École Polytechnique

INF564 : Compilation

examen 2019 (X2016)

Jean-Christophe Filliâtre

18 mars 2019

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

Dans tout ce sujet, on considère un tout petit fragment du langage C dont la syntaxe abstraite est donnée figure 1. Chaque fonction a exactement deux arguments, notés `x` et `y`. On note que ce fragment de C est un sous-ensemble strict de celui du projet. En particulier, les fonctions peuvent être récursives, mais pas mutuellement récursives : le corps d'une fonction f ne peut appeler que des fonctions définies préalablement ou la fonction f elle-même. On rappelle que la construction `if` du langage C teste si son argument est non nul. Voici un programme dans ce fragment :

```
int mul(int x, int y) {
  if (x) return y - (0 - mul(x - 1, y)); else return 0; }
int fact(int x, int y) {
  if (x) return fact(x - 1, mul(x, y)); else return y; }
```

Avec ce programme, un appel à `fact(`n`, `m`)`, pour deux entiers n et m, va calculer $n! \times m$ (en supposant une absence de débordement arithmétique). On peut supposer qu'une fonction `main` est ajoutée systématiquement à un tel programme, mais cet aspect-là ne nous intéresse pas ici.

1 Sémantique

Question 1 Indiquer ce que calcule un appel à `myst(`n`, `m`)`, pour deux entiers n et m, avec $n \geq 0$ et la définition suivante :

```
int myst(int x, int y) {
  if (x)
    if (x - 1) return myst(x - 2, myst(x - 1, y)); else return 1 - (0 - y);
  else
    return y;
}
```

Correction : Ce programme calcule $F_{\texttt{x}} + \texttt{y}$ où (F_n) est la suite de Fibonacci définie par

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

---

e	::=	**expression**
	| n	constante entière
	| `x`	la variable `x`
	| `y`	la variable `y`
	| e `-` e	soustraction
	| f`(`e`,` e`)`	appel de fonction
s	::=	**instruction**
	| `return` e`;`	retour de fonction
	| `if (`e`)` s `else` s	conditionnelle
d	::=	**définition de fonction**
	| `int` f`(int x, int y) {` s `}`	
p	::=	**programme**
	| $d \ldots d$	

FIGURE 1 – Syntaxe abstraite.

Question 2 Donner le code d'une fonction `lt` qui reçoit en arguments deux entiers n et m, en supposant $n \geq 0$ et $m \geq 0$, et renvoie un entier non nul si et seulement si $n < m$.

Correction : L'idée est de décrémenter `x` et `y` jusqu'à ce que l'un des deux atteigne 0.

```
int lt(int x, int y) {
  if (y)
    if (x) return lt(x - 1, y - 1); else return y;
  else
    return 0;
}
```

Sémantique opérationnelle à grands pas. On souhaite munir notre langage d'une sémantique opérationnelle à grands pas sous la forme de deux jugements

$$E, e \twoheadrightarrow n \qquad \text{et} \qquad E, s \twoheadrightarrow n$$

où E est une fonction donnant la valeur associée aux variables `x` et `y`. Le jugement $E, e \twoheadrightarrow n$ signifie « dans l'environnement E, l'évaluation de l'expression e termine, avec la valeur n ». Le jugement $E, s \twoheadrightarrow n$ signifie « dans l'environnement E, l'évaluation de l'instruction s termine, en aboutissant à une instruction `return` qui renvoie la valeur n ».

On suppose que les fonctions du programme sont données sous la forme d'un ensemble F de paires (f, s) où f est le nom d'une fonction et s l'instruction qui constitue le corps de cette fonction.

Question 3 Donner les règles d'inférence définissant les relations $E, e \twoheadrightarrow n$ et $E, s \twoheadrightarrow n$.

