École Normale Supérieure

Langages de programmation et compilation

examen 2021–2022

Jean-Christophe Filliâtre

21 janvier 2022 — 8h30–11h30

Les notes de cours manuscrites ou reprographiées sont les seuls documents autorisés.
L'épreuve dure 3 heures.
Les figures sont regroupées en fin de sujet, page 7. Suggestion : détacher la dernière feuille.

Dans tout ce sujet, on considère un petit langage de programmation dont la syntaxe abstraite est donnée figure 1. Dans ce langage, les seules valeurs sont des listes, construites à partir de la liste vide `[]` et de l'opération `::`. Les fonctions sont notées f. Elles sont récursives et uniquement globales. Chaque fonction a un seul argument.

Ce langage est moins pauvre qu'il n'y paraît. On peut en effet encoder des données en utilisant uniquement des listes. Par exemple, on peut représenter un entier n comme une liste contenant n éléments tous égaux à `[]`. Ainsi, l'entier 2 est représenté par la liste `[] :: [] :: []`. On peut également représenter une paire (x, y) par la liste à deux éléments x `::` y `::` `[]`, et s'affranchir ainsi du fait que les fonctions n'ont qu'un seul argument. Avec toutes ces idées, on peut proposer les définitions suivantes

```
rec fst x = match x with | [] -> [] | y :: _ -> y
rec snd x = match x with | [] -> [] | _ :: y -> fst y
rec add p = match fst p with
  | []     -> snd p
  | _ :: x -> [] :: add (x :: snd p :: [])
```

où la fonction `add` calcule donc la somme de deux entiers.

L'instruction `print` affiche la valeur d'une expression, suivie d'un retour chariot. Les éléments d'une liste sont affichés entre crochets, chaque élément étant suivi d'un point-virgule et d'un espace. Ainsi, avec les définitions ci-dessus, l'évaluation de l'instruction

```
print
  let three = [] :: [] :: [] :: [] in
  add (three :: three :: [])
```

va provoquer l'affichage suivant

```
[[]; []; []; []; []; []; ]
```

c'est-à-dire une liste qui représente l'entier 6.

Question 1 Donner la définition d'une fonction `fib` telle que, si `n` est une liste représentant l'entier n, alors `fib n` est une liste représentant l'entier F_n avec

$$\begin{cases} F_0 &= 0 \\ F_1 &= 1 \\ F_{n+2} &= F_n + F_{n+1} \quad \text{pour } n \geq 0. \end{cases}$$

Correction : On se ressert de la fonction `add` donnée dans l'énoncé.

```
rec fib n = match n with
  | []     -> []
  | _ :: p -> match p with
              | []      -> [] :: []
              | _ :: pp -> add (fib p :: fib pp :: [])
```

Sémantique opérationnelle. On souhaite munir notre langage d'une sémantique opérationnelle à petits pas, sous la forme d'une relation $e \to e'$ définie entre deux expressions par

$$\frac{e_1 \xrightarrow{\epsilon} e_2}{E(e_1) \to E(e_2)}$$

où E désigne un contexte d'évaluation et $\xrightarrow{\epsilon}$ désigne une réduction en tête.

Question 2 Proposer une définition pour les contextes E et pour la réduction $\xrightarrow{\epsilon}$, qui traduisent un passage par valeur et une évaluation de la gauche vers la droite. Ici, les valeurs sont définies par

$$v ::= \texttt{[]} \mid v \texttt{ :: } v$$

et on suppose qu'on a défini une opération de substitution $e[x \leftarrow v]$.

Indications : Il doit y avoir six cas dans la définition de E et quatre dans la définition de $\xrightarrow{\epsilon}$.

Correction : Les contextes :

$$\begin{array}{rcl} E & ::= & \square \\ & \mid & \texttt{let } x = E \texttt{ in } e \\ & \mid & f\ E \\ & \mid & E \texttt{ :: } e \\ & \mid & v \texttt{ :: } E \\ & \mid & \texttt{match } E \texttt{ with } \mid \texttt{[]} \to e \mid x \texttt{ :: } x \to e \end{array}$$

