TD 5

1 Portée

1. (Variables statiques.)
   (a) Quelle est la différence entre la fonction `find_max` décrite ci-dessous et la même fonction où `maxl`, `maxr`, et `k` seraient déclarés `static` ?

```
int find_max (int a[], int i, int j)
{
  int maxl, maxr, k;

  if (i==j)
    return a[i];
  k = (i+j)/2; /* note: quotient de la division,
                  pas division exacte */
  maxl = find_max (a, i, k);
  maxr = find_max (a, k+1, j);
  return (maxl > maxr)?maxl:maxr;
}
```

   (b) Supposons que j'écrive le code suivant en Caml :

```
let maxl = ref 0 in
let maxr = ref 0 in
let rec find_max a i j =
   if i=j
      then a.(i)
   else let k = (i+j)/2 in begin
            maxl := find_max a i k;
            maxr := find_max a (k+1) j;
            if !maxl > !maxr then !maxl else !maxr
        end;;
```

   Quelle variante de la fonction `find_max` de la fonction précédente implémente-t-il ? Autrement dit, quelles sont les variables déclarées « static » dans ce cadre ?

2. (Portée statique.) Que retourne chacune des expressions Caml suivantes ?
   (a)
```
let x=3 in
let y=x+1 in
let x=12 in
  x+y
```
   (b)
```
let x=3 in
let f y = x+y in
let x=4 in
  f 5
```
   (c)
```
let f = (let x=3 in fun y -> x+y) in
  f 4
```
   (d)
```
let f = (let x=3 in
          let y=x+1 in
            fun x -> x+y) in
let x=5 in
  f x
```

3. (Portée dynamique.) Certains dialectes de Lisp, notamment MacLisp et EmacsLisp, utilisent la règle de *portée dynamique*.

   (a) Pour l'expliquer, on va la simuler en Caml. La construction `(let ((i e)) body)` de ces dialectes de Lisp est typiquement équivalente à `fluid_let i e (fun () -> body)` où `fluid_let` est décrit comme suit en Caml :

   ```
   type 'a variable = 'a ref;;
   let rec mkvar v = ref v;;

   let rec fluid_let (x: 'a variable) (e: 'a) (body: unit -> 'b) =
     let save_x = !x in
     let result = (x := e; body ()) in
     begin
       x := save_x;
       result
     end;;
   ```

   Expliquer en français le principe de fonctionnement de `let` dans ces dialectes de Lisp.

   (b) On rappelle que Caml est un langage à portée statique (liaison lexicale). Quelle est la différence entre les deux expressions suivantes en Caml ?

   ```
                                let i = mkvar 0;;
   let i = 7 in                 fluid_let i 7              (fun () ->
   let f = fun x -> x+i in      fluid_let f (fun x -> x + !i)
   let i = 0 in                                            (fun () ->
      f 3                       fluid_let i 0              (fun () ->
                                    !f 3)))
   ```

   (c) Les deux programmes Lisp et Caml suivants semblent calculer la même chose (`setq` est l'affectation en Lisp) :

   ```
   (setq i 7)                       let i = 7;;
   (defun f (x) (+ x i))            let rec f x = x+i;;
   (let ((i 0))                     let i = 0 in
        (f 3))                          f 3;;
   ```

   Pourquoi ces deux programmes retournent-ils des résultats différents ? Et, au passage, quels sont-ils ?

   (d) En Caml, on peut aussi lancer des exceptions, et l'on pourrait par exemple écrire :

   ```
   let i = mkvar 0;;
   try
     fluid_let i 12 (fun () -> raise Failure "arg")
   with Failure _ -> !i
   ```

   Que vaut `!i` à la fin de l'exécution de ce code ? Quel est le problème ? Comment le corrigeriez-vous ?

   (e) Et si l'on a aussi des *threads* dans le langage ?

