

École Normale Supérieure
Langages de programmation et compilation
examen 2022–2023

Jean-Christophe Filliâtre
20 janvier 2023 — 8h30–11h30

L'épreuve dure 3 heures.

Les notes de cours manuscrites ou reprographiées sont les seuls documents autorisés.

Les questions sont indépendantes, au sens où il n'est pas nécessaire d'avoir répondu aux questions précédentes pour traiter une question, mais en revanche les questions peuvent faire appel à des définitions ou à des résultats introduits dans les questions précédentes.

Les figures 2–4 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 2. C'est une variante du langage mini-ML étudié en cours, avec des entiers, des fonctions et des *enregistrements*. Un enregistrement est construit avec la syntaxe $\{a = 1; b = 2\}$, ici avec deux champs a et b de valeurs respectives 1 et 2. Si r est un enregistrement possédant un champ a , alors $r.a$ désigne la valeur de ce champ. Dans la syntaxe abstraite, f désigne un nom de champ. Une sémantique opérationnelle à petits pas est donnée figure 3. Elle traduit un passage par valeur, avec une évaluation de la gauche vers la droite.

Question 1 Pour chacune des trois expressions suivantes, donner la suite de réductions et indiquer si elle se termine sur une valeur.

- $(\text{fun } (x:\{a:\text{int}\}) \rightarrow x.a) \{a=42; b=55\}$
- $\text{let } x = \{a=1\} \text{ in let } y = \{b=x.a; c=x.a\} \text{ in } y.a$
- $\text{let } x = \{a=1\} \text{ in } (\text{fun } (y:\{b:\text{int}\}) \rightarrow x.a) x$

Analyse syntaxique. On souhaite réaliser l'analyse syntaxique de notre petit langage avec l'outil Menhir. La figure 1 contient un fichier d'entrée pour Menhir contenant la grammaire des expressions de notre langage. Les actions sémantiques ne nous intéressent pas ici et sont omises ($\{\dots\}$).

Question 2 Lorsque l'outil Menhir est lancé sur le fichier donné en figure 1, il déclare de nombreux conflits de type lecture/réduction (*shift/reduce*).

1. Donner un exemple de conflit.
2. Pour lever les conflits, on choisit de donner la priorité la plus forte à la construction $e.f$, puis à l'application (construction $e e$) puis enfin à toutes les autres constructions. Donner des déclarations de priorité et d'associativité que l'on peut indiquer à l'outil Menhir pour cela. La grammaire ne doit pas être modifiée. On notera que la règle d'application est nommée.

Typage. Notre langage est typé avec des types monomorphes, notés τ et définis figure 2. Le jugement de typage est noté $\Gamma \vdash e : \tau$, où l'environnement de typage Γ donne le type des variables susceptibles d'apparaître dans e . Les règles de typage sont données figure 4. On ne cherche pas ici à faire de l'inférence de types, l'argument x d'une fonction $\text{fun } (x : \tau) \rightarrow e$ venant avec un type τ . Un enregistrement comme $\{a = 1; b = 2\}$ a un type de la forme $\{a : \text{int}; b : \text{int}\}$, qui donne le type de chacun de ses champs. (À la différence d'un langage comme OCaml, on ne déclare pas ici les types enregistrements.) Un enregistrement peut n'avoir aucun champ (c'est l'expression $\{\}$) et son type est alors $\{\}$.

expr :	
expr expr %prec app	{...}
LET IDENT EQ expr IN expr	{...}
FUN LPAR IDENT COLON typ RPAR ARROW expr	{...}
CONST	{...}
IDENT	{...}
expr DOT IDENT	{...}
LBRA separated_list(SEMI, field) RBRA	{...}
LPAR expr RPAR	{...}

FIGURE 1 – Fichier Menhir.

La particularité de ce langage est le *sous-typage* qui découle des types enregistrements. L'idée est qu'un enregistrement comme $\{a = 1; b = 2\}$ peut être considéré comme une valeur de type $\{b : \text{int}\}$, car il possède *au moins* le champ b attendu, avec le bon type. Plus précisément, on note $\tau \sqsubseteq \tau'$ la relation de sous-typage, qui s'interprète comme « toute valeur de type τ peut être acceptée comme une valeur de type τ' ». Cette relation est définie par les trois dernières règles de la figure 4. La relation de sous-typage est utilisée dans la règle de typage de l'application, pour accepter un argument e_1 d'un sous-type τ du type τ_1 attendu par la fonction e_2 .

Question 3 Pour chacune des trois expressions suivantes, indiquer si elle est bien typée dans un environnement de typage vide. Le cas échéant, donner la dérivation de typage. Dans le cas contraire, justifier.

