Programmation 1, leçon 4

Jean Goubault-Larrecq
ENS Cachan
goubault@lsv.ens-cachan.fr

18 octobre 2016

1 Fonctions

La notion de *fonction* est fondamentale en programmation. On peut par exemple écrire en C :

```
int lsb_mask (int n)
{
  if (n==0)
    return 0; /* false */
  else return n & -n;
}
```

ce qui permettra d'écrire `lsb_mask`(e) pour calculer le « lsb_mask » de l'expression entière e, ce qui sera bien plus clair que d'écrire `(`e`==0)?0:(`e` & - `e`)`. (La syntaxe a`?`b`:`c est le `if-then-else` au niveau des expressions, et calcule b si a est vrai [non nul], c sinon.)

L'usage de fonctions remplit donc un rôle de *documentation*, et améliore la lisibilité d'un programme. Elle n'apporte rien en termes d'expressivité : tout programme peut être écrit sans aucune fonction ! Ceci sera apparent lorsque nous verrons comment les fonctions sont réalisées en assembleur, lequel n'a pas de notion de fonction.

Ici, bien sûr, je ne vous ai pas dit ce qu'était le « lsb_mask » d'une expression. Je vous laisse le découvrir dans l'exercice 1.1.

▸ **EXERCICE 1.1**
Que fait la fonction `lsb_mask` ci-dessus ? Expérimentez sous `gdb`, proposez une théorie et démontrez-la. On supposera une arithmétique en complément à deux.

L'usage de fonctions permet aussi d'éviter certains bugs. Si l'expression e n'a aucun effet de bord (ne modifie aucune variable, n'agit pas sur le monde extérieur), alors on peut effectivement penser que `lsb_mask`(e) et `(`e`==0)?0:(`e` & -`e`)` sont deux expressions équivalentes.

Ce serait vrai si C avait une discipline d'appel des fonctions en *appel par nom*, mais C utilise l'*appel par valeur*. Comme ce dernier est le mode qui devrait vous sembler le plus naturel, nous commencerons par celui-ci en section 1.1.

En d'autres termes, lorsqu'en C on appelle `lsb_mask`(e), le programme va d'abord calculer la valeur du paramètre e, puis la passer à `lsb_mask`. L'appel `x = lsb_mask`(e) serait donc plutôt équivalent à écrire :

```
{
  int aux; /* nom de variable frais */

  aux = e;

  x = (aux==0)?0:(aux & -aux);
}
```

Les fonctions s'appellent de noms variés : fonctions et *procédures* en Pascal (selon qu'elles retournent, ou non, un résultat—une procédure, ne retournant aucun résultat, serait simplement déclarée comme une fonction retournant le type `void` en C), *sous-programmes* en divers autres langages.

1.1 Appel par valeur : le cas de C, Caml, Java, etc.

L'appel par valeur est le mode de passage de paramètres le plus courant. Il consiste à calculer d'abord les valeurs des paramètres, puis à appeler la fonction souhaitée avec ces valeurs pour ses paramètres formels (`n` dans le cas de la fonction `lsb_mask`). La fonction appelée fait les calculs spécifiés par son code source, puis retourne une valeur, dite *valeur de retour* (sauf si le type de retour est `void`).

La façon dont tout ceci se compile typiquement sur les Intel 32 bits va expliquer nombre de points qui seraient difficiles à comprendre sinon.

Tout d'abord, la pile joue un rôle fondamentale. Prenons l'exemple d'une fonction à deux paramètres :

```
int plus (int a, int b)
{
  return a+b;
}
```

que nous appelons, par exemple dans `main` :

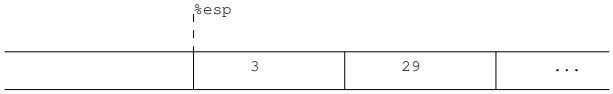
```
int main (int argc, char *argv[])
{
  int x;

  ...
  x = plus (argc, strlen (argv[1]));
  ...
}
```

L'*appelant* (ici, `main`) va d'abord calculer les valeurs de ses paramètres. On appellera ces paramètres les *paramètres réels*, pour les distinguer des paramètres dits formels que nous verrons plus bas.

