École Polytechnique

INF564 : Compilation

examen 2021 (X2018)

Jean-Christophe Filliâtre

15 mars 2021

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. Dans ce fragment, on a des entiers et des tableaux d'entiers alloués sur la pile en début de fonction, avec une taille connue statiquement. Pour manipuler ces tableaux, on dispose de pointeurs (des variables notées x désignant les arguments de la fonction ou les tableaux alloués dans la fonction) et d'une arithmétique de pointeurs. Si p est un pointeur vers la case d'indice i d'un tableau, alors `*`p désigne l'entier contenu dans cette case et p `+` j désigne un pointeur vers la case d'indice $i + j$ de ce même tableau. Lorsqu'on déclare un tableau `int` y`[`n`]`, alors y désigne un pointeur vers la case d'indice 0 de ce tableau. Une affectation `*`p `=` v donne la valeur v à la case de tableau désignée par le pointeur p.

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. Les fonctions ne renvoient pas de valeur. On rappelle que la construction `if` du langage C teste si son argument est non nul. Voici un exemple de programme dans ce fragment :

```
void fib(int *a, int *n) {
  if (*n) { *n = *n + -1; *a = *(a + -2) + *(a + -1); fib(a + 1, n); } else { }
}
void main() {
  int a[10], n[1]; *a = 0; *(a + 1) = 1; *n = 8; fib(a + 2, n); ...
}
```

Ce programme alloue deux tableaux sur la pile, un tableau `a` de taille 10 et un tableau `n` de taille 1, puis initialise le tableau `a` avec les dix premières valeurs de la suite de Fibonacci. Lors de l'appel à `fib`, ce sont uniquement deux pointeurs qui sont passés en arguments. On prendra le temps de bien lire et de bien comprendre cet exemple. En particulier, on note que l'addition `+` est surchargée : elle désigne à la fois l'addition de deux entiers et l'addition d'un pointeur et d'un entier.

1 Sémantique

Question 1 Donner le code d'une fonction `void mul(int *r, int *x, int *y)` qui a pour effet de mettre dans la case de tableau désignée par `r` le produit des deux entiers contenus dans les cases désignées par `x` et `y`. Les valeurs de `*x` et de `*y` ne doivent pas être modifiées (au final). On suppose que `r`, `x` et `y` désignent trois cases de tableau différentes et que la valeur de `*y` est positive ou nulle.

Correction :

e	::=	**expression**
	$\mid n$	constante entière
	$\mid x$	variable
	$\mid e$ `+` e	addition
	$\mid$ `*`e	déréférencement de pointeur
s	::=	**instruction**
	$\mid$ `*`e `=` e	affectation
	$\mid f$`(`$e, \ldots, e$`)`	appel de fonction
	$\mid$ `if (`e`)` s `else` s	conditionnelle
	$\mid$ `{` $s; \ldots; s$ `}`	bloc
d	::=	**définition de fonction**
	$\mid$ `void` f`(int *`$x, \ldots,$ `int *`x`)` `{ int` x`[`n`], ...,` x`[`n`]; ` $s; \ldots; s$ `}`	
p	::=	**programme**
	$\mid d \ldots d$	

FIGURE 1 – Syntaxe abstraite.

```
void mul(int *r, int *x, int *y) {
  if (*y) {
    *y = *y + -1;
    mul(r, x, y);
    *r = *r + *x;
    *y = *y + 1;
  } else {
    *r = 0;
  }
}
```

Sémantique opérationnelle des expressions. On souhaite munir notre langage d'une sémantique opérationnelle à grands pas sous la forme d'un premier jugement

$$E, M, e \twoheadrightarrow v$$

qui s'interprète comme « dans l'environnement E et la mémoire M, l'évaluation de l'expression e termine, avec la valeur v ». Ici, E est une fonction donnant les valeurs des variables et M est une fonction donnant les valeurs contenues dans les tableaux. Une valeur v peut être de deux sortes :

v	::=		**valeur**
	$\mid$	n	constante entière
	$\mid$	(a, n)	case de tableau

Une valeur de la forme (a, n) désigne la case d'indice n d'un tableau alloué à l'adresse a. La valeur d'une variable x est donnée par $E(x)$. La valeur contenue dans la case d'indice n d'un tableau alloué à l'adresse a est donnée par $M(a, n)$.

