PROGRAMMATION D'UNE CALCULATRICE AVEC OCAML

Matthieu DORIER
Christophe RIOLO

Sous la direction de Luc BOUGÉ et Christophe AVENEL

Remerciements à Vincent Picard pour la présentation du document.

Calculatrice en Caml

Matthieu DORIER Christophe RIOLO

10 décembre 2008

Résumé

Nous présenterons dans ce rapport les solutions que nous avons apportées au problème de la programmation d'une calculatrice en écriture postfixe.

Table des matières

1 DÉCOMPOSITION DE LA LIGNE ENTRÉE

1.1 La pile d'exécution

Principe

Considérons un exemple de calcul que l'on voudrait faire à la calculette :
`6 * (8 + cos(3.14159265358979312))`
(dont on sait que le résultat est, à peu de choses près, 42). Puisque la notation de la calculette est postfixée, on notera cette opération :
`6 8 3.14159265358979312 cos + * =`
(le '=' servant à indiquer que l'on veut afficher le résultat de l'opération).
Cette opération sera codée en une pile de jetons :

Affect
Fois
Plus
Cos
3.14159265358979312
8
6

Pour transformer cette suite de caractères en pile de jetons, nous avons implémenté un type `token`, pour définir la pile de jetons `istack`, puis nous avons parcouru l'entrée caractère par caractère, en avisant en fonction de ce qui était déjà dans la pile, comme nous allons le voir.

Les types

Le type `token` servant à lister les jetons recense les différentes opérations que l'on peut faire :

```
type token =
  | Int of int     (* entier         *)
  | Float of float (* flottant       *)
  | Plus           (* addition       *)
  | Moins          (*soustraction    *)
  | Fois           (* multiplication *)
  | Div            (* division       *)
  | Affect         (* affectation et impression de résultat *)
  | Neg            (* - unaire       *)
  | Sqrt           (* racine carrée  *)
  | Puiss          (* puissance      *)
  | Cos            (* cosinus        *)
  | Sin            (* ... sinus      *)
  | Exp            (* exponentielle (qui permettra pour les puristes de définir eux-mêmes l
  | Ln             (* logarithme népérien (le veinard !) *)
;;
```

Est ensuite créé le type `istack`, pile de `token`, muni des seules trois fonctions : création, empilement (push) et dépilement (pop), la gestion de la vacuité de la pile se faisant par soulèvement d'exception lors du dépilement.

1.2 La gestion caractère après caractère

Lors de la lecture d'un caractère sur l'entrée standard, plusieurs cas sont possibles, du plus simple (lecture de '+' par exemple) au plus complexe (lecture de '3.14159265358979312').

La lecture des nombres est la plus complexe puisqu'il s'agit de distinguer les entiers des flottants, Caml n'autorisant pas le transtypage automatique. De plus, quand on lit un chiffre, comme le chiffre 3, il y a trois cas très différents à distinguer :

1. 3 est dans 365 : c'est le premier chiffre lu
2. 3 est dans 635 : c'est un chiffre d'une partie entière
3. 3 est dans 6.35 : c'est une décimale

Pour pouvoir traiter correctement ces différents cas, plusieurs options sont possibles, la nôtre a été d'introduire deux variables :

- une variable `in_number` qui permet de retenir si on est en train de lire un nombre ou non
- une variable `virgule` qui mémorise la place après la virgule à laquelle doit se trouver le prochain chiffre

Si, à la lecture d'un chiffre, `in_number` est fausse, alors on est dans le cas 1 Si elle est vraie, alors si le premier élément de la pile est un entier, on est dans le cas 2, et si le premier élément de la pile est un flottant, alors on est dans le cas 3, et la puissance de 10 qui multiplie le chiffre est `virgule`.

La transformation d'un entier en flottant (nécessaire pour le cas 3) est faite à la lecture d'une virgule (un point en réalité).

Une fois les nombres traités, les autres semblent simple : les caractères spéciaux sont traités directement (un '+' correspond forcément à un Plus dans la pile), avec pour seule difficulté : penser à remettre `in_number` à faux.