Les réductions de tête :

$$\begin{array}{rcl} \texttt{let } x = v \texttt{ in } e & \xrightarrow{\epsilon} & e[x \leftarrow v] \\ \texttt{match [] with} \mid \texttt{[]} \to e_1 \mid x_1 \texttt{ :: } x_2 \to e_2 & \xrightarrow{\epsilon} & e_1 \\ \texttt{match } v_1 \texttt{ :: } v_2 \texttt{ with} \mid \texttt{[]} \to e_1 \mid x_1 \texttt{ :: } x_2 \to e_2 & \xrightarrow{\epsilon} & e_2[x_1 \leftarrow v_1][x_2 \leftarrow v_2] \\ f\ v & \xrightarrow{\epsilon} & e[x \leftarrow v] \quad \text{avec } \texttt{rec } f\ x \texttt{ = } e \end{array}$$

Analyse syntaxique. On souhaite réaliser l'analyse syntaxique de notre petit langage avec l'outil Menhir. La figure 2 contient un fichier d'entrée pour Menhir contenant la grammaire de notre langage. Les actions sémantiques ne nous intéressent pas ici et sont omises (`{...}`).

Question 3 Lorsque l'outil Menhir est lancé sur le fichier donné en figure 2, il déclare quatre conflits de type lecture/réduction (`shift/reduce`). Les identifier, les expliquer, faire le choix de favoriser lecture ou réduction et modifier en conséquence le fichier Menhir.

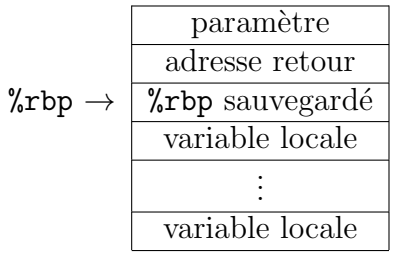
Correction : Les quatre conflits sont liés à la construction e `::` e et correspondent aux situations suivantes :

```
e :: e :: e
match ... -> e :: e
let x = e in e :: e
f e :: e
```

Une façon naturelle de résoudre ces conflits consiste à faire les mêmes choix que le langage OCaml, à savoir :
— *cons* est associatif à droite ;
— *cons* est plus prioritaire que `match` et `let` ;
— *cons* est moins prioritaire que l'application.
Ceci se traduit en Menhir avec les déclarations suivantes :

```
%nonassoc ARROW IN
%right CONS
%nonassoc IDENT
```

Compilation vers x86-64. On se propose maintenant de compiler notre petit langage vers l'assembleur x86-64. (Un aide-mémoire est donné en annexe.) On adopte le schéma de compilation suivant. Une valeur est soit l'entier 0 (sur 64 bits), représentant la liste vide, soit un pointeur vers un bloc de 16 octets alloué sur le tas, contenant deux valeurs v_1 et v_2 et représentant la liste v_1 `::` v_2. Les valeurs ont donc toutes la même taille (64 bits). Pour toute définition `rec` f x `=` e contenue dans le programme, on introduit une fonction assembleur f, recevant son paramètre x sur la pile. Le tableau d'activation de la fonction a la forme ci-contre, les adresses croissant vers le haut. Dans ce tableau, on trouve les variables locales à l'expression e, qu'elles soient introduites par `let` ou par `match`. Toutes les variables sont accessibles à partir de `%rbp`, soit avec $+16$ pour le paramètre, soit avec -8, -16, etc., pour les variables locales. Le résultat de la fonction est renvoyé dans le registre `%rax`.



Question 4 Donner un code assembleur pour la fonction `fst` suivante :

```
rec fst x = match x with | [] -> [] | y :: _ -> y
```

Correction :

```
fst:
    pushq %rbp
    movq %rsp, %rbp
    movq 16(%rbp), %rax
    testq %rax, %rax
```

```
    jz 1f
    movq (%rax), %rax
1:  popq %rbp
    ret
```

Note : On a respecté ici le tableau d'activation proposé dans l'énoncé, en sauvegardant `%rbp`, mais c'est inutilement compliqué pour une fonction aussi simple que `fst`. Le code

```
fst:
    movq 8(%rsp), %rax
    testq %rax, %rax
    cmovnzq (%rax), %rax
    ret
```

convient tout aussi bien.

Allocation des variables. Pour déterminer la position de chaque variable sur la pile, on se donne une fonction $A(a, n, e)$ qui renvoie une expression identique à e, où chaque variable est remplacée par sa position sur la pile. Le paramètre a est une fonction donnant les positions des variables déjà déterminées. Le paramètre n est la prochaine position libre sur la pile. Ainsi, pour une définition de fonction `rec` $f\ x$ = e, on va réaliser l'allocation avec $A(\{x \mapsto 16\}, -8, e)$, c'est-à-dire que l'on indique que le paramètre x est à la position $+16$ et que la prochaine position libre est -8.

