École Polytechnique

INF564 : Compilation

examen 2018 (X2015)

Jean-Christophe Filliâtre

19 mars 2018

Les notes de cours manuscrites ou reprographiées sont les seuls documents autorisés.
L'épreuve dure 3 heures.

Dans tout ce sujet, on considère un petit langage de programmation avec des entiers et des paires. Un programme est composé d'un ensemble de fonctions mutuellement récursives et d'une expression principale. Chaque fonction a un unique argument, toujours appelé `x`. Voici un programme dans ce langage, calculant la factorielle de 10 :

```
def mult(x) =
  ifzero fst(x) then 0 else add((snd(x), mult((add((fst(x),-1)), snd(x)))))
def fact(x) =
  ifzero x then 1 else mult((x, fact(add((x, -1)))))
fact(10)
```

Ce langage est muni d'une sémantique d'appel par valeur. Il y a trois fonctions prédéfinies :
- `add` : attend une paire d'entiers en argument et renvoie leur somme ;
- `fst` : attend une paire en argument et renvoie sa première composante ;
- `snd` : attend une paire en argument et renvoie sa seconde composante.

La syntaxe abstraite de ce langage est donnée figure 1.

Les différentes parties du sujet sont indépendantes. Néanmoins, elles nécessitent d'avoir bien compris la sémantique de ce langage décrite dans la partie 1.

1 Sémantique

On munit ce langage d'une sémantique opérationnelle à petits pas de la forme

$$e \to e'$$

où e et e' sont deux expressions. Les règles de la sémantique opérationnelle sont données figure 2. La notation $e[\mathtt{x} \leftarrow v]$ désigne l'expression obtenue en remplaçant dans e toute occurrence de `x` par v. On prendra le temps de bien comprendre les règles de la sémantique. L'*évaluation* d'une expression e est une séquence de réductions partant de e et conduisant à une valeur, c'est-à-dire

$$e \to e_1 \to e_2 \cdots \to v.$$

L'évaluation d'un programme est l'évaluation de son expression principale.

e	::=	**expression**
	$\mid\ n$	constante entière $n \in \mathbb{Z}$
	$\mid$ `x`	variable
	$\mid$ $(e\,,\,e)$	construction d'une paire
	$\mid$ $f(e)$	appel de fonction
	$\mid$ `ifzero` e `then` e `else` e	conditionnelle
d	::=	**définition de fonction**
	$\mid$ `def` f`(x) =` e	
p	::=	**programme**
	$\mid$ $d \ldots d\ e$	

FIGURE 1 – Syntaxe abstraite.

v	::=	**valeur**
	$\mid\ n$	constante entière
	$\mid\ (v\,,\,v)$	paire

$$\frac{e_1 \to e_1'}{(e_1\,,\,e_2) \to (e_1'\,,\,e_2)} \qquad \frac{e_2 \to e_2'}{(v_1\,,\,e_2) \to (v_1\,,\,e_2')}$$

$$\frac{e_1 \to e_1'}{f(e_1) \to f(e_1')} \qquad \frac{\texttt{def}\ f\texttt{(x) =}\ e}{f(v) \to e[\texttt{x} \leftarrow v]}$$

$$\frac{n = n_1 + n_2}{\texttt{add}((n_1\,,\,n_2)) \to n} \qquad \frac{}{\texttt{fst}((v_1\,,\,v_2)) \to v_1} \qquad \frac{}{\texttt{snd}((v_1\,,\,v_2)) \to v_2}$$

$$\frac{e_1 \to e_1'}{\texttt{ifzero}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \to \texttt{ifzero}\ e_1'\ \texttt{then}\ e_2\ \texttt{else}\ e_3}$$

$$\frac{}{\texttt{ifzero}\ 0\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \to e_2} \qquad \frac{v \neq 0}{\texttt{ifzero}\ v\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \to e_3}$$

FIGURE 2 – Sémantique opérationnelle à petits pas.