4. La portée dynamique ou la portée lexicale ne s'applique pas qu'aux variables déclarées par une construction `let`, mais aussi aux arguments des fonctions. Sachant que dans les variantes de Lisp ci-dessus, les arguments des fonctions sont aussi à portée dynamique, quel serait l'équivalent en Caml des déclarations suivantes ?

   ```
   (setq i 1)
   (defun g (y)  (+ x y))
   (defun f (x)  (+ (g x) i))
   ```

   Quel est le résultat de `(f 33)` ?

5. L'α-*renommage* consiste à renommer les variables liées (les arguments de fonction, les variables introduites par `let`, notamment). C'est une notion intuitivement simple, mais difficile à définir correctement, comme on le verra au premier cours de λ-calcul. On va donc supposer une compréhension intuitive de la notion dans cette question.

   Dans un langage à liaison lexicale, α-renommer une expression ne change pas la valeur qu'elle calcule. Montrer, par un contre-exemple adapté en Lisp, que ce n'est plus vrai pour un langage à portée dynamique.
6. Scheme est une variante de Lisp à portée statique, mais définit une macro `fluid-let`, similaire à `fluid_let`. Common Lisp utilise la portée statique par défaut, mais permet de définir des variables à portée dynamique. Quel intérêt ?

2 Appel par nom, par valeur, par référence

7. Que fait la fonction C suivante ?

```
void swap (int x, int y){
  int z;
  z = x;  x = y;  y = z;
}
```

8. Comment peut-on la corriger ? En C++ ? En Pascal ? En Java ?
9. En Fortran, la seule façon de passer des paramètres est par référence. Écrivons :

```
      SUBROUTINE SWAP (I, J)
      INTEGER K
      K=I
      I=J
      J=K
      END
```

   Quels sont les effets des extraits de code suivants, démarrés avec `I`=3, `J`=7 ?

   (a) `CALL SWAP(I,J)`

   (b) `CALL SWAP(I+0,J)`

   (c) `CALL SWAP(I+0, J*1)`

   Qu'en concluez-vous ?
10. (Macros.) En C, on peut écrire aussi bien la fonction de gauche que la macro de droite.

```
int abs (int i){
  if (i < 0)
    return -i;
  else return i;
}
```

```
#define ABS(i) ((i)<0)?(-(i)):(i)
```

   (a) Avant de commencer, pourquoi ai-je mis des parenthèses autour des trois instances de `i` dans la définition de `ABS` ?

   (b) Quelle est la différence entre `abs(i++)` et `ABS(i++)` ? Le passage des paramètres á une fonction C est dit en appel *par valeur*, et le passage des paramètres à une macro simule ce qu'on appelle l'appel *par nom*.

   (c) Il y a deux façons d'écrire un caractère `c` sur un fichier `f` en C, en appelant `putc()` ou `fputc()`. Voici une possibilité de les implémenter (ignorant les erreurs) :

```
#define putc(c,f) do{ \
    int __i = (f)->buflen; \
    if (__i>=MAXBUFLEN) { fflush(f); __i=0; } \
```

```
    (f)->buf[__i++] = c; \
    (f)->buflen = __i; \
  } while (0)
void fputc (int c, FILE *f) {
  putc (c, f);
}
```

Quelle différence y a-t-il entre les deux ?

(d) Questions subsidiaires : à quoi servent les '\' en fin de ligne ? Quel est le problème que pose la variable `__i` dans le code ci-dessus ? A quoi sert le `do ... while (0)` dans la définition de `putc` ?

11. Le langage Algol-60 dispose aussi bien de l'appel par nom (par défaut) que de l'appel par valeur (via le mot-clé `value`).

(a) Que fait, intuitivement, le programme suivant, aussi connu sous le nom de *Jensen's device* ?

```
real procedure Sum(k, l, u, ak)
      value l, u;
      integer k, l, u;
      real ak;
   begin
      real s;
      s := 0;
      for k := l step 1 until u do
         s := s + ak;
      Sum := s
   end;
```

(b) D'après vous, que calcule `Sum (i, 1, 100, V[i])` ?

