TP n° 9 : Programmation impérative

Quelques rappels :

- Les références sont créées avec `let identifiant = ref expression`, le contenu de la référence s'obtient avec `!identifiant` et sa modification avec `identifiant := expression` (qui est de type `unit`). L'entier sur lequel pointe une référence de type `int ref` peut être incrémenté ou décrémenté à l'aide des fonctions `incr` et `decr`.
- On peut exécuter des séquences d'expressions, séparées par des virgules, lesquelles seront exécutées dans l'ordre d'affichage. Toutes les expressions sauf la dernière doivent être de type `unit`. L'expression qui en résulte est du type de la dernière expression de la séquence. Si nécessaire, les séquences peuvent être entourées par `begin` et `end`.

OCaml
```
begin
  expression1;
  expression2;
  ... ;
  expressionN
end
```

☞ **Remarque 1**

Une séquence d'expressions est une expression.

- Les boucles non conditionnelles s'écrivent :

OCaml
```
for indice = debut to fin do
  expression
done
```

OCaml
```
for indice = debut downto fin do
  expression
done
```

- Les boucles conditionnelles s'écrivent :

OCaml
```
while condition do
  expression
done
```

⚠

- `if test then e1; e2 else e3` est interprété comme `(if test then e1); e2 else e3` et n'a aucun sens (erreur de syntaxe). Il faut utiliser `begin ... end`.
- `if test then e1; e2` est interprété comme `(if test then e1 else ()); e2` et pas comme `if test then (e1; e2) else ()`. La deuxième expression est toujours évaluée ! Il faut utiliser `begin ... end`.

1 Enregistrements modifiables

Par défaut, les champs des enregistrements ne sont pas modifiables. On peut cependant ajouter le mot-clé `mutable` lors de la définition du type :

OCaml
```
type complexe = {mutable re : float; mutable im : float};;
```

On utilise alors la syntaxe `<-` pour pour modifier la valeur d'un champ :

OCaml
```
let z = {re = 2.0; im = 1.0};;
z : complexe = {re = 2.0; im = 1.0}
z.re <- z.im -. 1.;;
- : unit = ()
z;;
- : complexe = {re = 0.0; im = 1.0}
```

Question 1

Réécrire la fonction `conjuge` du TP précédent, mais cette fois-ci de type `complexe -> unit` et non `complexe -> complexe`.

On propose le type suivant pour un arbre n-aire, dans lequel on autorise les étiquettes des nœuds à être modifiées.

OCaml
```
type 'a arbre = {mutable etiquette : 'a; fils : 'a arbre list};;
```

Question 2

Écrire une fonction `reinitialiser : int arbre -> unit` qui met à 0 toutes les étiquettes d'un arbre n-aire d'entiers.

2 Autour des références

Question 3

Anticiper, tester et comprendre les résultats produits par les instructions suivantes

OCaml
```
let b = 2;;
let double x = b * x;;
let b = 145;;
double 3;;
```

OCaml
```
let b = ref 2;;
let double x = !b * x;;
b := 145;;
double 3;;
```

Les références ne sont rien de plus que des enregistrements à un seul champ mutable, avec des opérateurs (infixes) prédéfinis [1].

OCaml
```
type 'a ref = {mutable contents : 'a};;
let ref = fun v-> {contents = v};;
let ( := ) = fun r v -> r.contents <- v;;
let ( ! ) = fun r -> r.contents
```

Question 4

Réécrire les quatre phrases CAML du deuxième exemple en utilisant uniquement des enregistrements et des opérations sur les enregistrements.

Question 5

Déterminer le type [a] puis écrire une fonction `echange` échangeant les contenus de deux références.

a. Ne trichez pas !

3 Suite de Fibonacci

Question 6

Écrire une fonction impérative `fibonacci : int -> int` calculant le terme d'indice n de la suite de Fibonacci définie par $F_0 = 0$, $F_1 = 1$ et pour tout $n \geqslant 0$, $F_{n+2} = F_n + F_{n+1}$, en utilisant deux références.

Question 7

Écrire une fonction impérative `fibonacci_bis : int -> int` calculant le terme d'indice n de la suite de Fibonacci en utilisant une seule référence. *Indication : utiliser une référence de couple.*

4 Algorithme d'Euclide

Question 8

Écrire une fonction `euclide : int -> int -> int` programmant la version impérative de l'algorithme d'Euclide [a].

a. Savez-vous que le **théorème de Lamé** dit que le nombre de divisions nécessaires pour calculer le pgcd de a et de $b < a$ par l'algorithme d'Euclide est majoré par $1 + \lfloor \ln b / \ln \varphi \rfloor$ (qui est lui-même plus petit que 5 fois le nombre de chiffres de b en base 10) où φ est le nombre d'or, majorant atteint pour le calcul du pgcd de deux termes successifs de la suite de Fibonacci ?