Question 1 Donner l'évaluation du programme

```
def f(x) = add((x, x))
f(add((add((3,5)), add((5,8))))
```

Correction :

```
   f(add((add((3,5)), add((5,8))))
-> f(add((8         , add((5,8))))
-> f(add((8         , 13       )))
-> f(21)
-> add((21, 21))
-> 42
```

Question 2 Donner un programme qui n'a pas d'évaluation.

Correction : Le plus simple est un programme réduit à l'expression `x`.

Listes. On peut facilement représenter une liste contenant n valeurs $x_1, \ldots, x_n$ avec des paires, sous la forme

$$(x_1, (x_2, \ldots, (x_n, 0) \ldots))$$

c'est-à-dire exactement une paire pour chaque élément de la liste et l'entier 0 pour représenter la fin de la liste. On peut alors définir des fonctions récursives sur de telles listes. Voici par exemple une fonction `len` qui renvoie la longueur d'une liste et une fonction `rev` qui renverse une liste :

```
def len(x) =
  ifzero x then 0 else add((1, len(snd(x))))
def rev_aux(x) =
  ifzero fst(x) then snd(x) else rev_aux((snd(fst(x)), (fst(fst(x)), snd(x))))
def rev(x) =
  rev_aux((x, 0))
```

Question 3 Définir une fonction `fib` telle que, pour tout entier $n \geq 1$, l'évaluation de `fib`(n) renvoie la liste des $n+1$ premiers entiers de la suite de Fibonacci, dans l'ordre croissant. On rappelle que la suite de Fibonacci est définie par

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

La complexité, définie comme le nombre total de petits pas, doit être $O(n)$.

Correction : On généralise avec une fonction `fib_aux` qui prend en argument la paire (k, l) où k est le nombre de nombres de Fibonacci restant à calculer et l la liste des entiers $F_{n-k}, F_{n-k-1}, \ldots, F_1, F_0$.

```
def fib_aux(x) =
  ifzero fst(x) then rev(snd(x))
  else fib_aux((add((fst(x), -1)),
                (add((fst(snd(x)), fst(snd(snd(x))))), snd(x))))
```

Il suffit alors de l'appeler avec $n-1$ et la liste à deux éléments F_1, F_0.

```
def fib(x) =
  fib_aux((add((x, -1)), (1, (0, 0))))
```

---

2 Interprète

On souhaite réaliser un interprète de notre langage, c'est-à-dire un programme qui reçoit en argument la syntaxe abstraite d'un programme de notre langage et qui

- affiche la valeur à laquelle aboutit l'évaluation de ce programme, le cas échéant ;
- signale une erreur si la séquence de réductions aboutit à une expression qui n'est pas une valeur et qui ne se réduit pas ;
- ne termine pas sinon.

Question 4 Donner les différents types, structures de données et fonctions/méthodes composant cet interprète, en OCaml ou en Java (au choix). On ne demande pas en revanche d'écrire le *code* de l'interprète.

Correction : On choisit par exemple OCaml. La syntaxe abstraite des expressions est immédiate :

```
type expr =
  | Ecst  of int
  | Evar  (* x *)
  | Epair of expr * expr
  | Eapp  of string * expr
  | Eifz  of expr * expr * expr
type defn = string * expr
type program = defn list * expr
```

On se donne un type pour les valeurs :

```
type value =
  | Vint of int
  | Vpair of value * value
```

Note : on aurait pu utiliser le type `expr` pour cela, vu qu'il contient déjà toutes les valeurs possibles. On se donne une table de hachage globales `defns` pour les fonctions. On y met les fonctions prédéfinies aussi bien que les fonctions du programme. On lui donne le type suivant :

```
  defns: (string, value -> value) Hashtbl.t
```

et on la remplit avec les fonctions prédéfinies :

$$
\begin{array}{rcl}
\mathit{compile}(n) & = & \texttt{movq \$}n\texttt{, \%rax} \\
\mathit{compile}(\texttt{x}) & = & \texttt{movq \%rdi, \%rax} \\
\mathit{compile}(\texttt{(}e_1\texttt{, }e_2\texttt{)}) & = & \ldots \\
\mathit{compile}(f\texttt{(}e_1\texttt{)}) & = & \ldots \\
\mathit{compile}(\texttt{ifzero } e_1 \texttt{ then } e_2 \texttt{ else } e_3) & = & \ldots \\
\\
\mathit{compile}(\texttt{def } f\texttt{(x) = } e) & = & f\texttt{:} \\
 & & \mathit{compile}(e) \\
 & & \texttt{ret}
\end{array}
$$