Correction :

$$\frac{}{E, n \twoheadrightarrow n} \qquad \frac{}{E, \texttt{x} \twoheadrightarrow E(\texttt{x})} \qquad \frac{}{E, \texttt{y} \twoheadrightarrow E(\texttt{y})} \qquad \frac{E, e_1 \twoheadrightarrow n_1 \quad E, e_2 \twoheadrightarrow n_2}{E, e_1 \texttt{ - } e_2 \twoheadrightarrow n_1 - n_2}$$

$$\frac{E, e_1 \twoheadrightarrow n_1 \quad E, e_2 \twoheadrightarrow n_2 \quad (f, s) \in F \qquad \{\texttt{x} \mapsto n_1,\ \texttt{y} \mapsto n_2\}, s \twoheadrightarrow n}{f\texttt{(}e_1\texttt{, }e_2\texttt{)} \twoheadrightarrow n}$$

$$\frac{E, e \twoheadrightarrow n}{E, \texttt{return } e\texttt{;} \twoheadrightarrow n} \qquad \frac{E, e \twoheadrightarrow n \quad n \neq 0 \quad E, s_1 \twoheadrightarrow n}{E, \texttt{if (}e\texttt{) } s_1 \texttt{ else } s_2 \twoheadrightarrow n} \qquad \frac{E, e \twoheadrightarrow 0 \quad E, s_2 \twoheadrightarrow n}{E, \texttt{if (}e\texttt{) } s_1 \texttt{ else } s_2 \twoheadrightarrow n}$$

Question 4 Donner un ensemble de fonctions F, un environnement E et une expression e pour lesquels il n'existe pas de valeur n telle que $E, e \twoheadrightarrow n$.

Correction : Il suffit de considérer une expression dont l'évaluation ne termine pas, par exemple

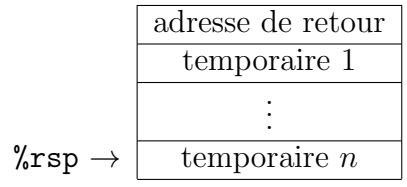
$$F = \{(f, \texttt{return } f\texttt{(x, y);})\} \qquad E \text{ quelconque} \qquad e = f\texttt{(42, 987)}.$$

---

2 Compilation vers x86-64

On se propose maintenant de compiler notre langage vers l'assembleur x86-64, *en utilisant exclusivement* les registres `%rdi`, `%rsi` et `%rax` pour les calculs et le registre `%rsp` pour la pile, et *exclusivement* les instructions qui sont données en annexe de ce sujet. On adopte le schéma de compilation suivant :

- Toutes les valeurs sont des entiers signés 64 bits.
- La valeur de `x` est passée dans `%rdi`, celle de `y` dans `%rsi` et la valeur renvoyée par la fonction dans `%rax`.
- Tous les registres sont *caller-saved*.
- Le tableau d'activation prend la forme suivante :



L'adresse de retour est celle déposée par l'instruction `call`. En dessous, on trouve la place allouée à n entiers 64 bits, pour toutes les valeurs temporaires qui ne peuvent pas être stockées dans les trois registres.

Question 5 Justifier qu'il n'est pas utile d'utiliser aussi le registre `%rbp` pour accéder au tableau d'activation (contrairement à ce qui a été vu en cours et fait en projet).

Correction : Comme toutes les fonctions n'ont que deux arguments, il n'y a jamais d'arguments à passer sur la pile. Par conséquent, il est possible de faire en sorte que la

valeur de `%rsp` ne varie jamais pendant l'exécution d'une fonction une fois que le tableau d'activation est alloué en début de fonction. On peut donc facilement repérer tous les temporaires de façon relative à `%rsp`.

Question 6 Donner un code x86-64 possible pour la fonction `myst` de la question 1, en optimisant l'appel terminal.

Correction :

```
myst:
        testq %rdi, %rdi
        jz exit0
        subq $1, %rdi
        jz exit1
        pushq %rdi
        call myst
        popq %rdi
        subq $1, %rdi
        movq %rax, %rsi
        jmp myst
exit1:  subq %rsi, %rdi
        movq $1, %rax
        subq %rdi, %rax
        ret
exit0:  movq %rsi, %rax
        ret
```

Compilateur optimisant. Pour écrire un compilateur optimisant pour notre langage, on suit l'architecture présentée en cours et réalisée en projet. On suppose que la sélection d'instructions et la traduction vers RTL ont déjà été réalisées (elles sont ici très simples) et on s'intéresse à l'étape ERTL. La figure 2 contient les instructions du langage ERTL que l'on considère ici, où n désigne une constante entière, r un pseudo-registre ou un registre physique et L une étiquette dans le graphe de flot de contrôle ERTL.