La lecture de lettres conduit à un choix, pour les fonctions complexes, mais il suffit de continuer à lire des caractères tant que ce n'est pas incompatible : lecture du "cos" :

```
'c' -> 'o' -> 's' -> push Cos
```

à comparer avec lecture de "cas" :

```
'c' -> 'a' -> raise Bad\_char
```

Ainsi, la pile des commandes à exécuter se construit caractère après caractère.

Remarque Pour simplifier les choses, les nombres négatifs sont des nombres positifs auxquels a été appliquée la fonction `neg`, ceci évite de faire du préfixé dans le cas du caractère '-' et du postfixé le reste du temps, et donc de simplifier le traitement.

2 LA BOUCLE D'EXÉCUTION

Pour décrire le parser, nous avons opté pour une description ascendante : en partant des types, nous somme montés vers la fonction qui utilisait ces types : handle_char. Pour la boucle d'exécution, nous allons prendre l'approche descendante, c'est à dire que nous allons partir de ce que l'utilisateur voit pour ne voir qu'ensuite les rouages internes.

2.1 L'interface homme/machine

Au lancement du programme, l'utilisateur se fait accueillir par un ">>>>". Cette simple fonction de prompt s'assure également de la synchronisation de l'écriture sur `stdout` avec la fonction `flush`, et de vider la pile de commandes (pour repartir sur un calcul neuf !). L'utilisateur peut ensuite saisir sa commande, voit son résultat affiché (ou non si son calcul est mauvais), et lui est proposé de nouveau de saisir un calcul. Toute cette boucle est gérée par la fonction `toplevel` que nous allons décrire.

La fonction `toplevel` se déroule en trois étapes : prompt, lecture, évaluation. Ou du moins, c'est ce qu'il apparaît à l'utilisateur, car en réalité, la lecture se fait caractère après caractère, et la fonction `handle_char` est appelée dans une boucle while infinie. Elle lève ensuite des exceptions pour signaler que son traitement est terminé. Ces exceptions sont rattrapées dans un harnais hors de la boucle while, ce qui permet d'en sortir. Trois exceptions sont possibles :

Exit : L'utilisateur a tapé "exit" (ou "!") pour signifier qu'il veut quitter, on lui demande alors confirmation

Bad_char c : Le caractère *c* n'était pas attendu la où il a été rencontré (appel d'une fonction non implémentée par exemple)

Execute : La commande a été correctement parsée (on est arrivé au caractère '\n' sans lever d'autre exception). En ce cas, on appelle la fonction `exec_istack`

2.2 La fonction exec_istack, cœur de la calculatrice

Il y a des fonctions de plus bas niveau pour l'exécution, et ce sont les fonctions qui font les calculs (la fonction d'addition, la fonction de multiplication, ...). Ce fonctions sont intéressantes en ce qu'elles doivent filtrer les valeurs qu'elles prennent en entrée pour les adapter aux fonctions ocaml (ex : cos ne prend que des flottants en entrée en OCaml, ce n'est pas ce que l'on voudrait dans notre calculatrice, donc il faut prévoir un float_of_int). Le véritable intérêt réside dans la commande `exec_istack`.

Comme on écrit en écriture postfixe, et que l'on place les jetons dans une pile, les différentes commandes se retrouvent en ordre préfixe, la fonction avant ses arguments. Alors il suffit de lire la fonction, on en déduit le nombre d'arguments à retirer de la pile ensuite, et on applique la fonction, c'est ce que fait récursivement la fonction `exec_istack`. Récursivement car dans l'exemple donné au début (en oubliant le Affect) :

Fois
Plus
Cos
3.14159265358979312
8
6
_____

Ce n'est pas parce que ce qui suit `fois` n'est pas des nombres que la commande est fausse, il faut juste continuer les calculs.

Le rôle du Affect (nommé ainsi en prévision de l'ajout de variables, mais cela ne s'est finalement pas fait) est simplement d'imprimer à l'écran le jeton suivant. Comme les jetons de calcul sont consommés lors de l'exécution de la pile, il n'y a affichage que des nombres (mais bon, dans le doute, ne nous privons pas de prévoir tous les cas …En plus cela retire des avertissements de filtrage non exhaustif !)

Le compte est bon !