Ces paramètres réels peuvent être de simples variables (`argc`) ou des expressions plus compliquées, qui peuvent elles-mêmes impliquer d'autres appels de fonctions (`strlen`, qui calcul la longueur de la chaîne de caractères qu'on lui passe en paramètre, ici `argv[1]`).

Imaginons que `argc` vaille 3 et que `strlen (argv[1])` ait retourné 29. Juste avant l'appel à `plus`, la fonction `main` va empiler les valeurs 3 et 29 sur la pile, autrement dit, passer de la pile :

à la pile :



A noter que pour arriver à ce résultat, on doit empiler *d'abord* 29 puis 3, ce qui peut sembler troublant. Les implémentations de Pascal empilent traditionnellement leurs arguments dans l'ordre intuitif, d'abord 3, puis 29, ce qui a l'inconvénient que le premier paramètre est celui qui est le plus profond dans la pile, et que le dernier est le premier.

▸ **EXERCICE 1.2**
Quel est l'autre inconvénient de la convention d'ordre des paramètres de Pascal ? A titre d'indication, tapez `man stdarg` sous la ligne de commande Unix.

L'appelant (`main`, donc, dans notre exemple), va ensuite appeler `plus` (l'*appelée*) par une instruction telle que :

```
call plus
```

(Pour l'assembleur, `plus` est juste une adresse, l'adresse du début du code de la fonction `plus`.) L'effet de l'instruction assembleur `call` est double :

— elle empile l'*adresse de retour*, qui est l'adresse du code juste après l'instruction `call` ;
— elle va ensuite à l'adresse `jump`.

En donnant les adresses explicitement, imaginons que nous ayons :

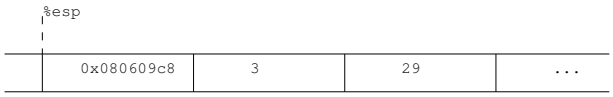
```
0x080609c3:  call 0x0805092c <plus>
0x080609c8:  ; suite du code
```

et :

```
0x0805092c:  pushl %ebp         ; debut du code de 'plus'
0x0805092d:  movl %esp, %ebp
0x0805092f:  movl 8(%ebp), %eax
```

```
0x08050932:  addl 12(%ebp), %eax
0x08050935:  leave
0x08050936:  ret
```

Lorsque `main` effectue l'appel `call plus`, la pile contient maintenant :



On voit que l'adresse de retour (`0x080609c8`) a maintenant été empilée. Elle permettra à l'appelée de savoir où retourner, à la fin de son propre code : l'instruction `ret` dépile la valeur sauvegardée au sommet de la pile et la stocke dans le registre `%eip` de compteur de programme, effectuant ainsi un saut à l'adresse de retour.

▸ **EXERCICE 1.3**
Y a-t-il une différence entre `call plus` ci-dessus et la suite de deux instructions `pushl $0x080609c8`, `jmp plus` ? Si oui, laquelle ?

▸ **EXERCICE 1.4**
Y a-t-il une différence entre `ret` et la suite de deux instructions `popl %ebx`, `jmp *%ebx` ? Si oui, laquelle ? On rappelle que `jmp *` est le saut indirect (saut à l'adresse contenue dans le registre, ici `%ebx`).

L'instruction `leave` est exactement synonyme de la suite de deux instructions `movl %ebp, %esp` et `popl %ebp`. Elle est donc l'exacte symétrique des instructions `pushl %ebp` et `movl %esp, %ebp` du début de l'appelée.

L'idée est la suivante : comme l'appelée est autorisée à utiliser la pile à volonté, on réserve le registre `%ebp` pour pointer vers une adresse fixe tout au long de l'appelée. Mais l'on doit d'abord sauvegarder `%ebp`, pour la restaurer en fin d'appelée, sinon l'appelant verrait sa propre valeur de `%ebp` modifiée.

On peut ensuite récupérer le premier paramètre réel de l'appelée en `%ebp` plus 8 octets (4 octets pour la sauvegarde de `%ebp`, 4 octets pour l'adresse de retour), le second paramètre réel en `%ebp` plus 12 octets, et ainsi de suite.