Question 5 Proposer une définition de la fonction A pour chacune des six constructions du langage. Sans chercher à faire quelque chose de trop sophistiqué, on essayera de proposer un schéma qui permet d'allouer plusieurs variables sur un même emplacement.

Correction :

```
A(a, n, x) =
  a(x)
A(a, n, f e1) =
  f A(a, n, e1)
A(a, n, let x = e1 in e2)
  let n = A(a, n, e1) in A(a+{x->n}, n-8, e2)
A([])
  []
A(e1 :: e2) =
  A(a, n, e1) :: A(a, n, e2)
A(match e1 with [] -> e2 | x :: y -> e3) =
  match A(a, n, e1) with
  | []       -> A(a, n, e2)
  | n :: n-8 -> A(a+{x->n}+{y->n-8}, n-16, e3)
```

Question 6 Donner une expression e contenant plusieurs variables locales, dont
- au moins deux peuvent être allouées au même emplacement ;
- au moins deux ne peuvent pas être allouées au même emplacement.

Correction : Il y a d'innombrables solutions. Par exemple, avec l'expression

```
match ... with | []     -> let z = [] in z
               | x :: y -> x :: y :: []
```

les deux variables `x` et `y` ne peuvent pas être allouées au même emplacement, mais en revanche les deux variables `z` et `x` peuvent être allouées au même emplacement.

Compilation. La compilation d'une expression e est un code assembleur noté $C(e)$ dont l'exécution a pour effet de stocker la valeur de l'expression e dans le registre `%rax`.

Question 7 Donner la définition de $C(e)$ pour chacune des six constructions du langage.

Correction :

```
C(x) =
  movq n(%rbp), %rax  # où n est la position de x sur la pile
C(f e1) =
  C(e1)
  pushq %rax
  call f
  addq $8, %rsp
C(let x = e1 in e2)
  C(e1)
  movq %rax, n(%rbp) # où n est la position de x sur la pile
  C(e2)
C([])
  xorq %rax, %rax
C(e1 :: e2) =
  C(e1)
  pushq %rax
  C(e2)
  pushq %rax
  movq $16, %rdi
  call malloc
  popq rdi
  movq %rdi, 8(%rax)
  popq rdi
  movq %rdi, (%rax)
C(match e1 with [] -> e2 | x :: y -> e3) =
  C(e1)
  testq %rax, %rax     # est-ce nil ?
  jnz 1f
  C(e2)
  jmp 2f
1: movq (%rax), %rcx
  movq %rcx, nx(%rbp)  # où nx est la position de x sur la pile
```

```
    movq 8(%rax) %rcx
    movq %rcx, ny(%rbp)  # où ny est la position de x sur la pile
    C(e3)
 2:
```

Question 8 Proposer un code assembleur pour une fonction `print` qui imprime une liste au format décrit au début de l'énoncé. On suppose que l'argument de `print` est passé dans le registre `%rdi`. On se servira de la fonction `putchar` pour imprimer un caractère (voir l'annexe). On cherchera à optimiser tous les appels terminaux. *Attention : un seul retour chariot doit être imprimé, tout à la fin.*

Correction : La difficulté ici, outre le fait de devoir imprimer un seul retour chariot, vient surtout du caractère *récursif* de `print`, qui oblige à bien sauvegarder la liste en cours d'impression. Dans une moindre mesure, attention à `putchar` qui peut écraser `%rdi`.

```
print: # imprime %rdi puis un retour chariot
      call _printl
      movq $10, %rdi
      jmp  putchar  # <- appel terminal
_printl: # imprime les crochets autour de l'appel à printe
      pushq %rdi
      movq $'[', %rdi
      call putchar
      popq %rdi
      call _printe
      movq $']', %rdi
      jmp  putchar  # <- appel terminal
_printe: # imprime les éléments de %rdi
      testq %rdi, %rdi
      jz 1f
      pushq %rdi
      movq (%rdi), %rdi
      call _printl
      movq $';', %rdi
      call putchar
      movq $' ', %rdi
      call putchar
      popq %rdi
      movq 8(%rdi), %rdi
      jmp _printe  # <- appel terminal
1:    ret
```