- $(\text{fun } (x:\{a:\text{int}\}) \rightarrow x.a) \{a=42; b=55\}$
- $\text{fun } (x:\{a:\text{int}\}) \rightarrow \{b:\text{int}\} \rightarrow x \{\}$
- $\text{let } x = \{\} \text{ in } \{ b=x; c=x \}$

Question 4 Parmi les règles de la figure 4, on a notamment défini le sous-typage des types de fonctions de la manière suivante :

$$\frac{\tau'_1 \sqsubseteq \tau_1 \quad \tau_2 \sqsubseteq \tau'_2}{\tau_1 \rightarrow \tau_2 \sqsubseteq \tau'_1 \rightarrow \tau'_2}$$

Justifier cette règle en donnant un exemple impliquant du sous-typage de fonctions, avec des types $\tau_1 \neq \tau'_1$ et $\tau_2 \neq \tau'_2$.

Question 5 Donner un exemple de « diminution » du typage pendant l'exécution, c'est-à-dire une expression e telle que $\vdash e : \tau$ et une réduction $e \rightarrow e'$ telle que $\vdash e' : \tau'$ avec $\tau' \neq \tau$.

Sûreté du typage. On se propose maintenant de montrer que le typage de notre langage est sûr, au sens où l'évaluation d'une expression bien typée, si elle termine, conduit nécessairement à une valeur. On admet la propriété de progrès (si une expression e est bien typée et n'est pas une valeur, alors $e \rightarrow e'$) et on se concentre ici sur la propriété de préservation, qui fait l'objet des questions suivantes.

Question 6 Montrer que la relation de sous-typage \sqsubseteq est transitive, c'est-à-dire que, si $\tau_1 \sqsubseteq \tau_2$ et $\tau_2 \sqsubseteq \tau_3$, alors $\tau_1 \sqsubseteq \tau_3$.

Question 7 Montrer que si $\Gamma + x : \tau_1 \vdash e_2 : \tau_2$ et $\tau'_1 \sqsubseteq \tau_1$ alors $\Gamma + x : \tau'_1 \vdash e_2 : \tau_2$ avec $\tau'_2 \sqsubseteq \tau_2$. Indiquer sur quoi porte l'induction et traiter uniquement le cas d'une application, i.e., $e_2 = e_4 e_3$.

Question 8 Montrer la préservation du typage par substitution, à savoir que si $\Gamma + x : \tau_1 \vdash e_2 : \tau_2$ et $\Gamma \vdash e_1 : \tau'_1$ avec $\tau'_1 \sqsubseteq \tau_1$ alors $\Gamma \vdash e_2[x \leftarrow e_1] : \tau'_2$ avec $\tau'_2 \sqsubseteq \tau_2$. Indiquer sur quoi porte l'induction et traiter uniquement les cas d'une application, i.e., $e_2 = e_4 e_3$, et d'une projection, i.e., $e_2 = e_3.f$.

Question 9 Montrer la propriété de préservation du typage, à savoir que si $\Gamma \vdash e : \tau$ et $e \rightarrow e'$, alors $\Gamma \vdash e' : \tau'$ avec $\tau' \sqsubseteq \tau$. Indiquer sur quoi porte l'induction et traiter uniquement les cas d'une application, i.e., $e = e_2 e_1$, et d'un **let**, i.e., $e = \text{let } x = e_1 \text{ in } e_2$.

Ajout d'une conditionnelle. On se propose d'ajouter une conditionnelle à notre langage, sous la forme d'une construction **ifzero** e_1 **then** e_2 **else** e_3 qui évalue l'expression e_1 , de type **int**, puis évalue l'expression e_2 si le résultat est zéro et l'expression e_3 sinon.

Question 10 Ajouter cette nouvelle construction à la sémantique opérationnelle à petits pas de la figure 3, c'est-à-dire indiquer les modifications à apporter, le cas échéant, (1) à la notion de valeur, (2) aux réductions de tête et (3) aux contextes de réduction.

Type intersection. On aimerait accepter au typage des programmes comme

```
if ... then { a=1; b=2 } else { b=3; c=4 }
```

ou encore

```
if ... then fun (x:{a:int;b:int}) -> 0
           else fun (x:{c:int})      -> 1
```

Pour cela, on propose la règle de typage suivante

$$\frac{\Gamma \vdash e : \text{int} \quad \Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \text{ifzero } e \text{ then } e_1 \text{ else } e_2 : \tau_1 \vee \tau_2}$$

où l'opération $\tau_1 \vee \tau_2$ désigne le plus petit sur-type de τ_1 et τ_2 , s'il existe, c'est-à-dire,

- $\tau_1 \sqsubseteq \tau_1 \vee \tau_2$
- $\tau_2 \sqsubseteq \tau_1 \vee \tau_2$
- pour tout type τ , si $\tau_1 \sqsubseteq \tau$ et $\tau_2 \sqsubseteq \tau$ alors $\tau_1 \vee \tau_2 \sqsubseteq \tau$.