Du point de vue de l'appelée, ce qu'on trouve en `%ebp` plus 8 octets est le premier *paramètre formel* (la variable `a` de la fonction `plus`), et nous trouvons 4 octets plus loin le second paramètre formel (`b` ici). L'appel de `plus` a donc pour effet de *lier* (voir la section 2) les paramètres formels `a`, `b` aux valeurs 3, 29 des paramètres réels `argc`, `strlen (argv[1])`.

Les variables locales (voir la section 2.1), qui sont allouées sur la pile, seront accédées aux adresses `%ebp` moins 4 octets, moins 8 octets, etc.

Finalement, la valeur de retour est par convention stockée dans le registre `%eax` avant d'appeler la séquence `leave`, `ret`.

▸ **EXERCICE 1.5**
Pourquoi ne place-t-on pas la valeur de retour sur la pile ?

Toutes ces conventions changent sur les Intel 64 bits : voir l'annexe B de la leçon 2. Elles changent bien sûr encore davantage sur les architectures MIPS, ou ARM, ou d'autres processeurs encore.

Ce qui reste commun est que C passe les paramètres *par valeur* : les paramètres réels sont *copiés* (sur la pile) à l'emplacement ou la fonction appelée s'attend à trouver les valeurs de ses paramètres formels.

1.2 Appel par référence : le cas de Pascal, de C++

En C++, on peut aussi déclarer certains paramètres formels comme étant passés *par référence*. Prenons l'exemple de la fonction :

```
void inc (int &n)
{
  n++;
  /* 'return;' implicite ici */
}
```

C'est une fonction d'incrémentation de l'expression passée en paramètre. On pourra donc écrire `inc(m)`, ou bien `inc(a[i])`, et ceci sera équivalent à écrire `m++` ou `a[i]++` respectivement.

Pour ceci, ce n'est pas la *valeur* du paramètre réel (`m` ou `a[i]`) que l'on empilera, mais son *adresse*. Ainsi on pourrait simuler l'effet de la fonction `inc` en appel par valeur en utilisant des pointeurs (ou des références, en Caml).

On écrirait donc en C (appel par valeur uniquement) :

```
void inc (int *np)
#define n (*np)
{
  n++;
  /* 'return;' implicite ici */
}
#undef n
```

(L'usage de macros permet de garder un code similaire au code C++ ci-dessus : `n` sera remplacé partout par `(*np)`... textuellement, au passage, d'où l'intérêt des parenthèses autour de `*np`.)

Et, toujours en C, plutôt que d'appeler `inc(m)` ou `inc(a[i])`, on écrira `inc(&m)`, resp. `inc(&a[i])`, ce qui a pour effet d'empiler l'*adresse* du paramètre réel plutôt que sa valeur.

▸ **EXERCICE 1.6**

L'appel par référence est toujours défini par un procédé similaire de compilation vers un appel par valeur à l'aide de pointeurs auxiliaires. Mais l'appel par référence recèle une part d'obscurité. Supposons que je déclare `inc` comme en ci-dessus en C++, avec un passage par référence. Quels sont les effets des appels `inc(m+3)` ? `inc(m+0)` ?

L'appel par référence a été introduit pour la première fois en Pascal. Dans ce cas, seule la syntaxe change. On écrirait l'équivalent de la version C (appel par valeur uniquement) de `inc` comme suit :

```
procedure inc (n : ^Integer);
begin
  n^ := n^+1;
end;
```

alors que l'équivalent de la version C++ s'écrirait :

```
procedure inc (var n : Integer);
begin
  n := n+1;
end;
```

C'est le mot-clé `var` (aussi utilisé pour déclarer des variables, locales ou globales, par ailleurs), qui déclare le paramètre formel `n` comme passé par référence ici.

1.3 Appel par nom : le cas d'Algol

Les langages Algol 60, Algol 68 et Algol W ont été de formidables entreprises de réflexion sur la sémantique des langages de programmation. Aucun de ces langages n'a jamais été réellement utilisé, mais Algol 68 reste la source d'inspiration principale pour Pascal.

