Université Pierre et Marie Curie
Formation permanente

– o –

Prise en Main d'Objective Caml

Travaux encadrés sur machines - J3

Emmanuel Chailloux Pascal Manoury Bruno Pagano Michel Mauny

Mars 2006

1 Troisième jour : modules, bibliothèques et outils

1.1 Module Pile simple

Soit la signature suivante :

```
module type PILE =
sig
  exception PileVide
  type 'a pile
  val create : unit -> 'a pile
  val push : 'a -> 'a pile -> unit
  val top  : 'a pile -> 'a
  val pop  : 'a pile -> unit
end
```

1. Ecrire le module Pile respectant cette interface. Le choix de l'implantation est libre, néanmoins la structure interne de la pile devra être physiquement modifiable.
2. Ecrire un petit test de ce module dans un fichier `test.ml` qui inverse le contenu d'une pile .
3. Utiliser ces fichiers au niveau du toplevel (`#use ...`).
4. Utilier le compilateur de byte-code `ocamlc` pour la production d'un exécutable non autonome.
5. Créer l'exécutable, tester le sur votre machine, puis tester le sur une autre machine (d'architecture différente)
6. Créer un exécutable autonome :
   (a) en utilisant le compilateur de byte-code
   (b) avec le compilateur natif

1.2 Calculette post-fixée

On reprend la réalisation d'une calculette post-fixée.
L'application recevra une suite de caractères :
– les nombres seront empilés dans la pile des opérandes
– les opérateurs effectueront le calcul demandé sur les 2 premiers élément de la pile et empileront le résultat.
– le retour chariot affichera le sommet de pile.
On utilisera la fonction `Stream.of_channel` sur l'entrée standard `stdin`.

1. Ecrire une première version où l'on entrera seulement des chiffres comme opérandes. On reçoit donc un flot de caractères. Les chiffres seront convertis en `int` ou `float` et empilés dans la pile des opérandes.
2. Que se passe-t-il si une division par zéro est effectuée ? Dans le cas d'un exécutable, comment peut-on afficher une exception non récupérée ?
3. On ajoute un analyseur lexical qui prend le flot de caractères et retourne un flot d'unités lexicales (jeton ou `token`. On utilisera le module prédéfini `Genlex` pour ce petit analyseur.
4. Modifier en conséquence l'application.

1.3 Module paramétré PileOp

On cherche à étendre les opérations sur les piles en réutilisant celles définies dans le module précédent. Pour cela on écrira un module paramétré, dont le paramètre a la signature `PILE`.
– `dup` : duplique le sommet de pile
– `exch` : échange les deux valeurs en sommet de pile
– `copy` : crée une copie de la pile
– `to_list` : crée une liste des éléments de la pile

1. Ecrire le module `PileOP`
2. L'utiliser avec le programme de test précédent
3. On ajoute un opérateur `pstack` qui affiche le contenu de la pile. Comme il n'existe pas de fonction `print` polymorphe, on ajoute un nouveau paramètre à ce module de signature `PRINTABLE` :

   ```
   module type PRINTABLE = sig type t val print : t -> unit end
   ```

   Modifier en conséquence la définition du module `PileOp`.
4. Définir un module `Integer` satisfaisant cette interface.
5. Afficher le contenu des piles créées dans le test en utilisant `pstack`.

1.4 Calculette infixe

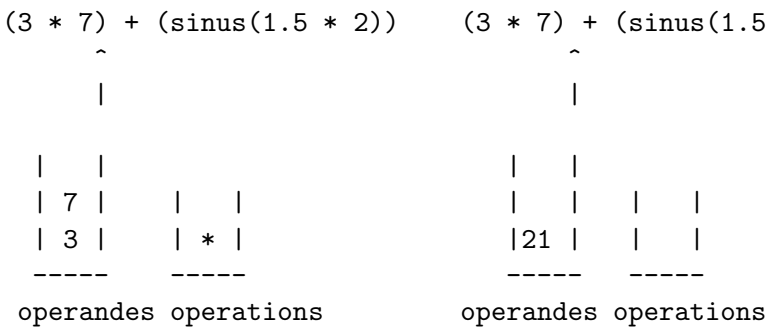
On utilisera deux piles : une pile des opérandes et une pile des opérateurs. A chaque jeton lu on associera une action :
– Une valeur (constante ou variable) est empilée dans la pile des opérandes.
– Un opérateur est empilé dans la pile des opérateurs.
– Une `(` est ignorée.