Question 2 Donner les règles d'inférence définissant la relation $E, M, e \twoheadrightarrow v$.

Correction :

$$\frac{}{E, M, n \twoheadrightarrow n} \qquad \frac{x \in dom(E)}{E, M, x \twoheadrightarrow E(x)} \qquad \frac{E, M, e \twoheadrightarrow (a, n) \quad (a, n) \in dom(M)}{E, M, *e \twoheadrightarrow M(a, n)}$$

$$\frac{E, M, e_1 \twoheadrightarrow n_1 \quad E, M, e_2 \twoheadrightarrow n_2}{E, M, e_1 \texttt{ + } e_2 \twoheadrightarrow n_1 + n_2} \qquad \frac{E, e_1 \twoheadrightarrow (a, n_1) \quad E, M, e_2 \twoheadrightarrow n_2}{E, M, e_1 \texttt{ + } e_2 \twoheadrightarrow (a, n_1 + n_2)}$$

Question 3 Donner un environnement E, une mémoire M et une expression e pour lesquels il n'existe pas de valeur v telle que $E, M, e \twoheadrightarrow v$.

Correction : Il y a plusieurs solutions, dont
— une variable qui n'est pas dans le domaine de E, c'est-à-dire $\emptyset, \emptyset, x \not\twoheadrightarrow v$;
— le déréférencement d'un pointeur qui n'est pas dans le domaine de M, c'est-à-dire $\{x \mapsto (a, 0)\}, \emptyset, *x \not\twoheadrightarrow v$;
— ou encore l'addition de deux pointeurs.

Sémantique opérationnelle des instructions. Pour les instructions, on se donne un second jugement

$$E, M, s \twoheadrightarrow M'$$

qui se lit comme « dans l'environnement E et la mémoire M, l'évaluation de l'instruction s termine, en aboutissant à une mémoire finale M' ». On donne la règle pour un appel de fonction :

$$\frac{\begin{array}{c}\texttt{void}\ f\texttt{(int *}x_1\texttt{,}\ldots\texttt{,int *}x_n\texttt{)\{ int }y_1\texttt{[}n_1\texttt{],}\ldots\texttt{,}y_m\texttt{[}n_m\texttt{]; }s_1\texttt{;}\ldots\texttt{;}s_k\ \texttt{\}} \\ E, M, e_i \twoheadrightarrow v_i \\ E' \stackrel{\text{def}}{=} \{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n,\quad y_1 \mapsto (a_1, 0), \ldots, y_m \mapsto (a_m, 0)\} \quad a_j \text{ fraîches} \\ M' \stackrel{\text{def}}{=} M + \{(a_j, l) \mapsto n_{j,l} \mid 0 \leq l < n_j\} \\ E', M', \{\ s_1; \ldots; s_k\ \} \twoheadrightarrow M''\end{array}}{E, M, f\texttt{(}e_1\texttt{,}\ldots\texttt{,}e_n\texttt{)} \twoheadrightarrow M''_{|M}}$$

Ici, les valeurs $n_{j,l}$ désignent des valeurs entières quelconques utilisées pour initialiser les cases des tableaux $y_1, \ldots, y_m$ alloués par la fonction. La notation $M''_{|M}$ désigne la fonction M'' restreinte au domaine de la fonction M. On exprime ainsi que les tableaux alloués par la fonction f ne sont plus visibles après l'appel à f.

Question 4 Donner les autres règles d'inférence définissant la relation $E, M, s \twoheadrightarrow M'$. (Qu'on se rassure, c'est *beaucoup* plus simple que pour la règle précédente.)