Algol disposait d'un autre mécanisme de passage de paramètres, très naturel pour un mathématicien : l'appel *par nom*. Le principe est simple : en appel par nom, si la fonction $f(x)$, de paramètre formel x, est définie par un code source C (où x apparaît, ou pas : c'est le *corps* de la fonction f), alors l'appel $f(e)$ sur un paramètre réel e est *équivalent* au code $C[x := e]$ obtenu en remplaçant, *textuellement*, toutes les occurrences de x dans C par e.

Pour être précis, le remplacement est un peu plus compliqué qu'un pur remplacement textuel en général. Si le langage contient des variables liées, il est nécessaire de prévoir une étape de renommage des variables liées. C'est un problème vicieux, déjà présent dans le λ-calcul, un langage théorique inventé par A. Church dans les années 1930. L'appel par nom d'Algol est dérivé en droite ligne du λ-calcul, où l'appel par nom est l'unique stratégie d'appel.

Ainsi, si la fonction `lsb_mask` du début de la section 1 était codée en appel par nom, `lsb_mask`(e) serait réellement équivalent à `(`e`==0)?0:(`e` & -`e`)`.

En attendant, le préprocesseur de C permet une forme restreinte d'appel par nom. (Il n'y manque que la possibilité de récursion, qui, si elle existe, est très limitée, et a un fonctionnement pour le moins baroque.)

Si l'on voulait une fonction `lsb_mask` en appel par nom, on pourrait écrire :

```
#define lsb_mask(n) (((n)==0)?0:((n) & -(n)))
```

Ceci a pour effet de remplacer *textuellement* (vraiment… sans aucun renommage de variable liée, c'est-à-dire ici de variable locale portant le même nom par exemple) toutes les occurrences subséquentes de `lsb_mask`(e), quelle que soit l'expression e, par `(`e`==0)?0:(`e` & -`e`)`.

Les nombreuses parenthèses sont là pour éviter quelques effets fâcheux du remplacement textuel :

▸ **EXERCICE 1.7**
Ecrivons :

```
#define lsb_mask_bug(n) (n==0)?0:(n & -n)
```

Par quoi sont remplacées les expressions `lsb_mask_bug(n+1)` ? `lsb_mask_bug(n^0xff)` ? `lsb_mask_bug(n*3)` ? Font-elles le calcul du « lsb_mask » sur les expressions souhaitées ?

L'appel par nom est de plus au cœur d'une optimisation classique, faite par les compilateurs : l'*inlining*, qui consiste justement à remplacer l'appel d'une fonction f par son corps C, avec substitution des paramètres formels par les paramètres réels. Ce remplacement devant préserver la sémantique, les compilateurs prennent bien sûr un certain nombre de précautions.

1.4 Appel par nécessité : le cas de Haskell

L'appel par nécessité, présent dans les langages dits *paresseux* comme Haskell, est une optimisation de l'appel par nom.

En appel par nom, si l'on calcule $C[x := e]$, et que C contient n occurrences de la variable x, on risque de devoir recalculer e, potentiellement, n fois. C'est gênant, comme on l'a vu, si $n \geq 2$ et que e a des effets de bord. Mais c'est aussi gênant même sans effet de bord, car on perd en efficacité. Le but de l'appel par nécessité est de n'évaluer qu'une fois lorsque $n \geq 1$.

Mais l'appel par valeur a cette propriété : on n'y évalue le paramètre réel e qu'une fois de toute façon, non ? Oui, mais l'appel par nom a un avantage distinct sur l'appel par valeur lorsque $n = 0$, c'est-à-dire lorsque le paramètre formel x n'est *jamais* évalué. Ceci peut paraître un cas rare, mais est l'une des forces des langages paresseux, où l'on peut écrire des programmes absolument remarquables fondés sur cette idée (voir exercices).

L'appel par nécessité est une optimisation de l'appel par nom, où si l'on ne demande jamais à évaluer x, on ne calculera jamais e, et si l'on demande à évaluer x n fois avec $n \geq 1$, alors on calculera e *exactement* une fois.