Optimisation. Dans la fonction `add` donnée en exemple au tout début de l'énoncé, à savoir

```
rec add p = match fst p with
  | []     -> snd p
  | _ :: x -> [] :: add (x :: snd p :: [])
```

l'appel récursif à `add` n'est pas un appel terminal, car il faut ensuite ajouter un élément au début de la liste renvoyée par cet appel récursif. Il existe cependant un moyen de le transformer en un appel terminal. Pour cela, on alloue le bloc correspondant à `[] :: ...` *avant* de faire l'appel récursif et on le passe à la fonction pour qu'elle y stocke son résultat, avec une affectation, plutôt que de le renvoyer. On obtient alors une fonction de la forme suivante :

```
rec add_dps d p = match fst p with
  | []     -> d.next <- snd p
  | _ :: x -> let d' = [] :: [] in
              let _ = d.next <- d' in
              let d = d' in
              add_dps d (x :: snd p :: [])
```

L'argument `d` est la *destination* du calcul, c'est-à-dire l'emplacement où il faut mettre le résultat. L'opération `d.next <- ...` modifie cet emplacement pour y stocker une valeur (dans le second champ du bloc mémoire, ici intitulé `next` puisqu'il s'agit de listes). Cette version de la fonction `add` ne renvoie plus rien, mais elle stocke son résultat dans `d`. On constate que son appel récursif, sur la dernière ligne, est maintenant un appel terminal. En particulier, il peut être optimisé pour conduire à une exécution qui se fera en espace de pile constant.

Pour retrouver une fonction `add` qui calcule la même chose que la fonction d'origine, il suffit alors de la définir ainsi :

```
rec add p = match fst p with
  | []     -> snd p
  | _ :: x -> let d = [] :: [] in
              let _ = add_dps d (x :: snd p :: []) in
              d
```

On note que cette fonction `add` n'est pas récursive : elle se charge uniquement du premier appel à la fonction récursive `add_dps`, lorsque la liste `fst p` n'est pas vide.

Ce style de programmation s'appelle *destination passing style*. On se propose de réaliser cette optimisation sur notre petit langage. Pour cela, on étend la syntaxe des expressions avec de nouvelles constructions :

$$\begin{array}{rcl} e & ::= & \ldots \\ & | & d.\texttt{next} \leftarrow e \\ & | & f^{dps}\ x\ e \end{array}$$

L'expression $d.\texttt{next} \leftarrow e$ affecte la valeur de e au second champ de la liste désignée par la variable d. Cette construction suppose que la variable d désigne une liste non vide. Cette construction ne renvoie rien (mais pour la faire rentrer dans le formalisme précédent, on pourrait supposer qu'elle renvoie une valeur non significative, comme par exemple la liste vide).

Par ailleurs, une définition de fonction `rec` $f\ x$ `=` e dans le langage d'origine devient maintenant une paire de fonctions mutuellement récursives

$$\begin{array}{rcl} \texttt{rec}\ f\ x & = & T(\texttt{false}, e) \\ \texttt{and}\ f^{dps}\ d\ x & = & T(\texttt{true}, e) \end{array}$$

La fonction f se comporte comme la fonction originale. La fonction f^{dps} ne renvoie rien et se comporte comme $d.\texttt{next} \leftarrow f\ e$. Le premier paramètre formel de f^{dps} s'appelle toujours d. La fonction T réalise l'optimisation. Elle prend en argument un booléen indiquant si on se trouve dans la fonction f ou dans la fonction f^{dps} et une expression à traduire.

Question 9 Donner un exemple de fonction justifiant le caractère mutuellement récursif des fonctions f et f^{dps}.

Correction : Il suffit de prendre une fonction f où un appel récursif n'est pas en position d'être optimisé. C'est le cas par exemple de cette fonction identité :

```
rec id l =  match l with
  | []     -> []
  | x :: y -> id x :: id y
```

Dans la traduction, le premier appel récursif `id x` reste un appel à `id` :

```
rec id_dps d l = match l with
  | []     -> d.next <- []
  | x :: y -> let d' = id x :: [] in
              let _ = d.next <- d' in
              let d = d' in
```

```
                id_dps d y
and id l = match l with
  | []     -> []
  | x :: y -> let d = id x :: [] in let _ = id_dps d y in d
```

On a bien là deux fonctions mutuellement récursives.

Question 10 Proposer une définition de la fonction T pour chacune des six constructions originales du langage.

