École Normale Supérieure

Langages de programmation et compilation

examen 2018–2019

Jean-Christophe Filliâtre

25 janvier 2019

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

Dans ce tout ce sujet, on étudie FORTH, un langage à piles datant de 1970. Un programme FORTH est une suite de *mots* séparés par des caractères blancs (espace, tabulation, retour-chariot). Un mot peut être composé de n'importe quelle séquence de caractères non blancs. Ainsi, le programme

```
42 dup emit emit 10 emit
```

contient exactement six mots. L'exécution d'un programme FORTH utilise une pile contenant des valeurs entières. Initialement la pile est vide. On exécute alors chaque mot en séquence. Si le mot correspond à une opération prédéfinie, comme `dup` ou `emit`, on effectue cette opération. Par exemple, `dup` duplique la valeur se trouvant au sommet de la pile et `emit` dépile un entier et affiche le caractère dont cet entier est le code ASCII. Si en revanche le mot à exécuter ne correspond pas à une opération prédéfinie, alors ce mot doit être une constante entière et sa valeur est mise au sommet de la pile. Ainsi, le programme ci-dessus affiche deux fois le caractère `*` (de code ASCII 42) suivi d'un retour-chariot (de code ASCII 10).

Les deux mots `if` et `endif` sont interprétés de façon particulière. Chaque occurrence d'un mot `if` doit nécessairement être associée à un mot `endif` se trouvant plus loin dans le texte du programme. Lorsque le mot `if` est rencontré, on dépile une valeur. Si elle est nulle, l'exécution se poursuit après la première occurrence du mot `endif` qui suit. Sinon, l'exécution se poursuit immédiatement après le mot `if`. Lorsque l'exécution arrive plus tard au mot `endif`, celui-ci est ignoré et l'exécution se poursuit au-delà. Ainsi, le programme

```
dup if dup emit endif drop
```

a pour effet de dépiler un entier et d'afficher le caractère correspondant uniquement s'il est non nul. L'opération prédéfinie `drop` dépile une valeur. L'ensemble des opérations prédéfinies est donné figure 1.

Question 1 Donner un programme qui attend deux entiers n_1 et n_2 au sommet de la pile et affiche soit le caractère `!` (code ASCII 33) s'ils sont différents, soit le caractère `=` (code ASCII 61) s'ils sont égaux. Les deux entiers doivent être dépilés au final.

Correction :

```
- dup if 33 emit endif 0= if 61 emit endif
```

---

primitive	sémantique
`dup`	duplique le sommet de la pile
`emit`	dépile un entier n et affiche le caractère de code ASCII n
`drop`	dépile un entier
`-`	dépile un entier n_2 puis un entier n_1 et empile l'entier $n_1 - n_2$
`0=`	dépile un entier n et empile soit -1 si $n = 0$, soit 0 si $n \neq 0$
`@`	dépile un entier a et empile la valeur se trouvant en mémoire à l'adresse a
`!`	dépile un entier a puis un entier n et écrit l'entier n en mémoire à l'adresse a

FIGURE 1 – Opérations primitives.

Grammaire. Pour effectuer l'analyse syntaxique du langage FORTH, on se donne la grammaire suivante :

$$\begin{array}{rcl} S & \to & L\ \texttt{\#} \\ C & \to & \texttt{word} \\ & | & \texttt{if}\ L\ \texttt{endif} \\ L & \to & \epsilon \\ & | & C\ L \end{array}$$

L'ensemble des terminaux est $\{\texttt{word}, \texttt{if}, \texttt{endif}, \texttt{\#}\}$ et l'ensemble des non terminaux est $\{S, C, L\}$. Le symbole de départ est S.

Question 2 Calculer NULL, FIRST et FOLLOW pour chacun des non terminaux de cette grammaire.

Correction : On a clairement $\text{NULL}(S) = \text{NULL}(C) = \texttt{false}$ et $\text{NULL}(L) = \texttt{true}$.
On calcule facilement $\text{FIRST}(C) = \text{FIRST}(L) = \{\texttt{word}, \texttt{if}\}$ et $\text{FIRST}(S) = \{\texttt{word}, \texttt{if}, \texttt{\#}\}$, puis $\text{FOLLOW}(E) = \{\texttt{word}, \texttt{if}, \texttt{endif}, \texttt{\#}\}$. et $\text{FOLLOW}(L) = \{\texttt{endif}, \texttt{\#}\}$.