On peut simuler l'appel par nécessité dans un langage à appel par valeurs à l'aide de la technique des *thunks*, aussi appelées parfois *promesses*. Faisons-le en Caml, pour changer (et parce que le problème de la libération de l'espace pris par la promesse en mémoire ne se posera pas) :

```
type 'a promise = {mutable value : 'a option; code : unit -> 'a};;
```

où le type `option` est défini dans la bibliothèque standard par :

```
type 'a option = None | Some of 'a;;
```

et représente une valeur de type `'a` (avec le constructeur `Some`), ou une absence de valeur (`None`).

On crée une promesse à partir du code e à évaluer. Pour que Caml évite de l'évaluer, on l'écrit comme une fonction `fun () ->` e : Caml ne calculera e que si on applique cette fois à `()` (de type `unit`). Cette fonction est représentée par l'argument `c` dans la fonction `delay` ci-dessous, qui crée la promesse :

```
let rec delay c = {value = None; code = c};;
```

Pour trouver la valeur de la promesse, on utilise la fonction `force : 'a promise -> 'a` :

```
let rec force p =
  match p with
    {value = None; code = c} ->
      let res = c () in
      begin
        p.value <- Some res;
        res
      end
  | {value = Some res} -> res;;
```

Ainsi, la première fois qu'on force la promesse, on calcule la valeur en appelant la fonction `c`, et on la stocke. Toutes les autres fois, on récupère la valeur stockée.

Pour simuler l'appel de la fonction f de paramètre formel x et de corps C sur l'expression e en *appel par nom*, on peut donc à la place effectuer l'appel d'une fonction f' de paramètre formel x, mais dont le corps C' est obtenu en remplaçant partout x par `force` x, et sur l'expression e' =`delay (fun () ->` e`)` au lieu de e.

Ce qui est fabuleux avec Haskell, et en général dans tout langage paresseux, c'est qu'on peut écrire des programmes qui ne terminent pas dans une sémantique en appel par valeur, mais termine en appel par nécessité.

Par exemple, en Haskell, on peut écrire la définition récursive suivante pour la liste (infinie) des entiers naturels :

```
nat :: [int]
nat = 0 : map (+1) nat
```

La première ligne est la déclaration de type, et dit que `nat` est une liste d'entiers. La seconde dit que `nat` est la liste composée d'abord de l'entier 0, puis de la liste de tous les éléments de `nat`, définie récursivement, auxquels on a ajouté 1.

Ce programme non seulement termine, mais encore sans rien faire d'autre que produire une promesse (si l'on compile Haskell vers un langage en appel par valeurs comme Caml). L'équivalent Caml serait :

```
type 'a lazy_list = Nil | Cons of 'a * 'a lazy_list promise;;
let rec nat_gen () : int lazy_list =
    Cons (0, delay (fun () ->
              lazy_map (fun n -> n+1) (nat_gen ())));;
let nat = nat_gen ();;
```

où `lazy_map` est l'équivalent paresseux du `map` de Haskell :

```
let lazy_map f =
   let rec lmap l =
     match l with
       Nil -> Nil
     | Cons (a, p) -> Cons (f a, delay (fun () -> lmap (force p)))
   in lmap;;
```

et doit être écrit entre la définition du type `lazy_list` et celle de la fonction `nat_gen`.

On voit que `nat` contient une liste avec 0 comme premier élément, et le reste de la liste non évalué :

```
val nat : int lazy_list = Cons (0, {value = None; code = <fun>})
```

Ecrivons les fonctions qui récupèrent le premier argument, et le reste d'une liste respectivement :

```
let hd l =
  match l with
    (Cons (a, _)) -> a
  | Nil -> failwith "hd on empty list";;
let tl l =
  match l with
    (Cons (a, p)) -> force p
  | Nil -> failwith "tl on empty list";;
```

Si l'on demande le deuxième élément de la liste `nat` :

```
hd (tl nat);;
```

on obtient bien 1, et maintenant `nat` vaut :

```
Cons (0,
 {value = Some (Cons (1, {value = None; code = <fun>})); code = <fun>})
```

où l'on voit que `nat` stocke maintenant les deux premières valeurs de la liste infinie.