Question 11 Dans chacun des cas suivants, indiquer si $\tau_1 \vee \tau_2$ existe et donner sa valeur le cas échéant :

τ_1	τ_2	$\tau_1 \vee \tau_2$
$\{a : \text{int}; b : \text{int}\}$	$\{b : \text{int}; c : \text{int}\}$???
$\{a : \text{int}; b : \text{int}\} \rightarrow \text{int}$	$\{c : \text{int}\} \rightarrow \text{int}$???
int	int \rightarrow int	???

Question 12 Donner une définition *algorithmique* de l'opération \vee .

Compilation. On se propose maintenant de compiler notre petit langage. Une valeur de la forme $\{f_1 = v_1; \dots; f_n = v_n\}$ est l'adresse d'un bloc mémoire alloué sur le tas, de la forme

v_1	v_2	\dots	v_n
-------	-------	---------	-------

où les valeurs v_1, \dots, v_n des champs f_1, \dots, f_n sont rangées en mémoire dans l'ordre alphabétique des noms des champs (en supposant donc ici $f_1 < f_2 < \dots < f_n$). Cette représentation des valeurs pose un problème évident. Dans une application comme

```
(fun (x:{b:int}) -> x.b) {a=1;b=2;c=3}
```

la valeur construite pour le paramètre effectif contient trois champs, et le champ b y est rangé en deuxième position, mais le corps de la fonction a été compilé en supposant que x contient un seul champ, rangé à la première position.

Pour y remédier, on va transformer le programme avant de le compiler, pour modifier toute valeur dont le type « change » à cause du sous-typage. Ainsi, dans l'exemple ci-dessus, on va transformer la valeur $\{a=1;b=2;c=3\}$ en la valeur $\{b=2\}$ avant de la passer à la fonction. Pour cela, on introduit une fonction $C(e, \tau)$ qui transforme une expression e de type τ' en une expression de type τ dès lors que $\tau' \sqsubseteq \tau$. Avec cette fonction, on va modifier le programme e en un programme $T(e)$ de la façon suivante :

$$T(e_2 \ e_1) = T(e_2) (C(T(e_1), \tau_1))$$

$$T(\text{ifzero } e_1 \text{ then } e_2 \text{ else } e_3) = \text{ifzero } T(e_1) \text{ then } C(T(e_2), \tau_2 \vee \tau_3) \text{ else } C(T(e_3), \tau_2 \vee \tau_3)$$

où τ_1 d'une part et τ_2, τ_3 d'autre part sont les types impliqués dans le typage de ces deux constructions. Partout ailleurs, T est un morphisme, c'est-à-dire

$$T(n) = n$$

$$T(x) = x$$

$$T(\text{fun } (x : \tau) \rightarrow e) = \text{fun } (x : \tau) \rightarrow T(e)$$

$$T(\text{let } x = e_1 \text{ in } e_2) = \text{let } x = T(e_1) \text{ in } T(e_2)$$

$$T(e.f) = T(e).f$$

$$T(\{f_1 = e_1; \dots; f_n = e_n\}) = \{f_1 = T(e_1); \dots; f_n = T(e_n)\}$$

Question 13 Donner une définition de la fonction C .

Compilation vers l'assembleur x86-64. (Un aide-mémoire est donné en annexe.) On adopte le schéma de compilation suivant. Toutes les valeurs occupent un mot mémoire de 64 bits. Une valeur entière n est directement représentée par un entier 64 bits. Une valeur de la forme $\text{fun } (x : \tau) \rightarrow e$ est l'adresse d'un bloc mémoire alloué sur le tas représentant une fermeture, c'est-à-dire un bloc de $n + 1$ mots de la forme

code	v_1	...	v_n
------	-------	-----	-------

où *code* est l'adresse du code correspondant à la fonction et v_1, \dots, v_n sont les valeurs des variables libres x_1, \dots, x_n de $\text{fun } (x : \tau) \rightarrow e$, dans un ordre arbitraire choisi par le compilateur. Enfin, une valeur de type enregistrement est l'adresse d'un bloc mémoire contenant les valeurs de ses champs, comme expliqué plus haut. Dans le code d'une fermeture, l'unique argument est passé dans `%rdi`, la fermeture est passée dans `%rsi` et le résultat est renvoyé dans `%rax`.