Correction :

$$\frac{E, M, e_1 \twoheadrightarrow (a, n_1) \quad E, M, e_2 \twoheadrightarrow n_2}{E, M, \texttt{*}e_1\ \texttt{=}\ e_2 \twoheadrightarrow M[(a, n_1) \leftarrow n_2]}$$

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

$$\frac{}{E, M, \{\ \} \twoheadrightarrow M} \qquad \frac{E, M, s_1 \twoheadrightarrow M_1 \quad E, M_1, \{\ s_2; \ldots; s_n\ \} \twoheadrightarrow M'}{E, M, \{\ s_1; s_2; \ldots; s_n\ \} \twoheadrightarrow M'}$$

---

2 Analyse syntaxique

On souhaite réaliser l'analyse syntaxique de notre petit langage avec un outil de type YACC (par exemple CUP pour Java ou Menhir pour OCaml). Pour les expressions, on écrit notamment un morceau de grammaire de la forme suivante (la syntaxe précise peut légèrement varier selon l'outil), où `expr` est l'unique non terminal :

```
expr:
  | IDENT          {...}
  | CONST          {...}
  | expr PLUS expr {...}
  | STAR expr      {...}
  | LPAR expr RPAR {...}
```

Les actions sémantiques ne nous intéressent pas ici et sont omises (`{...}`).

Question 5 Lorsque l'outil YACC est lancé sur le fichier ci-dessus, il déclare deux conflits de type lecture/réduction (`shift/reduce`). Les identifier, les expliquer, faire le choix de favoriser lecture ou réduction (notamment pour que le programme de la première page soit reconnu tel qu'on le souhaite) et expliquer comment indiquer ce choix à l'outil YACC. Les règles de grammaire ne doivent pas être modifiées.

Correction : Le premier conflit provient d'une expression telle que

$$*e_1 \texttt{ + } e_2$$

où on a le choix entre réduire $((*e_1)$ `+` $e_2)$ ou lire $(*(e_1$ `+` $e_2))$. Le second conflit provient d'une expression telle que

$$e_1 \texttt{ + } e_2 \texttt{ + } e_3$$

où on a le choix entre réduire $((e_1$ `+` $e_2)$ `+` $e_3)$ ou lire $(e_1$ `+` $(e_2$ `+` $e3))$.
Pour ce qui est du second conflit, on peut choisir arbitrairement entre lecture ou réduction, l'addition étant associative. Choisissons une associativité à gauche, c'est-à-dire de privilégier la réduction. Pour ce qui est du premier conflit, on choisit de favoriser également la réduction (comme la grammaire du langage C). On déclare donc ceci :

```
%left PLUS
%nonassoc STAR
```

La priorité plus élevée à `STAR` favorise la réduction dans le cas du premier conflit.

---

3 Typage

On se propose maintenant de réaliser le typage statique de notre langage, notamment sous la forme d'un jugement de typage

$$\Gamma \vdash e : \tau$$

qui signifie « dans l'environnement Γ, l'expression e est bien typée de type τ ». L'environnement Γ est l'ensemble des variables pouvant apparaître dans e (toutes de type `int*`). Un type est noté τ et peut être de deux sortes :

τ	::=		**type**
		`int`	type d'un entier
	|	`int*`	type d'un pointeur

Question 6 Donner les règles d'inférence définissant le jugement $\Gamma \vdash e : \tau$.

Correction :

$$\frac{}{\Gamma \vdash n : \texttt{int}} \qquad \frac{x \in \Gamma}{\Gamma \vdash x : \texttt{int*}} \qquad \frac{\Gamma \vdash e_1 : \texttt{int*}}{\Gamma \vdash *e_1 : \texttt{int}}$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 \texttt{ + } e_2 : \texttt{int}} \qquad \frac{\Gamma \vdash e_1 : \texttt{int*} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 \texttt{ + } e_2 : \texttt{int*}}$$

Question 7 Donner un exemple d'expression mal typée ne contenant aucune variable.