Question 3 Cette grammaire est-elle LL(1) ? LR(0) ? SLR(1) ? LR(1) ? ambiguë ? Justifier à chaque fois.

Correction : La table d'expansion est

	`word`	`if`	`endif`	#
S	$L\#$	$L\#$		$L\#$
C	`word`	`if`L`endif`		
L	CL	CL	ϵ	ϵ

donc la grammaire est LL(1).
Dans l'automate LR(0), on a trois états de la forme

$$\begin{array}{|rcl|} L & \to & \bullet \\ L & \to & \bullet CL \\ C & \to & \bullet\texttt{word} \\ C & \to & \bullet\texttt{if}L\texttt{endif} \\ & & \dots \end{array}$$

et contenant donc un conflit lecture/réduction (lecture de `word` ou `if` et réduction de $L \to \epsilon$). Donc la grammaire n'est pas LR(0). Ce conflit disparaît dans l'analyse SLR(1), car

FOLLOW(L) = {`endif`, `#`}. Les cinq autres états de l'automate LR(0) ne contiennent pas de conflit (ni lecture/réduction, ni réduction/réduction). La grammaire est donc SLR(1). La grammaire est LR(1) par inclusion car elle est LL(1) (ou SLR(1), au choix). La grammaire est non ambiguë par inclusion car elle est LL(1) (ou SLR(1), ou LR(1), au choix).

Variables et fonctions. Le langage FORTH contient également des déclarations de variables et de fonctions. Une variable est déclarée avec la syntaxe

`variable` *ident*

où *ident* est le nom de la variable. Après une telle déclaration, le nom *ident* peut être utilisé et son effet est de déposer sur la pile l'adresse où la variable est stockée en mémoire. Grâce aux opérations primitives `!` et `@` données figure 1, on peut donc accéder au contenu d'une variable et le modifier. Ainsi, le programme

```
variable x
42 x !
x @ emit
```

déclare une variable `x`, y stocke l'entier 42, puis affiche le caractère dont le code est stocké dans la variable. Une fonction est déclarée avec la syntaxe

`:` *ident ...code...* `;`

La définition commence avec le mot « `:` » et se termine avec le mot « `;` ». Le mot immédiatement après « `:` », noté ici *ident*, est le nom de la fonction. Le corps de la fonction, noté ici « *...code...* », est une liste de mots obéissant à la grammaire L donnée plus haut. En particulier, le corps de la fonction ne peut pas contenir récursivement de définition de fonction, ni une définition de variable. On appelle une fonction en donnant son nom et sa définition est alors exécutée. Ainsi, le programme

```
: print
    dup if dup emit endif drop ;
: print3
    dup print dup print print 10 emit ;
```

définit deux fonctions, dont une fonction `print3` qui dépile un entier et affiche trois fois le caractère correspondant, s'il est non nul, puis un retour-chariot. Le corps d'une fonction ne peut faire référence qu'à des fonctions et des variables définies *précédemment* dans le programme. En particulier, une fonction ne peut pas être récursive.

Question 4 Donner le code d'une fonction `swap` qui échange les deux valeurs au sommet de la pile. Est-il possible de le faire en utilisant une seule variable ?

Correction : C'est évidemment très facile en utilisant deux variables :

```
variable tmp1
variable tmp2
: swap
    tmp1 ! tmp2 ! tmp1 @ tmp2 @
;
```

C'est faisable avec une seule variable, mais plus difficile :

```
variable tmp
: smart_swap          ( a b -- tmp:?   )
    tmp !             ( a   -- tmp:b   )
    dup tmp @ - tmp ! ( a   -- tmp:a-b )
    tmp @ - dup       ( b b -- tmp:a-b )
    0 tmp @ - -       ( b a -- tmp:a-b )
;
```

Les parenthèses sont des commentaires illustrant le sommet de la pile et le contenu de `tmp`.