Question 14 Expliquer pourquoi il n'y a pas de risque de confondre, à l'exécution,

- un entier avec une adresse ;
- l'adresse d'une fermeture avec l'adresse d'un enregistrement.

Question 15 On considère un programme de la forme

```
let x = ... in
let y = fun (z: int) -> { a = x; b = z } in
...
```

1. À quoi ressemble la fermeture représentant la valeur de y ?
2. Donner un code assembleur pour cette fermeture. (On ne demande pas en revanche le code qui construit la fermeture, ni le code qui l'applique.)

Question 16 On souhaite ajouter une primitive `add` à notre langage, sous la forme d'une fonction prédéfinie de type `int → (int → int)`. On peut alors écrire une expression comme `(add 40) 2`, où la première application `(add 40)` construit une fermeture, qui est ensuite appliquée à `2`. La fonction `add` elle-même est représentée par une fermeture dont l'environnement est vide. Donner le code assembleur de ces deux fermetures.

Ajout de références. On ajoute enfin une dernière chose à notre langage, à savoir des références à la OCaml, c'est-à-dire les trois constructions :

$$\begin{array}{l}
 e ::= \dots \\
 \quad | \text{ref } e \quad \textit{création} \\
 \quad | !e \quad \textit{accès} \\
 \quad | e := e \quad \textit{affectation}
 \end{array}$$

La sémantique est la même que celle d'OCaml : une référence est un bloc alloué sur le tas, contenant une unique valeur, à laquelle on accède avec `!` et que l'on modifie avec `:=`. Les références nous permettent notamment de définir des fonctions *récurrentes*, en utilisant la technique dite du *nœud de Landin* :

1. créer une référence r contenant une fonction arbitraire ;
2. affecter à r une fonction qui utilise `!r` pour effectuer des appels récursifs.

Question 17 Donner une expression de type `int → int` qui, appliquée à un entier $n \geq 0$, renvoie le terme F_n de la suite de Fibonacci. (À toutes fins utiles, on rappelle que la suite de Fibonacci (F_n) est définie par $F_0 = 0$, $F_1 = 1$ et $F_{n+2} = F_n + F_{n+1}$ pour $n \geq 0$.) On pourra utiliser librement la fonction `add` introduite à la question 16 et remarquer qu'une constante n peut être négative.

Typage des références. On introduit un nouveau constructeur de types pour les références :

$$\begin{array}{l}
 \tau ::= \dots \\
 \quad | \text{ref } \tau \quad \textit{type d'une référence}
 \end{array}$$

On se donne les règles de typage suivantes :

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \text{ref } e : \text{ref } \tau} \quad \frac{\Gamma \vdash e : \text{ref } \tau}{\Gamma \vdash !e : \tau} \quad \frac{\Gamma \vdash e_1 : \text{ref } \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \tau_2 \sqsubseteq \tau_1}{\Gamma \vdash e_1 := e_2 : \{\}}$$

Question 18 On pourrait être tenté d'étendre la définition de \sqsubseteq en ajoutant la règle suivante :

$$\frac{\tau' \sqsubseteq \tau}{\text{ref } \tau' \sqsubseteq \text{ref } \tau}$$

1. Montrer que cette règle n'est pas sûre.
2. Proposer quelque chose de correct. On ne demande pas de justification.

Langage objet. Avec le langage initial et les références que nous venons d'ajouter, on a maintenant un langage qui permet de modéliser des *objets* au sens de la programmation orientée objets. Ainsi, on peut définir un objet « compteur » c de la façon suivante :

```
let c =
  let s = {n = ref 0} in
  { get = fun (x : {}) → !(s.n);
    inc = fun (x : {}) → s.n := (add !(s.n)) 1 }
  in ...
```

(On utilise ici la fonction `add` introduite à la question 16.) Il s'agit d'un enregistrement contenant des *méthodes* (ici `get` et `inc`) agissant sur un *état* interne à l'objet (l'enregistrement `s`, contenant ici un unique attribut `n`). Pour appeler une méthode, il suffit de faire

`c.get {}`

c'est-à-dire d'extraire la valeur correspondant à un champ (ici `get`) et de l'appliquer (ici à un enregistrement vide). Voici un exemple plus développé, avec à gauche du code Java et à droite sa traduction dans notre langage :