– Une) entraîne le calcul de l'opérateur au sommet de la pile des opérateurs sur un ou deux arguments de la pile des opérandes, le résultat est empilé dans la pile des opérandes

La fin de la liste entraîne le calcul sur les opérateurs restants.
On s'aperçoit de l'intéret du polymorphisme paramétrique par l'utilisation de 2 piles contenant des éléments de type différent.

1. Ajouter les parenthèses dans les mots clé de ce petit langage
2. Modifier la fonction d'évaluation du flot de jetons.
3. Ajouter les fonctions `sinus` et `cosinus` comme fonction prédéfinies de la calculette.

Exemple :



1.5 Création de Makefile

En utilisant la commande `ocamldep` sur un ensemble de fichiers `.ml` et `.mli` on obtient les dépendances entre modules.
Ces dépendances peuvent ensuite être intégrées dans un `Makefile`.
En reprenant le squelette de `Makefile` de la présentation :

1. Ecrire un `Makefile` pour une des calculettes pour produire :
   (a) un exécutable autonome en byte code
   (b) un exécutable autonome natif
2. Créer ces applications
3. Modifier un des fichiers `.ml` ou `.mli`, puis relancer la compilation. Les dépendances sont-elles bien respectées ?

1.6 Interface avec C

Difficultés utiliser le GC d'O'Caml
– allocation dans le tas O'Caml d'obejts C
  – `alloc_*`
– prendre en compte des racines du monde C
  – `CAMLparami` : pour les paramètres
  – `CAMLlocali` : pour les déclations locales
  – `CAMLreturn` : pour le résultat

Exemple Calcul sur les polynômes (bibliothèque écrite en C)
La bibliothèque `poly` définit en C les structures de données `monome` et `polynome` (à une seule variable) ainsi que les opérations d'addition de 2 polynômes, de produit d'un monôme par un polynôme et la dérivée. Le fichier `poly.h` contient l'interface de cette bibliothèque.
Le programme O'Caml `poly.ml` définit les types O'Caml pour représenter les polynômes et la définition de la multiplication en utilisant les fonctions de la bibliothèque. Le lien entre nom O'Caml et nom C est indiqué par les déclarations `external`.
L'interface entre les deux mondes est réalisée par l'encapsulation des fonctions C pour tenir compte de la gestion mémoire. Toute allocation s'effectuera dans le tas O'Caml. Le fichier `vpoly.c` contient les fonctions de conversion de types et les fonctions d'encapsulation des primitives de la bibliothèque.

1. Compiler et tester cette application en utilisant son `Makefile`.
2. Ajouter en C la soustraction d'un monôme à un polynôme.
3. Ecrire en O'Caml la soustraction de 2 polynômes.
4. Ecrire en C une fonction pour la dérivée seconde.
5. Appeler cette fonction à partir d'O'Caml.

1.7 ocamllex

```
{
entetete ML
}
regles lex
```

fichier : `nom.mll`, `ocamllex` produit un fichier `nom.ml` de ce fichier.
module `Lexing` :
– enregistrement `lexbuf`
– `from_channel : in_channel -> lexbuf` et `from_string : string -> lexbuf`
– autres fonctions d'accès aux champs de l'enregistrement.
On définit ainsi un type `jeton` et un petit analyseur lexical :

```
type jeton = I of int | V of string | Plus | Moins;;
```

```
{
  open Jeton;; (* module contenant le type jeton *)
  exception Eof
}
rule jeton  = parse
    [' ' '\t']  { jeton lexbuf}
  | ['0'-'9']+  { I(int_of_string (Lexing.lexeme lexbuf)) }
  | '+'         {Plus}
  | '-'         {Moins}
  | ['A'-'Z' 'a'-'z'] ['A'-'Z' 'a'-'z' '0'-'9'] *
                { V(Lexing.lexeme lexbuf)}
```

1. Compiler et tester cet analyseur syntaxique à partir de chaînes de caractères.
2. Modifier cet analyseur en convertissant les noms de variables en majuscule.
3. Modifier le type `jeton` en lui ajoutant les constructeurs constants pour `*`, `/` et les parenthèses.
4. Utiliser cet analyseur sur l'exemple de la calculette.