Question 7 Donner un code ERTL pour la fonction suivante.

```
int mul(int x, int y) {
  if (x) return y - (0 - mul(x - 1, y)); else return 0; }
```

On pourra commencer par construire un code RTL, pour le transformer ensuite en un code ERTL. On pourra omettre les étiquettes quand les instructions sont purement séquentielles.

Correction : Le point d'entrée est L_1.

```
L1: alloc_frame
    mov %rdi #1
```

$\texttt{mov}\ n\ r \to L$
$\texttt{mov}\ r\ r \to L$
$\texttt{sub}\ n\ r \to L$
$\texttt{sub}\ r\ r \to L$
$\texttt{jz}\ r \to L, L$
$\texttt{call}\ f \to L$
$\texttt{alloc\_frame} \to L$
$\texttt{delete\_frame} \to L$
$\texttt{return}$

FIGURE 2 – Instructions ERTL.

```
    mov %rsi #2
    jz  #1      -> Lz, Ln
Lz: mov 0 #3    -> Le
Ln: mov #1 #4
    sub 1 #4
    mov #2 #5
    mov #4 %rdi
    mov #5 %rsi
    call mul
    mov %rax #6
    mov 0 #7
    sub #6 #7
    mov #2 #3
    sub #7 #3
Le: mov #3 %rax
    delete_frame
    return
```

Question 8 Pour le code ERTL suivant, donner le résultat de l'analyse de durée de vie (variables vivantes en chaque point de programme), le graphe d'interférence, un coloriage possible de ce graphe et le code LTL donné par ce coloriage.

$$
\begin{aligned}
L_1 :&\ \texttt{alloc\_frame} \to L_2\\
L_2 :&\ \texttt{mov \%rdi \#1} \to L_3\\
L_3 :&\ \texttt{mov \%rsi \#2} \to L_4\\
L_4 :&\ \texttt{mov \#1 \#4} \to L_5\\
L_5 :&\ \texttt{mov \#2 \#5} \to L_6\\
L_6 :&\ \texttt{sub \#5 \#4} \to L_7\\
L_7 :&\ \texttt{mov \#1 \#3} \to L_8\\
L_8 :&\ \texttt{sub \#4 \#3} \to L_9\\
L_9 :&\ \texttt{mov \#3 \%rax} \to L_{10}\\
L_{10} :&\ \texttt{delete\_frame} \to L_{11}\\
L_{11} :&\ \texttt{return}
\end{aligned}
$$

Le point d'entrée du graphe est L_1.

Correction : Les variables vivantes sont indiquées dans la première colonne, entre les instructions. Dans la seconde colonne, on note `=` une arête de préférence et `/` une arête d'interférence. Noter que `#1`–`#4` est à la fois de préférence et d'interférence, donc d'interférence au final.

```
               pref(=)intf(/)          LTL
 %rdi %rsi
alloc_frame                            goto
 %rdi %rsi
mov %rdi #1    %rdi=#1 #1/%rsi         goto
   #1 %rsi
mov %rsi #2    #1/#2  #2=%rsi          goto
   #1 #2
mov #1 #4      #2/#4  #1=#4            mov %rdi, %rax
   #1 #2 #4
mov #2 #5      #5/#1 #5/#4 #2=#5       goto
   #1 #4 #5
sub #5 #4      #4/#1                   sub %rsi, %rax
   #1 #4
mov #1 #3      #3/#4  #1=#3            goto
   #3 #4
sub #4 #3                              sub %rax, %rdi
   #3
mov #3 %rax    #3=%rax                 mov %rdi, %rax
   rax
delete_frame                           goto
   rax
return                                 return
```

Il y a un coloriage possible avec uniquement des registres, à savoir

```
     #4 -> %rax   (on renonce à #3=%rax)
 #1, #3 -> %rdi
 #2, #5 -> %rsi
```

Le code LTL est indiqué dans la troisième colonne. (Il y a bien sûr d'autres coloriages possibles, notamment en utilisant la pile.)