FIGURE 3 – Compilation vers x86-64.

```
let () =
  Hashtbl.add defns "add"
    (function Vpair (Vint n1, Vint n2) -> Vint (n1 + n2)
            | _ -> error "bad argument for add");
  ...
```

On définit ensuite trois fonctions pour l'interprète proprement dit :

```
val of_value: value -> expr
val subst: expr -> expr -> expr
val expr: expr -> value
```

Pour évaluer un programme, on ajoute ses fonctions à la table de hachage, ainsi :

```
let defn e v = expr (subst e (of_value v)) in
List.iter (fun (f, e) -> Hashtbl.add defns f (defn e)) dl;
```

puis on évalue son expression principale avec `expr`.

---

3 Compilation vers x86-64

On se propose maintenant de compiler notre langage vers l'assembleur x86-64. On adopte le schéma de compilation suivant :

- une valeur est soit un entier signé 64 bits, soit un pointeur vers un bloc de 16 octets sur le tas contenant les deux composantes d'une paire (juxtaposées) ;
- la valeur d'une expression est calculée dans `%rax` ;
- le valeur de `x`, le cas échéant, est contenue dans `%rdi`.

La figure 3 contient une partie du compilateur. Une expression e est compilée par un appel à $\mathit{compile}(e)$. Un aide-mémoire x86-64 est donné en annexe.

Question 5 Donner le code du compilateur pour

- la construction de la paire $\texttt{(}e_1\texttt{, }e_2\texttt{)}$;
- la conditionnelle $\texttt{ifzero } e_1 \texttt{ then } e_2 \texttt{ else } e_3$;
- l'application $f\texttt{(}e_1\texttt{)}$.

Correction :

— la construction (e_1, e_2) : il faut penser à sauvegarder `%rdi` avant d'appeler `malloc`

```
    compile(e1)          # la première composante est évaluée
    pushq %rax           # et placée sur la pile
    compile(e2)          # la seconde composante est évaluée
    pushq %rax           # et placée sur la pile
    pushq %rdi           # on sauvegarde %rdi
    movq  $16, %rdi      # on alloue 16 octets sur le tas
    call  malloc
    popq  %rdi           # on restaure %rdi
    popq  %rcx           # on récupère la seconde composante
    movq  %rcx, 8(%rax) # et on la stocke dans la paire
    popq  %rcx           # on récupère la première composante
    movq  %rcx, (%rax)  # et on la stocke dans la paire
```

— la conditionnelle `ifzero` e_1 `then` e_2 `else` e_3 : pas de difficulté ici

```
    compile(e1)
    testq %rax, %rax
    jz    L2
    compile(e3)
    jmp   L1
 L2:compile(e2)
 L1:
```

où L_1 et L_2 sont deux étiquettes fraîches.

— l'application f`(`e_1`)` : là aussi il faut préserver `%rdi` qui est *caller-save*

```
    pushq %rdi        # on sauvegarde %rdi
    compile(e1)       # on évalue l'argument
    movq  %rax, %rdi  # et on le place dans %rdi
    call  f           # on fait l'appel
    popq  %rdi        # on restaure %rdi
```

Question 6 Donner le code assembleur des fonctions prédéfinies `fst`, `snd` et `add`.