Tout ceci est bien plus facile à écrire en Haskell…

▸ **EXERCICE 1.8**

Ecrire la liste des nombres premiers en Haskell. (C'est une constante `primes` prédéfinie, mais faites comme si elle n'existait pas.)

▸ **EXERCICE 1.9**

En Haskell, on définit les types suivants :

```
type N = Integer
type Bit = Int
type Cantor = N -> Bit
```

Le type `N` est celui des entiers (mathématiques, c'est-à-dire de longueur arbitraire), et `Bit` est vu comme le type des bits (le langage ne le contraint pas, mais nous promettons de n'utiliser que les valeurs 0 et 1 de ce type). Le type `Cantor` est le type des suites infinies de bits. Que fait la fonction suivante, due à Ulrich Berger ?

```
berger :: (Cantor  -> Bool) -> Cantor
berger p = if p(0 # berger(\a -> p(0 # a)))
           then 0 # berger(\a -> p(0 # a))
           else 1 # berger(\a -> p(1 # a))
```

Disons que `p :: Cantor -> Bool` est *totale* si et seulement si l'application de `p` à n'importe quelle suite infinie de bits termine. Pourquoi `berger p` termine-t-elle lorsque `p` est totale ?

C'est loin d'être évident. La difficulté est qu'il faut notamment montrer que `berger p` termine même lorsque le prédicat `p` est faux de *toutes* les chaînes infinies de bits. Voir Martin Escardó, *Infinite sets that admit fast exhaustive search*, LICS 2007, pour une preuve topologique élégante, et un programme qui fait la même chose en beaucoup plus rapide.

1.5 Appel par unification : le cas de Prolog

Il existe encore d'autres modes de passages de paramètres. Celui du langage logique Prolog est sans doute l'un des plus étranges… et désolé, je ne vais pas vous le décrire formellement.

En Prolog, on écrit des prédicats, comme par exemple :

```
append ([], L, L).
append ([X | L1], L2, [X | L3]) :-
  append (L1, L2, L3).
```

Il s'agit d'une spécification logique d'un prédicat `append` (L_1, L_2, L_3), vrai lorsque la liste L_3 est la concaténation de L_1 et de L_2 : lorsque L_1 est la liste vide `[]` et que L_2 et L_3 sont égales (première ligne), et aussi lorsque L_1 est de la forme `[X | L1]` (ce qu'on noterait `X::L1` en Caml), que L_3 est aussi de la forme `[X | L3]` *avec le même* `X`, et à condition que `L3` soit la concaténation de `L1` et de `L2`$= L_2$.

On peut donc demander à Prolog de vérifier une concaténation :

```
? append ([1, 2], [3], [1, 2, 3]).
yes
```

C'est modérément intéressant. On peut lui faire calculer une concaténation :

```
? append ([1, 2], [3], X).
yes
X = [1, 2, 3]
```

Il le fait en appliquant un algorithme, dit de *résolution*, qui trouve pour quelle(s) valeur(s) de `X` la requête écrite après le point d'interrogation est vraie.

On peut donc aussi lui demander de résoudre des problèmes plus compliqués :

```
? append ([1, 2], Y, [1, 2, 3, 4, 5]).
yes
Y = [3, 4, 5]
```

ou bien :

```
? append ([1, 5], Y, [1, 2, 3, 4, 5]).
no
```

2 Portée, liaison

2.1 Variables locales, globales

Le langage C connaît trois sortes de variables : les variables *globales*, *locales* et *statiques*.

— Une variable *globale* est allouée à une adresse donnée une fois pour toutes (par le *linker* `ld`, sous Unix). On peut y accéder depuis toutes les fonctions, et toutes accèdent à la même case mémoire.

— Une variable *locale* est alloué dynamiquement, typiquement sur la pile (ou dans un registre) lors de l'appel de la fonction (ou du bloc) dans laquelle elle est déclarée. Seule la fonction qui l'a déclarée y a accès, et pour même dire mieux : seule l'instance courante de la fonction appelée y a accès, des instances différentes ayant accès à des variables locales distinctes. (Ce sera plus clair plus bas.)

— Une variable *statique* est une variable globale, mais déclarée à l'intérieur d'une fonction, et uniquement accessible depuis cette fonction.