Analyse de durée de vie plus précise. Pour calculer les variables vivantes en tout point du graphe de flot de contrôle ERTL, on se sert notamment des *définitions* (*def*) et des *utilisations* (*use*) de chaque instruction. Dans le cas où l'instruction à l'étiquette L est un appel (`call` f), on pose a priori

$$\begin{aligned} \mathit{def}(L) &= \{\texttt{\%rdi}, \texttt{\%rsi}, \texttt{\%rax}\} \\ \mathit{use}(L) &= \{\texttt{\%rdi}, \texttt{\%rsi}\} \end{aligned}$$

(On rappelle que les trois registres sont *caller-saved*.) C'est toujours correct mais c'est parfois une sur-approximation des registres que la fonction utilise ou définit effectivement.

Question 9 Donner un exemple de fonction f pour laquelle les ensembles de registres effectivement utilisés et définis par un appel à f sont tous les deux strictement plus petits que les ensembles ci-dessus.

Correction : Il suffit de prendre une fonction aussi simple que

```
int f(int x, int y) { return 42; }
```

dont un code ERTL peut être

```
alloc_frame
mov $42 #3
mov #3 %rax
delete_frame
return
```

On a alors $def = \{$`%rax`$\}$ et $use = \{\}$.
Note : on n'a pas inclus les deux instructions `mov %rdi #1` et `mov %rsi #2`, car elles vont disparaître à l'allocation de registres et du coup le code n'a effectivement pas besoin du contenu de ces deux registres. Voir la réponse à la question suivante à ce sujet.

Question 10 Expliquer comment on peut améliorer le calcul de *use* pour un appel à une fonction dans le cas général.

Correction : Il suffit de regarder si le code de f utilise la variable `x` (resp. `y`). Dans le cas contraire, on ne mentionne pas `%rdi` (resp. `%rsi`) dans $use(L)$. Note : Cela reste correct de toujours inclure les deux instructions `mov %rdi #1` et `mov %rsi #2` au début du code ERTL. Si `#1` (resp. `#2`) n'est jamais utilisé, l'allocation de registres va affecter `#1` à `%rdi` (resp. `#2` à `%rsi`) et l'instruction `mov` va tout simplement disparaître.

Question 11 Expliquer comment on peut améliorer le calcul de *def* pour un appel à une fonction f dans le cas général, pour une fonction f *non récursive*. (Indication : il faut aller jusqu'au bout de l'allocation de registres.)

Correction : Une fois l'allocation de registres de la fonction f terminée, on peut faire l'union de tous les *def* de toutes les instructions du code LTL obtenu. Si la fonction f n'appelle aucune autre fonction, ou uniquement des fonctions pour lesquelles on a trouvé préalablement des ensembles *def* strictement plus petits que $\{$`%rdi`, `%rsi`, `%rax`$\}$, on peut espérer que le *def* de la fonction f soit également plus petit que $\{$`%rdi`, `%rsi`, `%rax`$\}$.
C'est par exemple le cas avec la fonction qu'on a utilisée pour répondre à la question 9.

Question 12 Même question avec maintenant une fonction f récursive.

Correction : C'est plus compliqué, car il faut calculer un *point fixe* : pour calculer le *def* de la fonction, le faut connaître le *def* de cette même fonction. La difficulté est que le

calcul n'est pas ici monotone, car il dépend de l'allocation de registres. On ne peut donc pas faire un calcul de plus petit point fixe avec le théorème de Tarski.
En revanche, on peut procéder ainsi. On commence par supposer que

$$def = \{\,\texttt{\%rdi},\ \texttt{\%rsi},\ \texttt{\%rax}\,\}$$

ce qui permet l'analyse de durée, puis l'allocation de registres. On calcule alors l'ensemble *def* comme l'union de tous les *def* de toutes les instructions du code LTL obtenu (comme dans la question précédente). S'il est égal à $\{\,\texttt{\%rdi},\ \texttt{\%rsi},\ \texttt{\%rax}\,\}$, on s'arrête là. S'il est plus petit, on reprend l'analyse de durée de vie avec cet ensemble, puis l'allocation de registres et enfin le calcul de *def*. Et ainsi de suite. Cela converge nécessairement.

Question 13 Donner un exemple de fonction récursive, ainsi que son code ERTL, pour lequel l'ensemble *def* est strictement plus petit que $\{\,\texttt{\%rdi},\ \texttt{\%rsi},\ \texttt{\%rax}\,\}$ pour une allocation de registres que l'on précisera.

