Programmation 1

TD n°5

Aliaume Lopez

15 octobre 2019

1 Précédemment dans les pointeurs de l'amour

Exercise 1 : Les tableaux en C/ASM de long en large

1. On considère le programme C suivant :

```
int a[10];

int main(){
  int i, j;

  a[0] = 1;  a[1] = 1;
  for (i=2; i<10; i++)
    a[i] = a[i-2]+a[i-1];
  for (j=9; j>=0; j--)
    printf ("%d\n", a[j]);
  return 0;
}
```

   Le tableau `a[]` est global (toutes les fonctions y ont accès) et *statique* (il est alloué une fois pour toutes : « statique » désigne en général tout ce qui se fait à l'écriture du programme ou lors de sa compilation, par opposition à « dynamique », qui se réfère à ce qui a lieu lors de l'exécution).

   (a) Qu'affiche ce programme ?

   (b) On supposera qu'on tourne sur une architecture MIPS 32 bits, auquel cas le tableau `a[]` occupe 40 octets. Écrire les directives assembleur nécessaires à l'allocation (statique) de 40 octets de mémoire dans la section `.data` et lui donne l'étiquette `a`. On pourra soit utiliser la pseudo-instruction `.word` suivie de 10 zéros, soit la pseudo-instruction `.space` suivie du nombre d'octets à réserver. Il faudra en outre penser à utiliser `.align`.

   (c) Quelle est la différence entre les instructions `.word` et `.space` ?

   (d) Même exercice en assembleur x86 32 bits, en remplaçant `.word` par `dd`, `.space` par `.fill`.

   (e) On rappelle que la donnée `a[i]` réside á l'adresse `a`+4`i` (toujours sur une machine 32 bits). Convertir le programme ci-dessus en code à trois valeurs et le traduire en assembleur MIPS.

   (f) Pareil en assembleur x86.

2. On considère le même programme que ci-dessus, à part le changement suivant dans les premières lignes :

```
int main(){
  int a[10];
  int i, j;

  [...]
```

Dans ces conditions, le tableau `a[]` est alloué *dynamiquement*, chaque fois que l'on rentre dans `main`. Il est aussi désalloué à la sortie de `main`

En pratique, il est alloué en décrémentant le registre de pile (`sp` sur MIPS, `%esp` sur x86) de 40.

(a) Si l'on doit modifier le registre de pile, il vaut mieux le sauvegarder à l'entrée de `main` (dans `fp` sur MIPS, dans `%ebp` sur x86)... et sauvegarder aussi le registre de sauvegarde. Comment le fera-t-on en assembleur MIPS ?

(b) Modifier votre programme assembleur MIPS de la question précédente pour qu'il effectue cette sauvegarde, ainsi que la restauration correspondante en fin de procédure. Il devra aussi allouer 40 octets sur la pile, comme indiqué plus haut, et accéder au tableau `a[]` non plus via le label `a` (qui n'existe plus dans notre nouvelle version) mais aux endroits adéquats de la pile.

(c) Et en assembleur x86 ?

3. On pourrait vouloir optimiser le programme assembleur précédent de sorte à ce qu'il laisse le registre de pile inchangé, et en retrouvant le tableau `a[]` en utilisant des déplacements négatifs à partir de `sp` (resp., `%esp`).

(a) Pourquoi cette stratégie de compilation ne serait-elle pas acceptable si `main()` appelait des fonctions auxiliaires ?

(b) Pourquoi n'est-elle en fait pas acceptable en l'état ? À titre d'indication, savez-vous comment sont gérées les interruptions asynchrones sur les processus modernes, et les signaux sous Unix ?

Exercise 2 : Structures en C, pointeurs sur sturctures

On définit les structures suivantes en C.

```
struct s1 { int i; };
struct s2 { struct s1 s; };
struct s3 { struct s1 *p; };
```

1. Écrire deux fragments de code C qui créent des structures de type `struct s2`, resp. `struct s3`, contenant une structure contenant la valeur 42.

2. Dessiner un diagramme boîtes-flèches de la mémoire après l'exécution du code suivant :

```
struct s1 s1; // oui, on peut donner le meme nom a une
struct s2 s2; // variable et a un type struct...
struct s3 s3, ss3;
s1.i = 42; s2.s = s1;
s3.p = &s1;
ss3.p = malloc (sizeof (struct s1)); ss3.p->i = 54;
```

On différenciera bien le tas de la pile.