1.8 ocamlyacc

```
%{
prologue ML
%}
declarations
%%
  regles yacc
%%
  epilogue ML
```

fichier : `nom.mly`, `ocamlyacc` produit un fichier `nom.ml` de ce fichier. module `Parsing` : exception `Parse_error` et localisation des caractères
On définit ainsi le type `expr` et un petit analyseur syntaxique :

```
type expr = C of int |   Plus of expr * expr
                     |   Moins of expr * expr
```

```
%{
open Types;;
print_string "passez par ici\n";;
%}
%token <int> INT
%token PLUS MOINS
%token <string> IDEN
%start debut
%type <expr> debut

%%
debut:
    INT     {C $1}
  | expr PLUS expr { Plus ($1,$3) }
  | expr MOINS expr { Moins ($1,$3) }
;
%%
print_string "2eme passage\n";;
```

1. Ecrire le type `expr` dans le module `Types`, et compiler le.
2. Compiler cet analyseur avec la commande `ocamlyacc`.
3. Visualiser le fichier interface engendré.
4. Modifier l'analyseur lexical précédent pour qu'il puisse être combiné avec l'analyseur syntaxique (le type `jeton` précédent devient le type `token` défini dans le fichier d'interface.).
5. Ecrire l'ensemble de la chaîne de production de valeurs de type `expr` à partir du clavier
6. Augmenter la grammaire pour traiter tous les opérateurs arithmétiques.
7. Ecrire une fonction `eval` qui prend une `expr` e en entrée et calcule sa valeur numérique.

8. On ajoute dans le langage les déclarations globales. Une phrase du langage est donc soit une expression, soit une déclaration.
   (a) Modifier les types et la grammaire de ce langage.
   (b) Modifier la fonction `eval` pour tenir compte de l'environnement global.

1.9 Affichage avec `graphics`

tracé	remplissage	type
`moveto`		`int→int→unit`
`lineto`		`int→int→unit`
	`fill_rect`	`int→int→int→int→unit`
	`fill_poly`	`( int * int) array → unit`
`draw_arc`	`fill_arc`	`int→int→int→int→int→unit`
`draw_ellipse`	`fill_ellipse`	`int→int→int→int→unit`
`draw_circle`	`fill_circle`	`int→int→int→unit`

Pour utiliser la bibliothèque `graphics` avec le toplevel, il est nécessaire sous Unix de créer un nouveau toplevel incluant la bibliothèque. Sous X-window, l'ouverture de la page graphique `open_graph` permet de donner les dimensions de la fenêtre.

1. Créer le toplevel `ocamlgraph` en utilisant la commande `ocamlmktop` et en suivant les instructions du manuel de référence.
2. Ecrire une fonction qui trace un repère orthogonal normé pour un intervalle donné.
3. Ecrire une fonction qui prend en argument une fonction f sur les réels, un intervalle, et un pas et qui dessine sur la page graphique la fonction f.

1.10 Bitmaps et persistants

Bitmaps
– type `image`
– `make_image : color array array -> image` et `dump_image : image -> color array array`
– `draw_image : image -> int -> int -> unit` pour l'affichage d'une image
Persistants
– `input_value : in_channel -> 'a`
– `output_value : 'a -> out_channel -> unit`
Le fichier xxx.dat, provenant de xxx.gif, contient une valeur de type `color array array`. Ce fichier a été créé par la fonction `output_value` qui écrit dans un canal n'importe quelle valeur non fonctionnelle O'Caml.

1. Relire ce fichier par `input_value`. Comment faire pour que la valeur retournée soit de type `color array array` ?
2. Afficher ce dessin après une conversion vers le type `image`.
3. Ecrire une fonction `negatif` qui transforme les pixels blancs en pixels noirs, et réciproquement.
4. Sauver dans un fichier cette nouvelle image.

1.11 Interface avec `graphics`

Evénements `wait_next_even : event list -> status`

```
type event =  Button_down| Button_up
            | Key_pressed| Mouse_motion| Poll;;

type status =
  { mouse_x : int;  mouse_y : int;
    button : bool;  keypressed : bool;
    key : char};;
```

La gestion d'évènements proposée par `Graphics` est vraiment minimum pour la construction d'interfaces interactives. Néanmoins, le code est portable sur différents systèms graphiques comme Windows, MacOS ou X-Window.

– `mouse_pos` : `unit→ int * int` : retoune la position de la souris par rapport à la fenêtre ;
– `button_down` : `unit→bool` indique si un bouton est appuyé ;
– `read_key` : `unit → char`, attente bloquante sur l'appui d'une touche ;
– `key_pressed` : `unit → bool` indique si une touche est appuyée, attente non bloquante.

Etude du jeu `Demineur` *explorer un champ de mines sans marcher sur l'une d'elles.*
Les différents programmes de cette application se trouvent dans le catalogue `/distrib/ocaml/stage` sur le serveur `serveur.formation.jussieu.fr`.

1. Après avoir copié ce catalogue dans votre `HOMEDIR`, compiler puis exécuter cette application.
2. Modifier l'interface pour pouvoir marquer une case en appuyant sur une touche et sur le bouton de la souris.