Analyse syntaxique avec Menhir. La figure 3 contient la syntaxe complète du langage FORTH, au format de l'outil Menhir. Volontairement, on n'utilise pas ici les fonctionnalités de Menhir pour reconnaître des listes (`+`, `*`, `nonempty_list`, etc.). On note qu'il y a cinq mot-clés, à savoir

`if   endif   variable   :   ;`

Cet analyseur syntaxique renvoie des arbres de syntaxe abstraite dont les types OCaml sont donnés figure 2.

Question 5 Lorsque l'outil Menhir est lancé sur ce fichier, il déclare un conflit de type *shift/reduce*. Identifier et expliquer ce conflit. Expliquer ensuite comment le résoudre en ajoutant dans ce fichier des indications de priorité et/ou d'associativité. Les règles de grammaire ne doivent pas être modifiées.

Correction : Le conflit provient de la lecture d'une entrée telle que

```
  WORD WORD
```

Après la lecture du premier mot, on a le choix entre réduire `codes->` (pour construire une liste vide et donc une liste `Run` avec un élément) ou lire le mot suivant (pour construire une liste `Run` à deux éléments). Plus généralement, on voit bien qu'une liste de `code` (que ce soit `WORD` ou `IF`), peut être décomposée de plusieurs façons en listes de listes dès lors qu'elle contient au moins deux éléments. Il y a donc ambiguïté.

Une façon de résoudre le problème consiste à construire des listes `Run` les plus longues possibles, *i.e.*, jusqu'à la prochaine déclaration de variable ou de fonction ou la fin du fichier. Pour cela, il faut privilégier la lecture de `WORD` et `IF` par rapport à la réduction de `codes->`. On le fait en déclarant les priorités

```
%nonassoc prec_codes
%nonassoc WORD IF
```

et en ajoutant la priorité `prec_codes` à la règle correspondant à la réduction :

```
codes:
| %prec prec_codes { []     }
| c=code l=codes   { c :: l }
;
```

Il n'y a pas d'autre conflit.

```
type code =
  | Wrd of string               (* un mot             *)
  | If  of code list            (* if ... endif       *)
type decl =
  | Var of string               (* variable x         *)
  | Def of string * code list   (* : f ... ;          *)
  | Run of code list            (* du code à exécuter *)
type file =
  decl list
```

Figure 2 – Types OCaml pour la syntaxe abstraite (dans un fichier `ast.mli`).

```
%{ open Ast %}

%token <string> WORD
%token COLON SEMICOLON VARIABLE IF ENDIF
%token EOF

%start file
%type  <Ast.file> file

%%

file:
| EOF           { []     }
| d=decl f=file { d :: f }
;

decl:
| VARIABLE x=WORD                { Var x        }
| COLON f=WORD c=codes SEMICOLON { Def (f, c)   }
| c=code l=codes                 { Run (c :: l) }
;

codes:
|                  { []     }
| c=code l=codes   { c :: l }
;

code:
| IF l=codes ENDIF { If  l }
| w=WORD           { Wrd w }
;
```

Figure 3 – Analyse syntaxique avec Menhir.

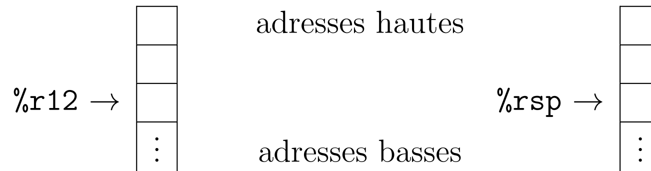
compile(x)	=	`addq $-8, %r12` `movq $`x`, (%r12)`	si x est une variable ou un entier
compile(x)	=	`call` x	si x est une fonction

compile(`if` *code*... `endif`)	=	`movq (%r12), %rcx`
		`addq $8, %r12`
		`testq %rcx, %rcx`
		`jz` L
		compile(*code*)...
		L`:`
			avec L une étiquette fraîche

compile(`:` x *code*... `;`)	=	x`:` *compile*(*code*)... `ret`	compilation d'une fonction
compile(`variable` x)	=	`.data` x`: .quad` 0	compilation d'une variable

FIGURE 4 – Compilation de FORTH vers x86-64.