Correction :

```
fst:
        movq (%rdi), %rax
        ret
snd:
        movq 8(%rdi), %rax
        ret
add:
        movq (%rdi), %rax
        addq 8(%rdi), %rax
        ret
```

Question 7 Donner le code assembleur d'une fonction `print_list` qui reçoit en argument (dans `%rdi`) une valeur supposée être une liste d'entiers et affiche ses éléments dans l'ordre. On suppose l'existence d'une fonction `print_int` pour afficher un entier, avec les conventions d'appel usuelles.

Correction : On l'écrit comme une boucle :

```
print_list:
        testq %rdi, %rdi
        jz    Lexit
        pushq %rdi            # on sauvegarde %rdi
        movq  (%rdi), %rdi
        call  print_int
        popq  %rdi            # on restaure %rdi
        movq  8(%rdi), %rdi
        jmp   print_list      # on boucle
Lexit:
        ret
```

Question 8 On aimerait pouvoir écrire une fonction assembleur `print` qui affiche n'importe quelle valeur supposée bien formée, par exemple sous la forme

```
((1, 2), ((3, 4), 0))
```

Ce n'est malheureusement pas possible, car il faudrait être capable de faire la différence entre un entier et un pointeur (vers une paire). Proposer une solution pour y remédier.

Correction : Deux solutions au moins :
— celle d'OCaml consistant à représenter l'entier n par $2n+1$, en exploitant le fait qu'un pointeur est toujours pair ;
— représenter *toute* valeur par un pointeur sur le tas, vers un bloc contenant une étiquette indiquant s'il s'agit d'un entier ou d'une paire.

---

4 Analyse syntaxique

On cherche maintenant à construire un analyseur syntaxique pour notre langage. On se donne la grammaire

$$
\begin{array}{rcl}
S & \to & E \\
E & \to & \texttt{int} \\
 & | & \texttt{x} \\
 & | & \texttt{(}\ E\ \texttt{,}\ E\ \texttt{)} \\
 & | & \texttt{ifzero}\ E\ \texttt{then}\ E\ \texttt{else}\ E \\
 & | & \texttt{ident (}\ E\ \texttt{)}
\end{array}
$$

où S est l'axiome et où les terminaux sont `int` (une constante entière), `x`, `(`, `,`, `)`, `ifzero`, `then`, `else` et `ident` (un identificateur autre que `x`, `ifzero`, `then` et `else`).

Question 9 Montrer que cette grammaire est LR(0). Indication : on pourra construire l'automate déterministe directement ; il contient une vingtaine d'états.

Correction : Les états de l'automate LR(0) déterministe sont les suivants :

```
1.  S -> . E #                          -- E  --> 2
    E -> . int                          -- int --> 3
    E -> . x                            -- x  --> 4
    E -> . ( E , E )                    -- (  --> 5
    E -> . ifzero E then E else E       -- ifzero --> 6
    E -> . ident ( E )                  -- ident  --> 7

2. S -> E . #

3. E -> int .

4. E -> x .

5. E -> ( . E , E )                      -- E --> 5b
   et les 5 derniers items de l'état 1
5b E -> ( E . , E )                      -- , --> 5c
5c E -> ( E , . E )                      -- E --> 5d
   et les 5 derniers items de l'état 1
5d E -> ( E , E . )                      -- ) --> 5e
5e E -> ( E , E ) .

6. E -> ifzero . E then E else E         -- E --> 6b
   et les 5 derniers items de l'état 1
6b E -> ifzero E . then E else E         -- then --> 6c
6c E -> ifzero E then . E else E         -- E --> 6d
   et les 5 derniers items de l'état 1
6d E -> ifzero E then E . else E         -- else --> 6e
6e E -> ifzero E then E else . E         -- else --> 6f
   et les 5 derniers items de l'état 1
6f E -> ifzero E then E else E .

7. E -> ident . ( E )                    -- ( --> 7b
7b E -> ident ( . E )                    -- E --> 7c
   et les 5 derniers items de l'état 1
7c E -> ident ( E . )                    -- ) --> 7d
7d E -> ident ( E ) .
```

On a des réductions (uniques) dans les états 3, 4, 5e, 6f, 7d. À chaque fois, il n'y a aucune lecture, donc pas de conflit. La grammaire est donc LR(0)