<pre>class A { int n = 0; int get() { return n; } void inc() { n += 1; } } class B extends A { int i = 42; void reset() { n=i; } } void inc2(A a) { a.inc(); a.inc(); } ... inc2(new B());...</pre>	<pre>let methA = fun (s:{n:ref int}) -> { get = fun (_:{}) -> !(s.n); inc = fun (_:{}) -> s.n := add !(s.n) 1 } in let newA = fun (_:{}) -> let s = { n = ref 0 } in methA s in let methB = fun (s:{n:ref int; i:ref int}) -> let super = methA s in { get = super.get; inc = super.inc; reset = fun (_:{}) -> s.n := !(s.i) } in let newB = fun (_:{}) -> let s = { n = ref 0; i = ref 42 } in methB s in let inc2 = fun (a:{get:{}->int;inc:{}->{}}) -> let _ = a.inc {} in a.inc {} in ... inc2 (newB {}) ...</pre>
--	---

Question 19 Indiquer à quels endroits du code ci-dessus le sous-typage a été utilisé.

Question 20 On souhaite maintenant faire la même chose pour les deux classes suivantes :

<pre>class C { int n = 0; int get() { inc(); return n; } void inc() { n += 1; } }</pre>	<pre>class D extends C { void inc() { n += 2; } }</pre>
---	---

Ce cas de figure est plus subtile que le précédent, car la méthode `get` de la classe `D`, héritée de `C`, doit appeler la méthode `inc` de la classe `D` (appel dynamique). Proposer un schéma de traduction.

$e ::= n$	<i>constante</i> $n \in \mathbb{Z}$
x	<i>variable</i>
$\mathbf{fun} (x : \tau) \rightarrow e$	<i>fonction</i>
$e e$	<i>application</i>
$\mathbf{let} x = e \mathbf{in} e$	<i>variable locale</i>
$e.f$	<i>accès au champ</i>
$\{f = e; \dots; f = e\}$	<i>création enregistrement</i>
$\tau ::= \mathbf{int}$	<i>type des entiers</i>
$\tau \rightarrow \tau$	<i>type des fonctions</i>
$\{f : \tau; \dots; f : \tau\}$	<i>type des enregistrements</i>

FIGURE 2 – Syntaxe abstraite.

valeurs

$v ::= n$	
$\mathbf{fun} (x : \tau) \rightarrow e$	
$\{f = v; \dots; f = v\}$	

réductions de tête

$(\mathbf{fun} (x : \tau) \rightarrow e) v$	$\xrightarrow{\epsilon} e[x \leftarrow v]$
$\mathbf{let} x = v \mathbf{in} e$	$\xrightarrow{\epsilon} e[x \leftarrow v]$
$\{f_1 = v_1; \dots; f_n = v_n\}.f_i$	$\xrightarrow{\epsilon} v_i$

contextes de réduction

$E ::= \square$	
$E e$	
$v E$	
$\mathbf{let} x = E \mathbf{in} e$	
$E.f$	
$\{f = v; \dots; f = v; f = E; f = e; \dots; f = e\}$	

réduction

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

FIGURE 3 – Sémantique opérationnelle à petits pas.

$\frac{}{\Gamma \vdash n : \mathbf{int}}$	$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}$	$\frac{\Gamma + x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash (\mathbf{fun} (x : \tau_1) \rightarrow e) : \tau_1 \rightarrow \tau_2}$
$\frac{\Gamma \vdash e_2 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_1 : \tau \quad \tau \sqsubseteq \tau_1}{\Gamma \vdash e_2 e_1 : \tau_2}$	$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma + x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \mathbf{let} x = e_1 \mathbf{in} e_2 : \tau_2}$	
$\frac{\Gamma \vdash e : \{\dots; f : \tau; \dots\}}{\Gamma \vdash e.f : \tau}$	$\frac{f_i \text{ distincts} \quad \forall i, \Gamma \vdash e_i : \tau_i}{\Gamma \vdash \{f_1 = e_1; \dots; f_n = e_n\} : \{f_1 : \tau_1; \dots; f_n : \tau_n\}}$	
$\frac{}{\tau \sqsubseteq \tau}$	$\frac{\tau'_1 \sqsubseteq \tau_1 \quad \tau_2 \sqsubseteq \tau'_2}{\tau_1 \rightarrow \tau_2 \sqsubseteq \tau'_1 \rightarrow \tau'_2}$	$\frac{\forall 1 \leq j \leq m, \exists 1 \leq i \leq n, f'_j = f_i \text{ et } \tau_i \sqsubseteq \tau'_j}{\{f_1 : \tau_1; \dots; f_n : \tau_n\} \sqsubseteq \{f'_1 : \tau'_1; \dots; f'_m : \tau'_m\}}$

FIGURE 4 – Règles de typage.

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.

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