Correction : C'est facile : $*0$. (C'est effectivement une expression mal typée en C.)

Question 8 Montrer la propriété de préservation du typage, à savoir que si $\Gamma \vdash e : \texttt{int}$ et si $E, M, e \twoheadrightarrow v$ alors la valeur v est un entier et, de même, si $\Gamma \vdash e : \texttt{int*}$ et si $E, M, e \twoheadrightarrow v$ alors la valeur v est de la forme (a, n). On suppose que les valeurs données par E sont toutes de la forme (a, n) et que les valeurs données par M sont toutes de la forme n.

Correction : On procède par récurrence structurelle sur la dérivation de typage.
- $e = n$: alors le type est `int` et la valeur un entier.
- $e = x$: alors le type est `int*` et la valeur de la forme (a, n).
- $e = *e_1$: alors le type de e_1 est `int*` et par HR sa valeur est de la forme (a, n) et donc la valeur de $*e_1$ est un entier (hypothèse sur M).
- $e = e_1 \texttt{ + } e_2$: il y a deux cas, selon qu'il s'agit d'une addition de deux entiers ou d'une arithmétique de pointeurs. Dans les deux cas, on applique l'HR aux deux sous-expressions et on conclut facilement.

Question 9 A-t-on également la propriété de sûreté de typage, c'est-à-dire qu'une expression bien typée s'évalue toujours en une valeur ? Si oui, le démontrer. Sinon, donner un contre-exemple.

Correction : On n'a pas la propriété de sûreté du typage. Il suffit de considérer un accès en dehors des bornes d'un tableau. En effet, avec la règle donnée plus haut, seules les valeurs de la forme (a_j, l) avec $0 \leq l < n_j$ sont ajoutées au domaine de M. Dès lors, un accès en dehors des bornes du tableau, par exemple une expression comme $*(y \texttt{ + } -1)$ pour un tableau y, n'a pas de valeur.

Typage des instructions. Pour typer les instructions, on se donne un second jugement de typage, de la forme

$$\Delta, \Gamma \vdash s$$

qui signifie « dans les environnements Δ et Γ, l'instruction s est bien typée ». L'environnement Δ est la liste des fonctions connues, avec pour chacune son arité.

Question 10 Donner les règles d'inférence définissant le jugement $\Delta, \Gamma \vdash s$. On discutera notamment le choix du type à exiger pour l'expression e dans l'instruction `if (`e`)` s_1 `else` s_2.

Correction :

$$\frac{\Gamma \vdash e_1 : \texttt{int*} \quad \Gamma \vdash e_2 : \texttt{int}}{\Delta, \Gamma \vdash *e_1 \texttt{ = } e_2} \qquad \frac{f \text{ d'arité } n \in \Delta \quad \forall i.\, \Gamma \vdash e_i : \texttt{int*}}{\Delta, \Gamma \vdash f(e_1, \ldots, e_n)} \qquad \frac{\forall i.\, \Delta, \Gamma \vdash s_i}{\Delta, \Gamma \vdash \{\ s_1; \ldots; s_n\ \}}$$

Enfin, pour la conditionnelle, le langage C permet à priori une condition d'un type quelconque. Cependant, on peut noter que notre langage ne contient pas de pointeur nul. Dès

lors, on peut sans problème restreindre le type de la condition à `int`.

$$\frac{\Gamma \vdash e : \texttt{int} \quad \Delta, \Gamma \vdash s_1 \quad \Delta, \Gamma \vdash s_2}{\Delta, \Gamma \vdash \texttt{if (}e\texttt{)}\ s_1\ \texttt{else}\ s_2}$$

Question 11 Donner enfin la règle permettant de vérifier qu'une définition de fonction est bien typée.