(c) Grâce à l'appel par nom, on peut aussi écrire l'échange de deux données, comme avec l'appel par référence :

```
procedure swap(a, b)
  integer a, b;
  begin
    integer temp;
    temp := a;
    a := b;
    b := temp;
  end;
```

Mais que fait `swap(i,A[i])` ?

12. Pour chacune des expressions suivantes, indiquer combien de fois est évaluée l'expression `e` en appel par valeur, par nom, et par nécessité.

(a) `let f x = x+1 in f e`

(b) `let f x = x+x in f e`

(c) `let f x = 43 in f e`

(d) `let f x = e+x in f 4`

(e) `let f x = x() + 1 in f (fun () -> e)`

(f) `let f x = x() + x () in f (fun () -> e);`

(g) `let f x = 43 in f (fun () -> e).`

13. (Appel par nécessité.) Voici une implémentation des thunks (ou promesses) en C. Une autre est donnée en Caml dans le poly de cours numéro 4.

```
typedef struct thunk{
  int thawed;
  union {
    void *value;
    void *(*compute) (void);
  } what;
} thunk;

thunk *delay (void *(*compute) (void)){
  thunk *t = malloc (sizeof (thunk));
  t->thawed = 0;
  t->what.compute = compute;
  return t;
}

void *force (thunk *t){
  if (!t->thawed) {
    t->what.value = (*t->compute) ();
    t->thawed = 1;
  }
  return t->what.value;
}
```

On définit le type des listes paresseuses d'entiers par :

```
typedef struct stream{
  int head;
  thunk *next;
} stream;
```

où `next` est un thunk encapsulant le reste de la liste.

(a) Ecrire les fonctions

- `int hd (stream *s);`
- `stream *tl (stream *s);`
- `stream *cons (int i, stream *next).`

On utilisera `force` et `delay` là où c'est nécessaire ; il est interdit d'appeler `malloc` ou d'aller lire les champs des thunks directement !

(b) Écrire un extrait de programme C qui fabrique une liste paresseuse infinie ne contenant que des 1.

(c) Écrire une fonction C qui prend une liste paresseuse d'entiers, et retourne la liste des mêmes entiers auxquelles on a ajouté 1.

(d) En déduire un bout de programme C qui construise la liste paresseuse de tous les entiers naturels (on ignorera le fait que `int` ne contient que des entiers machine...).

(e) Tenter d'écrire une fonction C qui prend en entrée un entier `n` et produit la liste paresseuse de tous les multiples de `n`. Quel est le problème ? Pourquoi n'aurait-on pas ce problème si on avait codé les thunks en Caml, comme dans le cours ? Comment pourrait-on le corriger ?

14. En Haskell, langage dit paresseux (=en appel par nécessité), on peut écrire :

```
hamming :: [Integer]
hamming = 1 : mergeUnique (map (2*) hamming)
                          (mergeUnique (map (3*) hamming)
                                       (map (5*) hamming))
```

où `mergeUnique` fusionne deux listes paresseuses (possiblement infinies) triées en ordre croissant, et retourne la liste union des deux, ordonnée en ordre croissant, et avec les doublons supprimés.

Pour comprendre cette fonction, il faut noter que la construction d'une liste `a:b` termine tout de suite. Ce n'est que si on appelle `head` ou `tail` dessus qu'on forcera l'évaluation de `a` ou de `b` respectivement.

(a) Que vaut la liste `hamming` ?

(b) Qu'obtiens-je si je demande `hamming !! 1`, `hamming !! 2`, etc. ? (`!!` retourne le $n$ième élément d'une liste, `l !! 1` est donc un équivalent de l'appel `head l`, et `l !!` $n+1$ un équivalent de `tail (l !!` n`)`)

(c) Si je demande la valeur de `hamming !!` n, quelle est la partie de la liste des nombres de Hamming qui aura été effectivement calculée ?

(d) Écrire la fonction `mergeUnique`.