Correction : On prend une fonction qui n'utilise pas l'un de ces arguments, disons `y`, afin de ne pas devoir affecter les deux registres `%rdi` et `%rsi` au moment de l'appel récursif.

```
int f(int x, int y) { if (x) return f(x - 1, y); else return 42; }
```

Le code ERTL peut être le suivant :

```
    mov %rdi #1
    mov %rsi #2
    jz #1 -> L2, L3
L2: mov $42 #3 -> Le
L3: mov #1 #4          // #3 <- call f(#4, #5)
    sub $1 #4
    mov #2 #5
    mov #4 %rdi
    mov #5 %rsi
    call f
    mov %rax #3
Le: mov #3 %rax
    return
```

Une allocation de registres peut être

```
    jz %rdi -> L2, L3
L2: mov $42 %rax -> Le
L3: sub $1 %rdi
    call f
Le: return
```

Du coup, on obtient

$$def = \{\,\texttt{\%rdi},\ \texttt{\%rax}\,\}$$

Question 14 Expliquer pourquoi il convient de calculer les *def* / *use* de chaque fonction dans l'ordre où elles apparaissent dans le programme.

Correction : Si le *def* d'une fonction f diminue strictement, alors le *def* d'une fonction g qui l'appelle peut également diminuer (cf la réponse à la question 11), d'où l'intérêt de procéder dans l'ordre.

---

3 Sous-expressions communes

Pour compiler encore plus efficacement nos programmes, on propose l'idée suivante : si le corps d'une fonction fait apparaître au moins deux fois la même expression e, on cherchera à ne l'évaluer qu'une seule fois, *quand cela est possible*. On appelle cela le partage des sous-expressions communes.

Question 15 Identifier dans le programme suivant les sous-expressions dont le calcul pourrait être partagé :

```
int f(int x, int y) { return x - (0 - y); }
int g(int x, int y) { return g(f(x - 1, x - 1), f(x - 1, x - 1)); }
int h(int x, int y) {
  if (x - 1) return g(x - 1, y);
             else if (y - 1) return 2; else return g(x - 1, y - 1); }
```

Correction : Dans la fonction `g`, on peut partager
— le calcul des deux occurrences de `x - 1`,
— puis celui des deux occurrences de `f(x - 1, x - 1)`.
Dans la fonction `h`, on peut partager
— le calcul des trois occurrences de `x - 1`,
— le calcul des trois occurrences de `y - 1`,
— mais pas celui de `g(x - 1, y - 1)` car
  — son évaluation ne termine pas et l'évaluer avant les tests sur `x - 1` et `y - 1` changerait la sémantique du programme,
  — quand bien même son évaluation terminerait, on risquerait de changer la complexité du programme en évaluant inutilement `g(x - 1, y - 1)` dans le cas `x` = `y` = 1.

Question 16 Donner un critère permettant d'assurer que l'évaluation d'une expression ou d'une instruction termine. Le problème étant indécidable, on proposera une approximation correcte (mais nécessairement incomplète).

Correction : L'idée est de considérer les fonctions dans l'ordre du programme et de stocker pour chacune un booléen indiquant si son évaluation est assurée de terminer. Pour une fonction récursive, on met le booléen à faux. Sinon, on considère toutes les fonctions qu'elle appelle et on met le booléen à vrai si toutes ces fonctions ont leur booléen à vrai.

Pour une expression ou une instruction, il suffit alors de regarder tous les appels effectués et de vérifier que les booléens des fonctions concernées sont tous vrais.

Partage des sous-expressions communes. Pour mettre en œuvre le partage des sous-expressions communes, on étend la syntaxe abstraite de notre langage avec de nouvelles variables (notées $\#_i$ comme les pseudo-registres du code RTL).

$$
\begin{array}{rcll}
e & ::= & \ldots & \\
 & | & \#_i & \text{utilisation d'une variable de partage} \\
 & & & \\
s & ::= & \ldots & \\
 & | & \#_i \leftarrow e\texttt{;}\ s & \text{introduction d'une variable de partage}
\end{array}
$$

Ainsi, on peut écrire maintenant des programmes comme

```
int p(int x, int y) { #1 <- x - 1; if (#1) return p(#1, y); else return #1 - 1; }
```

pour factoriser ici le calcul de l'expression `x - 1`.