Correction : Pour qu'une définition de fonction

$$\texttt{void}\ f\texttt{(int *}x_1, \ldots, \texttt{int *}x_1\texttt{)\{ int}\ y_1\texttt{[}n_1\texttt{]}, \ldots, y_m\texttt{[}n_m\texttt{];}\ s_1; \ldots; s_k\ \}$$

soit bien typée, il faut vérifier que le bloc $\{\ s_1; \ldots; s_k\ \}$ est bien typé dans l'ensemble Γ défini comme

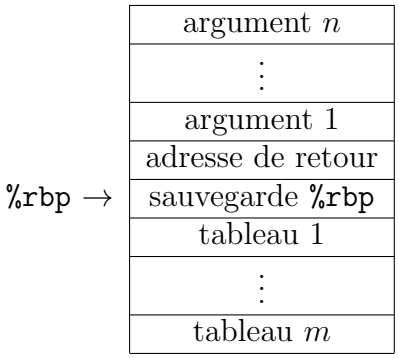
$$\Gamma \stackrel{\text{def}}{=} \{x_1, \ldots, x_n, y_1, \ldots, y_m\}$$

c'est-à-dire réunissant toutes les variables de la fonction, et en ajoutant la fonction f à Δ pour permettre une définition récursive.

---

4 Compilation vers x86-64

On se propose maintenant de compiler notre langage vers l'assembleur x86-64. (Un aide-mémoire est donné en annexe.) On adopte le schéma de compilation suivant. Toutes les valeurs sont sur 64 bits (qu'il s'agisse d'entiers ou de pointeurs). Tous les arguments sont passés sur la pile. Ce sont tous des pointeurs, occupant chacun 8 octets. Tous les tableaux sont alloués sur la pile, de façon contiguë, dans la partie basse du tableau d'activation. Le tableau d'activation prend donc la forme ci-contre. L'adresse de retour est celle déposée par l'instruction `call`. On suppose qu'on a calculé pour chaque variable, c'est-à-dire pour chaque argument et chaque tableau de la fonction, sa position par rapport à `%rbp`. On note `ofs`(x) ce décalage pour une variable x. Pour un tableau x de taille n on a donc $8n$ octets pour ses éléments, à partir de l'adresse `%rbp` + `ofs`(x).



Question 12 Donner un code x86-64 possible pour la fonction `fib` de la page 1, en optimisant l'appel terminal.

Correction : On a `ofs`(`a`) = +16 et `ofs`(`n`) = +24.

```
fib:
        pushq %rbp
        movq  %rsp, %rbp
1:      movq  24(%rbp), %rdi
        movq  (%rdi), %rax
        testq %rax, %rax      # if (*n) {
```

```
        jz    2f
        addq  $-1, %rax       # *n = *n + -1;
        movq  %rax, (%rdi)
        movq  16(%rbp), %rdi  # a
        movq  -16(%rdi), %rax # *(a + -2)
        addq  -8(%rdi), %rax  # *(a + -1)
        movq  %rax, (%rdi)    # *a = ...
        addq  $8, %rdi
        movq  %rdi, 16(%rbp)  # appel terminal optimisé
        jmp   1b
2:      popq  %rbp
        ret
```

Compilation d'une expression. 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 13 Donner la définition de $C(e)$ pour les différentes expressions du langage, sans chercher à utiliser efficacement les registres (ce sera l'objet de la partie suivante). On utilisera donc la pile pour stocker les résultats intermédiaires et uniquement un second registre si besoin (par exemple `%rcx`). On prendra soin de distinguer l'addition de deux entiers et l'addition d'un pointeur et d'un entier.

Correction : On se sert donc la pile pour stocker les résultats intermédiaires.

```
C(n) = movq $n, %rax
C(x) = movq n(%rbp), %rax   si n = ofs(x) > 0
     = leaq n(%rbp), %rax   si n = ofs(x) < 0
C(*e)= C(e)
       movq (%rax), %rax
C(e1+e2) =
       C(e2)
       shl   $3, %rax     # seulement dans le cas pointeur+entier
       pushq %rax
       C(e1)
       popq  %rcx
       addq  %rcx, %rax
```

Compilation d'une instruction. La compilation d'une instruction s est un code assembleur noté $C(s)$. On rappelle que les instructions ne renvoient aucune valeur (il n'y a pas d'instruction `return`).