La *portée* d'une variable est l'ensemble des lignes de code qui y ont accès. Dans le premier cas, on dira que la *portée* de la variable est globale. Dans les deux autres cas (local et statique), la portée est statique.

Considérons le code suivant :

```
int ncalls_to_f = 0; /* variable globale */

int fact (int n)
{
  int i; /* variable locale */

  if (n==0)
    return 1;
  i = fact (n-1);
  return n * i;
}

int f (int n)
{
```

```
  static memo = -1, memres; /* variables statiques */

  ncalls_to_f++;
  if (n==memo)
    return memres;
  memo = n;
  memres = fact (n);
  return memres;
}
```

La variable `ncalls_to_f` est *globale* : toutes les fonctions y ont accès. Ici, `ncalls_to_f` est censée compter combien de fois on a appelé la fonction `f`. C'est une variable initialisée à 0. (On n'est pas obligé d'initialiser une variable globale, ni même n'importe quelle variable.) La fonction `f` y accède pour la mettre à jour, et d'autres fonctions peuvent la consulter, par exemple.

En assembleur, la variable `ncalls_to_f` serait typiquement allouée par :

```
  .comm ncalls_to_f,4,4  ; allocation de 4 octets, alignes sur 4
```

(ce qui omet l'initialisation à 0), et l'incrémentation `ncalls_to_f++` serait effectuée par un code assembleur tel que :

```
movl ncalls_to_f, %eax   ; adressage absolu
addl $1, %eax
movl %eax, ncalls_to_f
```

Les variables `memo` et `memres` sont *statiques*. C'est la même chose que des variables globales—en particulier elles seraient compilées exactement comme si elles étaient globales—à part pour leur *visibilité* : seule la fonction `f` peut y accéder. L'intérêt de cette restriction est que l'on peut être sûr que `memo` est bien la dernière valeur du paramètre donné à `f`, et `memres` est bien la valeur calculée par `f` à ce dernier appel (sauf si `memo` vaut -1).

La variable `i` dans la fonction `fact` est *locale* : seule la fonction `fact` y a accès. Une variable locale est typiquement implémentée sous forme d'une mot (de 32 bits sur une architecture 32 bits) alloué sur la pile. Il est naturel de voir tous les paramètres formels d'une fonction comme ayant une portée locale. En C, ce sont des variables locales (pratiquement) comme les autres.

En assembleur, le code de `fact` ressemblerait donc à :

```
fact:
  pushl %ebp
  movl %esp, %ebp     ; prologue standard
  subl $4, %esp       ; allocation de 4 octets sur la pile pour i

  movl 8(%ebp), %eax  ; récupération de n
  tstl %eax
  jne .fact_1         ; si pas 0, aller en .fact_1
```

```
  movl $1, %eax
  jmp .fact_ret       ; return 1

.fact_1:
  subl $1, %eax       ; %eax == n-1
  pushl %eax          ; on empile le parametre reel n-1
  call fact           ; appel (recursif) de fact
  addl $4, %esp       ; ne pas oublier de depiler les parametres!
                      ; %eax == fact (n-1)
  movl %eax, -4(%ebp) ; on stocke fact (n-1) dans i

  movl 8(%ebp), %eax  ; %eax == n
  movl -4(%ebp), %ebx ; %ebx == i
  mull %ebx, %eax     ; %eax = n*i
.fact_ret:            ; comment retourner de la fonction
  leave
  ret
```

On notera bien qu'il y a non pas une case mémoire pour la variable `i`, mais autant de cases mémoires que d'appels récursifs à `fact`. Ceci permet à chaque instance de `fact` d'avoir sa propre variable `i`, sans crainte qu'une autre instance ne vienne modifier la sienne, comme ce serait le cas avec une variable statique.