Question 17 Réécrire le programme de la question 15 en partageant le calcul des sous-expressions communes.

Correction :

```
int f(int x, int y) {
  return x - (0 - y); }
int g(int x, int y) {
  #1 <- x - 1;
  #2 <- f(#1, #1);
  return g(#2, #2); }
int h(int x, int y) {
  #1 <- x - 1;
  if #1 return g(#1, y);
  else #2 <- y - 1; if #2 return 2; else return g(#1, #2); }
```

Question 18 Proposer un algorithme pour effectuer le partage des sous-expressions communes, c'est-à-dire un algorithme pour transformer un programme dans la syntaxe abstraite originale (de la figure 1) vers un programme de la syntaxe abstraite étendue avec les variables de partage. La sémantique du programme doit être conservée. La complexité du programme ne doit pas être dégradée.

Indications : Écrire cette transformation à l'aide de dictionnaires associant à des expressions des variables de partage. La transformation prend en arguments un tel dictionnaire (les expressions à partager déjà connues) et un élément de programme à transformer (expression ou instruction). La transformation renvoie à la fois un nouveau dictionnaire et l'élément transformé.

On pourra supposer qu'on a répondu à la question 16 et qu'on dispose donc d'un critère pour assurer que l'évaluation d'une expression termine.

Correction : On ne fait rien pour les variables et les entiers :

```
T(M, n) = M, n
T(M, x) = M, x
T(M, y) = M, y
```

Pour la soustraction et l'appel, on passe séquentiellement dans les deux sous-expressions, puis on ajoute à la fin l'expression dans le dictionnaire si elle termine.

```
T(M, e1 - e2) =
  M, e1 <- T(M, e1)
  M, e2 <- T(M, e2)
  si e1-e2 termine alors Add(M, e1-e2) sinon M, e1-e2
T(M, f(e1, e2)) =
  M, e1 <- T(M, e1)
  M, e2 <- T(M, e2)
  si f(e1,e2) termine alors Add(M, f(e1,e2)) sinon M, f(e1,e2)
```

où la fonction `Add` trouve ou ajoute une nouvelle entrée dans le dictionnaire :

```
Add(M, e) =
  si M(e) = #i alors M, #i
  sinon M+{e -> #j}, #j avec #j frais
```

Pour les instructions, on procède ainsi :

```
T(M, return e) =
  M, e <- T(M, e)
  M, return e
T(M, if (e) s1 else s2) =
  M, e <- T(M, e)
  M1, s1 <- T(M, s1)
  M2, s2 <- T(M, s2)
  Inter(M1, M2), if (e) Close(M1\M2, s1) else Close(M2\M1, s2)
```

où `Inter` calcule l'intersection de deux dictionnaires, `\` la différence de deux dictionnaires et `Close` est l'opération suivante :

```
Close({e1 -> #1, ..., en -> #n}, s) =
  #1 <- e1; ...; #n <- en; s
```

Enfin, pour traduire le corps `s` d'une fonction, on part d'un dictionnaire vide et on appelle `Close` :

```
T(s) =
  M, s = T({}, s)
  Close(M, s)
```

Annexe : sous-ensemble de x86-64 considéré

Dans ce sujet, on se limite au fragment du jeu d'instructions x86-64 décrit ci-dessous et aux quatre registres `%rdi`, `%rsi`, `%rax` et `%rsp`. Dans ce qui suit, L désigne une étiquette, r désigne un registre, n une constante entière et o une opérande qui est soit un registre r, soit une opérande indirecte n`(`r`)`.

`mov` r`,` o	copie le registre r dans l'opérande o
`mov` o`,` r	copie l'opérande o dans le registre r
`mov` `$`n`,` r	charge la constante n dans le registre r
`sub` o`,` r	calcule $r - o$ et l'affecte à r
`sub` r`,` o	calcule $o - r$ et l'affecte à o
`sub` `$`n`,` o	calcule $o - n$ et l'affecte à o
`push` r	empile la valeur de r
`pop` r	dépile une valeur dans le registre r
`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