1. Mettre le nom d'un opérateur infixe entre parenthèses permet de considérer sa version préfixe, par exemple `( + ) 3 4 = 3 + 4`.

Question 9

Modifier la fonction pour qu'elle affiche les divisions euclidiennes successives. Par exemple :

OCaml
```
euclide_affiche (fibonacci 11) (fibonacci 10);;
89 = 55 x 1 + 34
55 = 34 x 1 + 21
34 = 21 x 1 + 13
21 = 13 x 1 + 8
13 = 8 x 1 + 5
8 = 5 x 1 + 3
5 = 3 x 1 + 2
3 = 2 x 1 + 1
2 = 1 x 2 + 0
- : int = 1
```

Indication : on pourra définir la fonction suivante :

OCaml
```
let affiche_division a b =
  print_int a;
  print_string " = ";
  print_int b;
  print_string " * ";
  print_int (a / b);
  print_string " + ";
  print_int (a mod b);
  print_newline ()
;;
```

5 Nombre et somme des chiffres en base b

On propose la fonction suivante :

OCaml
```
let myst f n b =
  let r = ref n in
  let res = ref 0 in
  while !r <> 0 do
    let c = !r mod b in
    res := !res + (f c);
    r := !r / b
  done;
  !res
;;
```

Question 10

Que fait cette fonction ?

Question 11

En déduire des fonctions `nombre_de_chiffres : int -> int -> int` et `somme_des_chiffres : int -> int -> int` calculant le nombre de chiffres et la somme des chiffres d'un entier dans une certaine base.

Question 12

Modifier ce qui doit l'être pour que ces fonctions affichent les chiffres de l'entier au fur et à mesure qu'ils sont calculés. Par exemple :

OCaml
```
somme_des_chiffres 145792 10;;
2 9 7 5 4 1
- : int = 28
```

☞ **Remarque 2**

Ce n'est pas grave si le dernier chiffre est suivi d'une espace.

6 Exponentiation : encore et toujours...

Question 13

Écrire une version itérative du calcul de x^n de manière naïve.

On souhaite implémenter l'exponentiation rapide [2] en itératif. Le principe sera le suivant : on remarque que

$$x^{2^{k+1}} = \left(x^{2^k}\right)^2$$

Il suffit donc d'une multiplication pour passer de x^{2^k} à $x^{2^{k+1}}$.

L'idée est alors de décomposer n en base 2. Si n s'écrit

$$n = b_0 + b_1 \cdot 2 + b_2 \cdot 2^2 + \cdots + b_k \cdot 2^k$$

avec $b_0, \cdots, b_k \in \{0, 1\}$ (écriture en base 2) alors

$$x^n = x^{b_0} \left(x^2\right)^{b_1} \cdots \left(x^{2^k}\right)^{b_k}$$

le calcul de x^{2^p} s'effectuant à partir de celui de $x^{2^{p-1}}$.

Par exemple,

- $19 = 1 + 2 + 2^4$ et
$$x^{19} = x \times x^2 \times \left(\left(\left(x^2\right)^2\right)^2\right)^2$$
(6 multiplications nécessaires),
- $39 = 1 + 2 + 2^2 + 2^5$ et
$$x^{39} = x \times x^2 \times \left(x^2\right)^2 \times \left(\left(\left(\left(x^2\right)^2\right)^2\right)^2\right)^2$$
(8 multiplications nécessaires).

2. C'est en fait la dérécursification de la version récursive (terminale) de l'exponentiation rapide.

Question 14

Compléter le modèle suivant pour implémenter cette version efficace.

OCaml
```
let exponentiation_rapide x n =
  let resultat = [...] in
  let puissance = [...] in
  let reste = [...] in
  while [...] do
    if !reste mod 2 = 1 then begin
      [...]
    end;
    reste := [...];
    puissance := [...];
  done;
  !resultat
;;
```

7 Test de primalité, nombre et somme des diviseurs et nombres parfaits

Question 15

Écrire une fonction `est_premier : int -> bool` testant si un entier est premier.

Question 16

Écrire des fonctions `nombre_diviseurs : int -> int` et `somme_diviseurs : int -> int` renvoyant respectivement le nombre et la somme des diviseurs positifs d'un entier et une fonction `est_parfait : int -> bool` testant si un nombre est parfait, c'est-à-dire s'il est égal à la somme de ses diviseurs stricts.