3. On définit la fonction suivante :

```
struct s1 *f(void) { // (void) = ne prend pas d'argument
  struct s1 s;
  s.i = 42;
  return &s;
}
```

Que se passe-t-il lorsqu'on exécute cette fonction ? En profiter pour répondre à la question : à quoi sert `malloc()` ?

4. On définit la fonction suivante :

```
struct s3 *f(void) {
  struct s1 s;   s.i = 9;
  struct s3 *p3 = malloc (sizeof (struct s3));
  p3->p = &s;
  return p3;
}
```

   Que se passe-t-il lorsqu'on exécute cette fonction ? Que doit-on faire pour corriger le problème ?

5. Quelle est la différence entre les deux déclarations de structures suivantes ? La question ne porte pas tant sur l'absence ou la présence d'un `*`, mais sur la façon dont sont représentés les objets des deux types en mémoire.

```
struct info1 { int value; struct s1 s1; };
struct info2 { int value; struct s1 *p1; };
```

6. L'une des deux déclarations suivantes sera rejetée par le compilateur C :

```
struct tree1 { int value; struct tree1 left, right; };
struct tree2 { int value; struct tree2 *left, *right; };
```

   Laquelle ? Pourquoi ?

7. En Java, la seule déclaration autorisée d'un type équivalent sera celle qui ne mentionne pas « * », et pour cause : * n'existe pas en Java. La voici :

```
public class Tree {
  int value;
  Tree left, right;
  // methodes omises
}
```

   De laquelle des définitions de la question précédente se rapproche-t-elle, en termes de représentation en mémoire ?

8. Pourquoi les deux formes d'« inclusions » de structures (voir la question 5) sont-elles autorisées en C ? (Seule la premiàre est effectivement appelée inclusion, au passage.)

9. Pourquoi Java et Caml n'autorisent-ils qu'une seule des formes d'« inclusion » ?

2 Portée lexicale, portée dynamique, variantes

Exercise 3 : Variables statiques

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+1, j);
  return (maxl > maxr)?maxl:maxr;
}
```

2. 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 ?

Exercise 4 : Portée statique

Que retourne chacune des expressions Caml suivantes ?

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 5 : Portée dynamique

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

1. 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.

2. 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 = 7 in
let f = fun x -> x+i in
let i = 0 in
   f 3
```

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

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

```
(setq i 7)
(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;;
```

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

4. 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 ?

5. Et si l'on a aussi des *threads* dans le langage ?

Exercise 6 : Portée dynamique ... et fonctions !

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)` ?

Alpha renommage

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 les questions suivantes.

Exercise 7 : Alpha renommage et portée dynamique

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.

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 ?

1. 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.

2. 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 ?

3 Appel par nom, par valeur, par référence

Exercise 8 : Étude de différents langages

1. Que fait la fonction C suivante ?

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

2. Comment peut-on la corriger ? En C++ ? En Pascal ? En Java ?
3. 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 ?

Exercise 9 : 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)
```

1. Avant de commencer, pourquoi ai-je mis des parenthèses autour des trois instances de `i` dans la définition de `ABS` ?
2. 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*.
3. 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 ?

4. 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` ?

Exercise 10 : Algol-60

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`).

1. 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;
```

2. D'après vous, que calcule `Sum (i, 1, 100, V[i])` ?
3. 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])` ?

Exercise 11 : Efficacité des stratégies d'évaluation

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

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 12 : Implémentation de l'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.

1. 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 !
2. Écrire un extrait de programme C qui fabrique une liste paresseuse infinie ne contenant que des 1.
3. Écrire une fonction C qui prend une liste paresseuse d'entiers, et retourne la liste des mêmes entiers auxquelles on a ajouté 1.
4. 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...).
5. 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 ?

Exercise 13 : Paresse en Haskell

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.

1. Que vaut la liste `hamming` ?
2. 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`)`)
3. Si je demande la valeur de `hamming !!` n, quelle est la partie de la liste des nombres de Hamming qui aura été effectivement calculée ?
4. Écrire la fonction `mergeUnique`.