Programmation 1

TD n°5

13 October 2020

1 Lexical scope, dynamic scope, variables

Exercise 1 : Static variables

1. What is the difference between the function `find_max` described below and the same function where `maxl`, `maxr`, and `k` would be declared `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;
}
```

2. Suppose I write the following code in 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;;
```

   Which variant of the function `find_max` declared previously does it implement ? In other words, what are the variables declared « static » in this snippet ?

Exercise 2 : Static scope

What does each of the following Caml expressions return ?

1.
```
let x=3 in
let y=x+1 in
let x=12 in
  x+y
```

2.
```
let x=3 in
let f y = x+y in
let x=4 in
  f 5
```

3.
```
let f = (let x=3 in fun y -> x+y) in
  f 4
```
4.
```
let f = (let x=3 in
           let y=x+1 in
             fun x -> x+y) in
let x=5 in
  f x
```

Exercise 3 : Dynamic scope

Some dialects of Lisp, notably MacLisp and EmacsLisp, use the rule of dynamic scope.

1. To explain it, we will simulate it in Caml. The construction `(let ((i e)) body)` of these Lisp dialects is typically equivalent to `fluid_let i e (fun () -> body)` where `fluid_let` is written as follows in 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;;
```

   Explain the working principle of `let` in these dialects of Lisp.

2. We recall that Caml is a language with a static scope (lexical binding). What is the difference between the following two expressions in Caml ?

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

3. The two snippets in the following section seem to calculate the same thing (`setq` is an assignment in 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;;
```

   Why do these two programs return different results ? And what do they return ?

4. In Caml, we can also throw exceptions, and we could for example write :

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

   What is `!i` at the end of the execution of this code ? What is the problem ? How would you correct it ?

Exercise 4 : Dynamic scope and functions

The dynamic scope or the lexical scope does not apply only to variables declared by a `let` construct, but also to function arguments. Knowing that in the variants of Lisp above, the arguments of the functions are also with dynamic scope, what would be the Caml equivalent of the following statements ?

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

What is the result of `(f 33)` ?

2 Call by name, by reference, by value

Exercise 5 : Study of different languages

1. What does the following C function do ?
```
void swap (int x, int y){
  int z;
  z = x;  x = y;  y = z;
}
```
2. How can we correct it ? In C++ ? In Pascal ? In Java ?
3. In Fortran, the only way to pass parameters is by reference.
```
SUBROUTINE SWAP (I, J)
INTEGER K
K=I
I=J
J=K
END
```
What are the effects of the following snippets, starting with `I`=3, `J`=7 ?
   (a) `CALL SWAP(I,J)`
   (b) `CALL SWAP(I+0,J)`
   (c) `CALL SWAP(I+0, J*1)`
   What can you conclude ?

Exercise 6 : Macros

In C, we can write both the function on the left and the macro on the right.
```
int abs (int i){
  if (i < 0)
    return -i;
  else return i;
}
```
```
#define ABS(i) ((i)<0)?(-(i)):(i)
```

1. Before starting, why did I put parentheses around of the three instances of `i` in the definition of `ABS` ?
2. What is the difference between `abs(i++)` and `ABS(i++)` ? Passing parameters to a C function is known as call by value, and the passing of the parameters to a macro simulates what is known as call by name.
3. There are two ways to write a character `c` to a file `f` in C, by calling the function `putc()` or `fputc()`. Here is a possibility to implement them (ignoring errors) :
```
#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);
}
```

What is the difference between them ?

4. Subsidiary questions : what is the purpose of '\' at the end of the line ? What is the problem posed by the variable `__i` in the above code ? What does the snippet `do ... while (0)` do in the definition of `putc` ?

Exercise 7 : Algol-60

The Algol-60 language also has the call by name (by default) in addition to the call by value (via the keyword `value`).

1. Intuitively, what does the following program, also known as Jensen's device, do ?

```
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;
```

2. According to you, what is the value of `Sum (i, 1, 100, V[i])` ?
3. Thanks to call by name, we can also write the exchange of two data, as with call by reference :

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

But what does `swap(i,A[i])` do ?

Exercise 8 : Effectiveness of evaluation strategies

For each of the following expressions, indicate how many times `e` is evaluated in call by value, by name, and by need.

1. `let f x = x+1 in f e`
2. `let f x = x+x in f e`
3. `let f x = 43 in f e`
4. `let f x = e+x in f 4`
5. `let f x = x() + 1 in f (fun () -> e)`
6. `let f x = x() + x () in f (fun () -> e)` ;
7. `let f x = 43 in f (fun () -> e)`.

Exercise 9 : Lazy evaluation in Haskell

In Haskell, a language called lazy (= in call by need), we can write :

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

where`mergeUnique` merges two lazy lists (possibly infinites) sorted in ascending order, and returns their union (as a list), ordered in ascending order, and with the duplicates removed.

To understand this function, it should be noted that the construction of a list `a:b` finishes right away. It is only if we call `head` or `tail` above that we will force the evaluation of `a` or of `b` respectively.

1. What does the list `hamming` evaluate to ?
2. What do you obtain if you type `hamming !! 1`, `hamming !! 2`, etc. ? (`!!` returns the nth element of a list, `l !! 1` is therefore equivalent to calling `head l`, and `l !!` $n+1$ is equivalent to `tail (l !!` n`)`)
3. If you ask for the value of `hamming !!` n, what part of the list of Hamming numbers will be actually calculated ?
4. Write the function `mergeUnique`.