Compilation vers x86-64. On se préoccupe maintenant de la compilation du langage FORTH vers l'assembleur x86-64. (Un petit aide-mémoire x86-64 est donné à la fin du sujet.) Toutes les valeurs manipulées occupent 64 bits. Elles sont interprétées soit comme des adresses, soit comme des entiers, selon le contexte. Il y a *deux piles* matérialisées dans la mémoire. La pile de FORTH, dont nous avons parlé jusqu'à présent, est matérialisée quelque part sur le tas et son sommet est repéré grâce au registre `%r12`. Elle croît en direction des adresses basses. L'autre pile est la pile usuelle de x86-64, dont le sommet est repéré par le registre `%rsp` et qui croît également en direction des adresses basses. La première pile est appelée la *pile de données* et la seconde est appelée la *pile de retour*.



On pourra supposer que la compilation du programme commence par quelque chose comme

```
movq $0x4000, %rdi
call sbrk
leaq 0x4000(%rax), %r12
```

qui alloue de l'espace sur le tas pour la pile de données (ici 16 ko). La pile de retour s'utilise de la manière usuelle. En particulier, l'instruction assembleur `call` dépose l'adresse de retour au sommet de cette pile et l'instruction `ret` la dépile et y revient.

La compilation d'un programme FORTH est très simple. Toutes les variables sont allouées statiquement sur le segment de données. Chaque fonction FORTH devient une fonction assembleur, dont le code est obtenu en compilant successivement toutes les instructions qui la composent. La compilation de chaque instruction est donnée figure 4. Enfin, la compilation d'un programme contient également le code assembleur de chaque opération primitive. Ainsi, le code assembleur de la primitive `emit` est le suivant :

```
emit:
```

```
movq (%r12), %rdi
addq $8, %r12
call putchar
ret
```

Question 6 Donner le code assembleur des primitives `dup`, `drop`, `-`, `0=`, `@` et `!`.

Correction :

```
dup:
        movq (%r12), %rax
        addq $-8, %r12
        movq %rax, (%r12)
        ret
drop:
        addq $8, %r12
        ret
sub:
        movq (%r12), %rax
        subq %rax, 8(%r12)
        addq $8, %r12
        ret
is_zero:
        movq (%r12), %rcx
        testq %rcx, %rcx
        jz   1f
        movq $0, (%r12)
        ret
1:      movq $-1, (%r12)
        ret
fetch:
        movq (%r12), %rcx
        movq (%rcx), %rcx
        movq %rcx, (%r12)
        ret
store:
        movq (%r12), %rcx   # adresse
        movq 8(%r12), %rax  # valeur
        addq $16, %r12
        movq %rax, (%rcx)
        ret
```

Des boucles. Pour l'instant, les programmes que nous pouvons écrire sont extrêmement limités, car il n'y a aucune notion de boucle. (On rappelle que les fonctions ne sont pas récursives). On va y remédier en ajoutant deux nouvelles opérations primitives, `begin` et `until`, de telle sorte que

`begin` $code_1 \ldots$ `until` $code_2 \ldots$

exécute $code_1$ tant que la valeur trouvée en sommet de pile (et dépilée) par `until` est nulle. Ensuite, l'exécution se poursuit avec $code_2$. Ainsi, la fonction `many-stars` définie par

```
: many-stars
    begin 42 emit 1 - dup 0= until drop 10 emit ;
```

dépile un entier n, supposé strictement supérieur à zéro, et affiche n fois le caractère `*`, suivi d'un retour-chariot.

Question 7 Donner le code FORTH d'une fonction `fibonacci` qui trouve au sommet de la pile un entier $n \geq 1$ et affiche n lignes de texte contenant respectivement $F_1, F_2, \ldots, F_n$ caractères `*`, dans cet ordre[1]. Ainsi, l'exécution de `6 fibonacci` doit donner la sortie

```
*
*
**
***
*****
********
```

On pourra se resservir de la fonction `many-stars` donnée plus haut. L'entier n doit être dépilé au final.