Correction : Idée : on n'utilise T uniquement sur des positions terminales. Ailleurs, on laisse les expressions inchangées.

```
T(true, f e) =
   f_dps d e  // appel terminal !
T(false, f e) =
   f e
T(b, let x = e1 in e2) =
   let x = e1 in T(b, e2)
T (b, match e1 with [] -> e2 | x :: y -> e3) =
   match e1 with [] -> T(b, e2) | x :: y -> T(b, e3)
T(true, e1 :: f e2) =
   let d' = e1 :: [] in
   let _ = d.next <- d' in
   let d = d' in
   f_dps d e2
T(false, e1 :: f e2) =
   let d = e1 :: [] in
   let _ = f_dps d e2 in
   d
```

Et pour les autres cas, c'est-à-dire x, `[]` et e_1 `::` e_2 :

```
T(true, e) =
   d.next <- e
T(false, e) =
   e
```

Auto-interprète. Pour conclure en beauté, on se propose d'explorer dans les questions suivantes l'idée d'écrire un interprète de notre langage *dans le langage lui-même*, c'est-à-dire écrire une fonction

```
rec eval p = ...
```

qui reçoit en argument le codage sous forme de listes d'un programme et renvoie la liste des (codes des) caractères imprimés par l'évaluation de ce programme. On considère ici uniquement le langage original, tel que défini dans la figure 1, c'est-à-dire sans les extensions introduites dans les deux questions précédentes.

Pour écrire cet auto-interprète, on reprend notamment deux idées introduites au début du sujet :

— un entier n est représenté par une liste contenant n éléments tous égaux à `[]` ;

— une paire (x, y) est représentée par la liste à deux éléments x `::` y `::` `[]`.

Question 11 Donner la définition d'une fonction `equal` qui reçoit en argument (le codage d')une paire (x, y) et renvoie `[]` si les listes x et y sont identiques et `[] :: []` sinon. La comparaison des listes x et y doit être faite en profondeur.

Correction : On procède récursivement, en examinant les deux listes :

```
rec equal p = match fst p with
  | []       -> match snd p with
                | []   -> []
                | _::_ -> []::[]
  | x1 :: y1 -> match snd p with
                | []       -> []::[]
                | x2 :: y2 -> match equal (x1 :: x2 :: []) with
                              | []   -> equal (y1 :: y2 :: [])
                              | _::_ -> []::[]
```

Liste d'association. Une liste d'association est une liste de paires (k, v) où k est une clé et v une valeur. Ainsi, la liste

```
([] :: ([]::[]) :: []) :: (([]::[]::[]) :: ([]::[]::[]::[]) :: []) :: []
```

associe à la clé `[]` la valeur `[]::[]` et à la clé `[]::[]::[]` la valeur `[]::[]::[]::[]`.

Question 12 Donner la définition d'une fonction `assoc` qui reçoit en argument (le codage d')une paire (k, a), où a est une liste d'association, et renvoie la première valeur v associée à k dans a, le cas échéant, et `[]` si la clé k n'apparaît pas dans a.

Correction :

```
rec assoc p = match snd p with
  | []      -> []
  | b :: al -> match eq (fst p :: fst b :: []) with
               | []   -> snd b
               | _::_ -> assoc (fst p :: al :: [])
```

Question 13 Proposer un codage pour la syntaxe abstraite du langage, telle qu'elle est définie dans la figure 1. On distinguera un codage pour les expressions (e), un autre pour les déclarations (d) et un troisième pour les programmes (p).

Correction : On peut représenter les noms de fonctions (f) et de variables (x) par des entiers.

On peut représenter une expression par une liste

$$c :: a_1 :: a_2 :: \ldots$$

où $0 \leq c \leq 5$ est un entier qui identifie la construction (par exemple, $c = 0$ pour une variable, $c = 1$ pour une application, etc.) et où $a_1, a_2, \ldots$ sont les arguments (le nom x de la variable dans le cas $c = 0$, le nom f et l'expression e dans le cas $c = 1$, etc.).
De même, on peut représenter une déclaration par une liste

$$c :: a_1 :: a_2 :: \ldots$$

où $c = 0$ identifie une déclaration `rec` et $c = 1$ une déclaration `print`.
Enfin, un programme p est une liste de déclarations.

Question 14 Donner les grandes lignes de la fonction `eval`, c'est-à-dire sa décomposition en sous-fonctions et la spécification de chacune de ces sous-fonctions. On ne demande pas d'écrire le code de ces fonctions.