Question 10 Que peut-on en déduire quant à l'analyse de cette grammaire par un outil tel que `cup` ou `menhir` ?

$$\frac{}{\Gamma \vdash n : \texttt{int}} \qquad \frac{\Gamma = \texttt{x} : \tau}{\Gamma \vdash \texttt{x} : \tau} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2}$$

$$\frac{f : \tau_1 \to \tau_2 \quad \Gamma \vdash e_1 : \tau_1}{\Gamma \vdash f(e_1) : \tau_2} \qquad \frac{\Gamma \vdash e_1 : \texttt{int} \times \texttt{int}}{\Gamma \vdash \texttt{add}(e_1) : \texttt{int}} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \times \tau_2}{\Gamma \vdash \texttt{fst}(e_1) : \tau_1} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \times \tau_2}{\Gamma \vdash \texttt{snd}(e_1) : \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \texttt{ifzero } e_1 \texttt{ then } e_2 \texttt{ else } e_3 : \tau}$$

FIGURE 4 – Typage.

Correction : La classe de grammaire LR(0) étant incluse dans la classe LALR(1), cette grammaire peut donc être analysée par les outils `cup` et `menhir`.

---

5 Typage

Dans cette partie, on va typer notre langage avec des types de la forme

τ	::=		**type**
		$\texttt{int}$	type d'un entier
		$\tau \times \tau$	type d'une paire

Un environnement de typage Γ est soit vide, soit réduit à $\texttt{x} : \tau$. On note $\Gamma \vdash e : \tau$ le jugement « dans l'environnement Γ, l'expression e a le type τ ». On note $f : \tau_1 \to \tau_2$ si la fonction f attend un argument de type τ_1 et renvoie un résultat de type τ_2. Les règles de typage des expressions sont données figure 4. Un programme est typable s'il existe un type pour chacune de ses fonctions de telle sorte que

— si le type donné à f est $\tau_1 \to \tau_2$ et `def` f`(x) =` e alors $\texttt{x} : \tau_1 \vdash e : \tau_2$;
— l'expression principale est bien typée dans l'environnement vide.

Question 11 Pour chacun des trois programmes suivants, indiquer s'il est typable. Si oui, donner un type à sa fonction. Sinon, justifier qu'il n'est pas typable.

1. ```
   def oups(x) = oups(x)
   oups(0)
   ```
2. ```
   def len(x) = ifzero x then 0 else add((1, len(snd(x))))
   len((1, (0, 0)))
   ```
3. ```
   def foo(x) = ifzero fst(x) then x else foo((fst(snd(x)), (snd(snd(x)), fst(x))))
   foo((1,(2,0)))
   ```

Correction :

1. oui, avec $\texttt{oups} : \texttt{int} \to \tau$ pour n'importe quel type τ
2. non, car on devrait avoir $\texttt{len} : \tau \to \texttt{int}$ pour un certain type τ, mais l'appel `len(snd(x))` impose alors $\tau = \tau_1 \times \tau_2$ avec $\tau_2 = \tau$, ce qui n'a pas de solution
3. oui, avec $\texttt{foo} : \texttt{int} \times (\texttt{int} \times \texttt{int}) \to \texttt{int} \times (\texttt{int} \times \texttt{int})$

   ```

Sûreté du typage. Comme vu en cours, la sûreté du typage se déduit des résultats de progrès et de préservation, qui font l'objet des questions suivantes.

Question 12 Montrer la propriété de progrès : si $\vdash e : \tau$, alors soit e est une valeur, soit il existe e' telle que $e \to e'$.