Correction : Le plus simple est de se servir de deux variables pour stocker F_k et F_{k+1}. Le nombre d'étapes est manipulé au sommet de la pile.

```
variable a
variable b
: fibonacci
    0 a ! 1 b !
    begin
        b @ many-stars
        b @ dup 0 a @ - - b ! a !
        1 - dup 0=
    until
    drop ;
```

Compilation de `begin` et `until`. Nous allons ajouter les deux primitives `begin` et `until` *sans changer* ni la syntaxe, ni la compilation de notre langage, mais uniquement en fournissant du code assembleur pour ces deux nouvelles primitives. L'idée est d'utiliser la pile de retour pour y stocker l'adresse du code situé immédiatement après `begin` et pouvoir ainsi y revenir autant de fois que nécessaire lorsque l'on exécute `until`.

Question 8 Donner le code assembleur des primitives `begin` et `until`.

Correction : Pour `begin`, c'est très simple : il suffit de dupliquer l'adresse de retour située au sommet de la pile de retour.

1. S'il faut le rappeler, on a $F_0 = 0$, $F_1 = 1$ et $F_n = F_{n-2} + F_{n-1}$ pour $n \geq 2$.

```
begin:
        pushq (%rsp)
        ret
```

Pour `until`, c'est un peu plus compliqué. On commence par dépiler la valeur sur la pile de données pour la comparer à zéro.

```
until:
        movq (%r12), %rcx
        addq $8, %r12
        testq %rcx, %rcx
        jz 1f
```

Si elle est non nulle, il faut s'arrêter. Pour cela, on supprime l'adresse de `begin` qui se trouve juste au dessus de l'adresse de retour (de `until`).

```
        popq %rcx
        popq %rax    # dépiler l'adresse de la boucle
        pushq %rcx
        ret
```

Si en revanche elle est nulle, il faut continuer. Comme on va revenir juste après le `begin`, il ne faut pas oublier de redupliquer l'adresse de retour de `begin` avant de la suivre.

```
1:      popq %rcx    # oublier l'adresse de retour
        pushq (%rsp) # dupliquer l'adresse de la boucle
        ret          # et recommencer
```

Compilation d'un langage structuré vers FORTH. On se donne un petit langage de programmation impératif et structuré, avec des variables globales et des valeurs entières. Les expressions sont formées de constantes, variables et soustractions, avec la syntaxe abstraite suivante :

$$\begin{array}{rcll} e & ::= & n & \textit{constante entière} \\ & | & x & \textit{variable} \\ & | & e - e & \textit{soustraction} \end{array}$$

Les instructions comprennent des affectations, des conditionnelles, des boucles `while` et des séquences, avec la syntaxe abstraite suivante :

$$\begin{array}{rcll} s & ::= & x \leftarrow e & \textit{affectation} \\ & | & \texttt{if } e = 0 \texttt{ then } s \texttt{ else } s & \textit{conditionnelle} \\ & | & \texttt{while } e = 0 \texttt{ do } s & \textit{boucle} \\ & | & s\texttt{;}\ s & \textit{séquence} \\ & | & \texttt{putchar}(e) & \textit{afficher un caractère} \\ & | & \texttt{skip} & \textit{ne rien faire} \end{array}$$

On se propose de compiler ce petit langage vers le langage FORTH. Plus précisément, on cherche à définir

- pour toute expression e, un programme FORTH $\textit{compile}(e)$ dont l'exécution a pour effet de déposer la valeur de e au sommet de la pile ;
- pour toute instruction s, un programme FORTH $\textit{compile}(s)$ dont l'exécution produit les mêmes affichages que le programme s et laisse la pile inchangée (au final).

On suppose que toutes les variables apparaissant dans le programme initial ont été déclarées comme autant de variables au début du programme FORTH que l'on construit.

Question 9 Donner la définition de $compile(e)$ pour chacune des trois constructions et la définition de $compile(s)$ pour chacune des six constructions.

