int -> int -> unit`

**Question 8** Lorsqu'on change une formule dans une cellule, il faut calculer les dépendances qu'induit cette formule. Une fonction prédéfinie `calcule_dep_base` se charge de se problème. Elle a le type suivant :

```
(int -> int -> int * int -> bool) ->
    int -> int -> expr -> ((int * int) list, int * int) opt
```

Elle prend en argument une fonction `est_dep`, les numéros de ligne et de colonne d'une cellule `l` et `c`, et une formule `e`. Elle retourne soit la liste des cellules (en notations absolues), dont dépend basiquement la cellule `(l,c)` via `e`, soit une erreur `(Erreur (i,j))` qui indique que `e` crée un cycle avec la cellule `(i,j)`. Pour détecter les cycles, cette fonction se fie à la fonction `est_dep` : celle-ci prend les numéros `l` et `c` d'une cellule et les coordonnées `(i,j)` d'une autre cellule, et indique si `(i,j)` dépend de `(l,c)`.

Ecrire la fonction, qui appliquée sur une matrice de cellules, sera passée comme premier argument de `calcule_dep_base`.
`est_dep : cellule array array -> int -> int -> int * int -> bool`

**Question 9** Ecrire la fonction qui prend en argument une feuille, les numéros `l` et `c` d'une cellule, et une liste de références dont dépend basiquement cette cellule (telle que retournée par `calcule_dep_base`), et qui met à jour les dépendances dans la feuille.

Il faut rendre compte du fait que si `(i,j)` est une coordonnée de cette liste, alors `(l,c)` dépend basiquement de `(i,j)`, et que pour toute cellule `(i1,j1)` dont dépend `(i,j)` et toute cellule `(i2,j2)` qui dépend de `(l,c)` alors `(i2,j2)` dépend de `(i1,j1)`. On pourra utiliser la fonction `List.iter` qui itère une fonction sur les listes, et la fonction `dep_sur_lc` précédente.

`add_dep_maj : feuille -> int -> int -> (int * int) list -> unit`

**Question 10** Ecrire la fonction correspondant à l'action `Change_form` sur une seule cellule. Cette fonction prend en argument une feuille, les numeros `l` et `c` de la cellule, et la formule à insérer.

Elle commence par appliquer `calcule_dep_base` et agit en fonction de son résultat. Si celui-ci est une erreur (un cycle), alors cette fonction affiche une erreur conformément à la description précédente. Sinon, elle commence par appeler `supprime_maj` (on supprime l'ancienne formule), puis on met à jour les dépendances induites par la nouvelle formule. Enfin, on met à jour le champ `formule` de la cellule.
`change_formule : feuille -> int -> int -> expr -> unit`

**Question 11** Ecrire la fonction `eval` qui prend une feuille, les numéros `l` et `c` d'une cellule, et qui retourne la valeur associée à cette cellule. Si le champ `statut` de cette cellule est à `false`, il faudra recalculer cette valeur.

Pour évaluer les formules, une fonction `eval_expr` a été prédéfinie. Son type est le suivant :
`(int -> int -> (int, 'a) opt) -> int -> int -> expr -> (int, 'a) opt`
Elle prend en argument une fonction `eval_table`, les numéros `l` et `c` d'une cellule, et `e` une formule. Elle retourne la valeur associée à cette formule dans la table. Pour connaître la valeur des cellules nécessaires à son évaluation, elle se fie à `eval_table`. Celle-ci prend les numéros `l'` et `c'` d'une cellule, et retourne la valeur associée dans la table. Remarquons enfin, que si une application de `eval_table` retourne une erreur, alors `eval_expr` retournera cette erreur.

Dans la définition `eval`, on pourra commencer par définir une fonction récursive `eval_table` qui pourra être passée comme paramètre de `eval_expr`. La fonction `eval` se contentera alors d'appliquer `eval_table`.
`eval: feuille -> int -> int -> (int, int * int) opt`

**Question 12** Ecrire la fonction correspondant à l'action `Impr` sur une seule cellule. Cette fonction doit respecter les contraintes de l'affichage exposées dans l'énoncé. Elle pourra utiliser la fonction précédente pour connaître la valeur des cellules.
`imprime_cell : feuille -> int -> int -> unit`

**Question 13** Ecrire la fonction qui prend une feuille, une action et les numéros `l` et `c` d'une cellule, et qui applique cette action sur cette cellule.
`applique_cell : feuille -> action -> int -> int -> unit`

**Question 14** Ecrire la fonction qui applique une action sur une plage de cellule. On pourra utiliser la fonction précédente.
`applique : feuille -> action -> int -> int -> int -> int -> unit`

**Question 15** Ecrire la fonction `cree_mini_tableur`. On pourra utiliser la fonction précédente.
`cree_mini_tableur : int -> int -> action -> int -> int -> int -> int -> unit`

**Question 16** Tester votre programme sur des exemples (en particulier en utilisant le fichier d'exemples `tableur-exemples.ml`).

**Question 17** Que se passe-t-il si l'utilisateur rentre une formule qui fait appel à une cellule dont les coordonnées sont hors du tableur ?