Correction : Par récurrence sur la dérivation $\vdash e : \tau$ et par cas sur la dernière règle :
- $e = n$ immédiat
- $e = \texttt{x}$ impossible (car Γ est vide)
- $e = (e_1, e_2)$: par HR e_1 est une valeur ou se réduit
  - si e_1 est une valeur, alors par HR e_2 est une valeur ou se réduit
    - e_2 est une valeur, alors e est une valeur
    - si e_2 se réduit, alors e se réduit
  - si e_1 se réduit, alors e se réduit
- $e = f(e_1)$: par HR e_1 est une valeur ou se réduit
  - si e_1 est une valeur, e se réduit quelle que soit la fonction ; en effet
    - si $\texttt{def } f\texttt{(x) = } e_2$ alors e se réduit vers $e_2[\texttt{x} \leftarrow e_1]$
    - si $f = \texttt{add}$ alors e_1 étant une valeur de type $\texttt{int} \times \texttt{int}$, c'est forcément une paire d'entiers, et donc e se réduit
    - si $f = \texttt{fst}$ (resp. $\texttt{snd}$), alors e_1 étant une valeur de type $\tau_1 \times \tau_2$ c'est forcément une paire, et donc e se réduit
  - si e_1 se réduit, alors e se réduit
- $e\texttt{ifzero } e_1 \texttt{ then } e_2 \texttt{ else } e_3$:par HR e_1 est une valeur ou se réduit
  - si e_1 est une valeur, e se réduit quelle que soit cette valeur (vers e_2 ou e_3 selon le cas)
  - si e_1 se réduit, alors e se réduit

Question 13 Montrer que si $\texttt{x} : \tau_1 \vdash e_2 : \tau_2$ et $\vdash e_1 : \tau_1$ alors $\vdash e_2[\texttt{x} \leftarrow e_1] : \tau_2$.

Correction : Récurrence immédiate sur la dérivation $\texttt{x} : \tau_1 \vdash e_2 : \tau_2$. Le seul cas particulier est celui de $e_2 = \texttt{x}$. Mais dans ce cas $\tau_1 = \tau_2$, ce qui permet de conclure.

Question 14 Montrer la propriété de préservation du typage : si $\vdash e : \tau$ et $e \to e'$, alors $\vdash e' : \tau$.

Correction : Par récurrence sur la dérivation $\vdash e : \tau$ et par cas sur la réduction :
- $e = (e_1, e_2) \to (e_1', e_2)$: on applique l'hypothèse de récurrence pour e_1
- $e = (v_1, e_2) \to (v_1, e_2')$: idem
- $e = f(e_1) \to f(e_1')$: idem
- $e = \texttt{ifzero } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \to \texttt{ifzero } e_1' \texttt{ then } e_2 \texttt{ else } e_3$: idem
- $e = f(v_1) \to e_2[\texttt{x} \leftarrow v_1]$ avec $\texttt{def } f\texttt{(x) = } e_2$: on a $\texttt{x} : \tau_1 \vdash e_2 : \tau_2$ d'une part (programme bien typé) et $\vdash e_1 : \tau_1$ d'autre part, donc le réduit est bien typé (de type τ_2) par le résultat de substitution (question précédente).
- $e = \texttt{add}((n_1, n_2)) \to n_1 + n_2$: les deux expressions ont bien le même type (à savoir $\texttt{int}$) ; idem pour la réduction de $\texttt{fst}$ ou $\texttt{snd}$

— $e = \texttt{ifzero}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \to e_2$ (resp. e_3) : les deux expressions ont bien le même type

---

Question 15 En déduire que, si $\vdash e : \tau$, alors la réduction de e est infinie ou termine sur une valeur.

Correction : Supposons la réduction finie

$$e \to e_1 \to e_2 \to \cdots \to e_n$$

avec e_n irréductible. Par récurrence, la préservation nous donne $\vdash e_i : \tau$ et en particulier $\vdash e_n : \tau$. Mais la propriété de progrès nous dit alors que e_n est une valeur.

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
`add` r_2`,` r_1	calcule $r_1 + r_2$ et l'affecte à r_1
`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
`test` r_2`,` r_1	positionne les drapeaux en fonction de la valeur de r_1 ET r_2
`jz` L	saute à l'adresse désignée par l'étiquette L si les drapeaux signalent un résultat nul
`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`.