Correction : Dans ce qui suit,
— `funs` est une liste d'associations $(f, (x, e))$ correspondant à une définition `rec` f x `=` e ;
— `vars` est une liste d'associations (x, v) où v est une valeur ;
— `out` est une suite de caractères, un caractère étant représenté par un entier.
On propose alors les fonctions suivantes :

```
evale ((funs,vars), e) -> v
  // évalue une expression e

affiche v -> out
  // affiche une valeur

evald (funs, d) -> (funs, out)
  // évalue une déclaration d
  //   si c'est rec f x = e, étend funs avec (f,(x,e)) et out est vide
  //   si c'est print e, funs est inchangé et out est l'affichage de e

concat (out1, out2) -> out1^out2
  // concatène deux sorties

evalp (funs, dl) -> out
  // évalue une liste de déclarations
  // essentiellement un fold de evald, en concaténant les sorties

eval p -> out
  // évalue un programme p, c'est-à-dire evalp ([] :: p :: [])
```

$$
\begin{array}{rcll}
e & ::= & x & \textit{variable} \\
& | & f\ e & \textit{application} \\
& | & \texttt{let}\ x = e\ \texttt{in}\ e & \textit{variable locale} \\
& | & \texttt{[]} & \textit{liste vide} \\
& | & e\ \texttt{::}\ e & \textit{cons} \\
& | & \texttt{match}\ e\ \texttt{with} \mid \texttt{[]} \to e \mid x\ \texttt{::}\ x \to e & \textit{filtrage} \\
\\
d & ::= & \texttt{rec}\ f\ x\ \texttt{=}\ e & \textit{fonction récursive} \\
& | & \texttt{print}\ e & \textit{affichage} \\
\\
p & ::= & d\ \ldots\ d & \textit{programme}
\end{array}
$$

FIGURE 1 – Syntaxe abstraite.

```
%token <string> IDENT
%token LET IN REC PRINT MATCH WITH
%token EQ BAR NIL CONS ARROW LEFTPAR RIGHTPAR EOF
%start program
%type <...> program
%%
program:
| list(decl) EOF                                          {...}
decl:
| REC IDENT IDENT EQ expr                                 {...}
| PRINT expr                                              {...}
expr:
| IDENT                                                   {...}
| IDENT expr                                              {...}
| LET IDENT EQ expr IN expr                               {...}
| NIL                                                     {...}
| expr CONS expr                                          {...}
| MATCH expr WITH
  BAR? NIL ARROW expr BAR IDENT CONS IDENT ARROW expr     {...}
| LEFTPAR expr RIGHTPAR                                   {...}
```

FIGURE 2 – Fichier Menhir.

Annexe : aide-mémoire x86-64

On donne ici un fragment du jeu d'instructions x86-64. Vous êtes libre d'utiliser tout autre élément de l'assembleur x86-64. Dans ce qui suit, r_i désigne un registre, n une constante entière et L une étiquette.

`mov` r_2`,` r_1	copie le registre r_2 dans le registre r_1
`mov` `$`n`,` r_1	charge la constante n dans le registre r_1
`mov` L`,` r_1	charge la valeur à l'adresse L dans le registre r_1
`mov` `$`L`,` r_1	charge l'adresse de l'étiquette L dans le registre r_1
`add` r_2`,` r_1	calcule la somme de r_1 et r_2 dans r_1 (on a de même `sub` et `imul`)
`mov` n`(`r_2`),` r_1	charge dans r_1 la valeur contenue en mémoire à l'adresse $r_2 + n$
`mov` r_1`,` n`(`r_2`)`	écrit en mémoire à l'adresse $r_2 + n$ la valeur contenue dans r_1
`push` r_1	empile la valeur contenue dans r_1
`pop` r_1	dépile une valeur dans le registre r_1
`cmp` r_2`,` r_1	positionne les drapeaux en fonction de la valeur de $r_1 - r_2$
`test` r_2`,` r_1	positionne les drapeaux en fonction de la valeur de $r_1 \& r_2$
`je` L	saute à l'adresse désignée par l'étiquette L en cas d'égalité (on a de même `jne`, `jg`, `jge`, `jl` et `jle`)
`jmp` L	saute à l'adresse désignée par l'étiquette L
`call` L	saute à l'adresse désignée par l'étiquette L, après avoir empilé l'adresse de retour
`ret`	dépile une adresse et y effectue un saut

On alloue de la mémoire sur le tas avec un appel à `malloc`, qui attend un nombre d'octets dans `%rdi` et renvoie l'adresse du bloc alloué dans `%rax`.

On imprime un caractère avec un appel à `putchar`, qui attend le code du caractère dans `%rdi`. Dans le code assembleur, on peut écrire (le code d')un caractère c symboliquement, avec la syntaxe `$'`c`'`. Ainsi, `movq $'a', %rdi` met le code du caractère `a` dans le registre `%rdi`.