Correction : Pour les expressions :

$$\begin{array}{rcl} compile(n) & = & n \\ compile(x) & = & x\ \texttt{@} \\ compile(e_1 - e_2) & = & compile(e_1)\ compile(e_2)\ \texttt{-} \end{array}$$

Pour les instructions :

$$\begin{array}{rcl} compile(x \leftarrow e) & = & compile(e)\ x\ \texttt{!} \\ compile(\texttt{if}\ e = 0\ \texttt{then}\ s_1\ \texttt{else}\ s_2) & = & compile(e)\ \texttt{dup 0=if}\ compile(s_1)\ \texttt{endifif}\ compile(s_2)\ \texttt{endif} \\ compile(\texttt{while}\ e = 0\ \texttt{do}\ s) & = & compile(e)\ \texttt{0= if begin}\ compile(s)\ compile(e)\ \texttt{until endif} \\ compile(s_1\texttt{;}\ s_2) & = & compile(s_1)\ compile(s_2) \\ compile(\texttt{putchar}(e)) & = & compile(e)\ \texttt{emit} \\ compile(\texttt{skip}) & = & \end{array}$$

Estimation de la taille de pile. Dans cette dernière partie, on va chercher à estimer l'utilisation de la pile par un programme FORTH. Plus précisément, pour chaque morceau de code c, on va calculer cette estimation sous la forme d'un intervalle $[x, y]$ avec la signification « l'exécution du code c augmente la taille de pile d'une valeur comprise entre x et y ». La borne inférieure x est soit un entier, soit $-\infty$; la borne supérieure y est soit un entier, soit $+\infty$. Lorsque ce sont deux entiers, on a nécessairement $x \leq y$. Le problème n'étant pas décidable, on va se contenter d'une approximation correcte.

Question 10 Donner un code FORTH pour lequel la meilleure estimation est $[3, +\infty]$.

Correction :

```
dup dup begin dup dup until
```

Question 11 Donner l'intervalle pour chacune des primitives `dup`, `emit`, `drop`, `-`, `0=`, `@` et `!`. (On ne demande pas de traiter ici les primitives `begin` et `until`.)

Correction :

`!`	-2
`emit - drop`	-1
`0= @`	0
`dup`	1

Question 12 On suppose que les primitives `begin/until` sont bien parenthésées et bien imbriquées avec les `if/endif`. Dit autrement, on peut se donner la syntaxe abstraite OCaml suivante pour le code FORTH :

```
type code =
  | Wrd of string       (* un mot          *)
  | If  of code list    (* if ... endif    *)
  | Beg of code list    (* begin ... until *)
```

Donner alors le code d'une fonction OCaml `eval: code -> bounds` qui renvoie l'intervalle d'estimation pour un code donné. L'intervalle est représenté par le type OCaml

```
type bounds = int option * int option
```

où la valeur `None` représente une valeur infinie. On pourra supposer l'existence de deux tables de hachage globales contenant respectivement les variables et les fonctions connues. La table des fonctions donne, pour chaque fonction, son intervalle d'estimation (supposé déjà calculé).

Correction : On commence par se donner quelques fonctions pour manipuler les intervalles :

```
type bound = int option
type bounds = { lo: bound; hi: bound }
let exact x = { lo = Some x; hi = Some x }
let lift f = function None, _ | _, None -> None | Some x, Some y -> Some (f x y)
let (++) b1 b2 = { lo = lift (+) (b1.lo, b2.lo); hi = lift (+) (b1.hi, b2.hi) }
let union b1 b2 = { lo = lift min (b1.lo, b2.lo); hi = lift max (b1.hi, b2.hi) }
```

Puis on définit le calcul pour un mot (fonction `one`) et pour une liste de mots (fonction `code`) :

```
let rec one = function
  | Wrd x when Hashtbl.mem vars x -> exact 1
  | Wrd f when Hashtbl.mem funs f -> Hashtbl.find funs f
  | Wrd _ -> exact 1 (* entier *)
  | If c -> union (exact (-1)) (exact (-1) ++ code c)
  | Beg c -> (match exact (-1) ++ code c with (* -1 pour until *)
     | { lo = Some 0; hi = Some 0 } as i -> i
     | { lo = Some x; } when x > 0 -> { lo = Some x; hi = None }
     | { hi = Some x; } when x < 0 -> { lo = None; hi = Some x }
     | _ -> { lo = None; hi = None })

and code = function
  | []     -> exact 0
  | w :: c -> one w ++ code c
```

(La fonction `code` n'est rien d'autre qu'un `List.fold_left`.) On a supposé le programme bien formé, c'est-à-dire qu'un mot qui n'est ni une variable ni une fonction est forcément un entier.

⋆ ⋆
⋆

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.

Instruction	Description
`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

Énoncé en français.