▸ **EXERCICE 2.1**

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, j);
  return (maxl > maxr)?maxl:maxr;
}
```

Expérimentez en compilant les deux programmes, et en les testant sur différents tableaux.

La plupart des langages ont des notions de variables globales et locales, peu connaissent la notion de variable statique.

En Pascal et en Caml, en revanche, on a davantage de portées. On a toujours une portée globale. Par exemple, déclarer :

```
let a = 0;;
```

déclare `a` globale. Mais on a aussi des portées locales *imbriquées*. Dans notre implémentation de la fonction `force` en Caml, la variable `p` a comme portée tout le corps de la fonction `force`, alors que la variable `res` n'a comme portée que le code notée entre `begin` et `end`.

Dans le cas de `lazy_map`, on a même défini une *fonction locale* `lmap` : en Caml, les variables locales peuvent être des fonctions ! Il n'y a aucune raison que les fonctions soient globales.

2.2 Portée lexicale, liaison dynamique

Il est évidemment possible de déclarer des variables de même nom. Si on veut pousser le cas à l'extrême, on peut par exemple écrire :

```
int i=0;

int f (int i)
{
  int j=i;

  if (i==0)
   return 1;
  {
    static int i=0;

    if (j++ % 2==0)
      i++;
    {
      int i = j-2;
      return i;
    }
  }
}
```

Quelle est la variable que l'on référence lorsque l'on écrit `i`, alors ?

La règle en C, comme en Pascal ou en Caml ou en Haskell ou en Java, est la *portée lexicale* : la variable à laquelle on se réfère par `i` est celle déclarée dans le plus petit bloc englobant l'utilisation de la variable. (Un *bloc* est tout ce qui est entre `{` et `}`, proprement parenthésés. Par extension, je considère aussi le programme tout entier comme un bloc ici, ainsi que les définitions de fonctions.)

Ainsi le code ci-dessus est-il équivalent au code ci-dessous, où j'ai remplacé les diverses variables de même nom `i` par des variables `i_0`, `i_1`, etc. :

```
int i_0=0;
```

```
int f (int i_1)
{
  int j=i_1;

  if (i_1==0)
   return 1;
  {
    static int i_2=0;

    if (j++ % 2==0)
      i_2++;
    {
      int i_3 = j-2;
      return i_3;
    }
  }
}
```

Certains autres langages, comme Lisp (Emacs Lisp, Common Lisp ; mais pas Scheme, qui est à portée lexicale) sont à *liaison dynamique*. Dans ces langages, une variable *est une case mémoire*. Toutes les variables `i` partagent donc un même emplacement en mémoire.

Ceci n'empêche pas d'avoir des variables globales et des variables locales, avec imbrications arbitraires. Par exemple, en Lisp, on peut écrire :

```
(let ((i (+ n 1)))
     ; ici calcul impliquant i
)
```

ce qui est similaire (mais pas équivalent) au code Caml :

```
let i=n+1 in
  (* ici calcul impliquant i *)
```

Alors qu'en Caml (portée lexicale), ce code serait typiquement compilé en du code assembleur allouant de la place sur la pile pour (la nouvelle variable) `i`, et y stockant la valeur de `n+1`, Lisp (liaison dynamique) fait la suite d'opérations suivantes :

1. sauvegarde l'ancienne valeur de `i` sur la pile ;
2. stocker `n+1` dans `i` ;
3. une fois le calcul impliquant `i` terminé, restaurer l'ancienne valeur de `i`.

De même, le code Lisp :

```
(defun f (i)
       ; ici calcul impliquant i
)
```

similaire au code Caml :

```
let rec f i =
    (* ici calcul impliquant i *)
;;
```

a pour effet, lorsqu'on appelle `f` sur un paramètre réel `e` (en écrivant `(f e)`), de sauvegarder l'ancienne valeur de `i` sur la pile, stocker la valeur de `e` dans `i`, et une fois le calcul du corps de `f` effectué, de restaurer la valeur de `i`.

L'exercice suivant, à méditer, montre la différence entre langages à liaison lexicale et à portée dynamique.

▸ **EXERCICE 2.2**

Les deux programmes Lisp et Caml suivants semblent calculer la même chose :

```
(setq i 7)  ; setq est l'affectation en Lisp
(defun f (x) (+ x i))
(let ((i 0))
     (f 3))
```

```
let i = 7;;
let rec f x = x+i;;
let i = 0 in
    f 3;;
```

Exécutez les deux programmes, et constatez la différence. Expliquez ce qui s'est passé.