Question 14 Donner la définition de $C(s)$ pour les différentes instructions du langage.

Correction :

```
C(*e1 = e2) =
  C(e1)
  pushq %rax
  C(e2)
  popq  %rcx
  movq  %rax, (%rcx)

C(f(e1,...,en)) =
  C(en)
  pushq %rax
  ...
  C(e1)
  pushq %rax
  call  f
  addq  $8n, %rsp  # dépiler les arguments

C(if (e) s1 else s2) =
  C(e)
  testq %rax, %rax
  jz    1f
  C(s1)
  jmp   2f
1:C(s2)
2:

C({s1;...;sn}) =
  C(s1)
  ...
  C(sn)
```

---

5 Allocation de registres

Dans cette dernière partie, nous allons essayer de mieux utiliser les registres pour la compilation d'une expression, mais avec une approche différente de celle du projet. Étant donnée une expression e et un registre r, on cherche à produire un code assembleur $C(e, r)$ qui place la valeur de e dans le registre r. Pour optimiser l'utilisation des registres, nous allons exploiter le fait que, dans la compilation d'une expression de la forme e_1 + e_2, on a le choix de l'ordre d'évaluation. En revanche, on ne s'autorise pas à effectuer des simplifications des expressions (telle que $x + 1 + -1 = x$), ni à exploiter l'associativité de l'addition.

Question 15 Donner, pour chacune des expressions suivantes, le nombre minimal de registres nécessaires à son calcul (en incluant le registre cible) :

1. $((*x + *y) + *z) + *t$
2. $(*x + *y) + (*z + *t)$
3. $*x + ((*y + *z) + *t)$

Correction : L'idée est d'effectuer les calculs en commençant toujours par la sous-expression nécessitant le plus de registres.

1. 2, par exemple $(((x1 + y2)1 + z2)1 + t2)1$
2. 3, par exemple $((x1 + y2)1 + (z2 + t3)2)1$
3. 2, par exemple $(x2 + ((y1 + z2)1 + t2)1)1$

Question 16 On note $R(e)$ le nombre minimal de registres nécessaires au calcul de l'expression e (en incluant le registre cible). Définir $R(e)$ par récurrence sur l'expression e.

Correction : On traduit l'idée consistant à commencer systématiquement par la sous-expression nécessitant le plus de registres.

$$\begin{array}{rcll} R(n) & = & 1 & \\ R(x) & = & 1 & \\ R(*e) & = & R(e) & \\ R(e_1 + e_2) & = & 1 + R(e_1) & \text{si } R(e_1) = R(e_2) \\ & = & \max(R(e_1), R(e_2)) & \text{sinon} \end{array}$$

Question 17 On définit la taille $T(e)$ d'une expression e comme le nombre d'additions qu'elle contient, c'est-à-dire $T(n) = T(x) = 0$, $T(*e) = T(e)$ et $T(e_1 + e_2) = 1 + T(e_1) + T(e_2)$. Minorer $T(e)$ par une expression impliquant $R(e)$. (On demande une preuve de cette inégalité.) En déduire la taille minimale d'une expression dont ne peut pas calculer la valeur avec seulement 5 registres.

Correction : On montre par récurrence sur e que

$$T(e) \geq 2^{R(e)-1} - 1.$$

En effet, si $e = n$ ou $e = x$ alors $T(e) = 0 \geq 2^{1-1} - 1 = 0$. Si $e = *e_1$, alors $T(e) = T(e_1)$ et $R(e) = R(e_1)$ et l'hypothèse de récurrence permet de conclure. Enfin, si $e = e_1 + e_2$, on distingue deux cas :

— si $R(e_1) = R(e_2)$ alors

$$\begin{array}{rcll} T(e) & = & 1 + T(e_1) + T(e_2) & \\ & \geq & 1 + 2^{R(e_1)-1} - 1 + 2^{R(e_2)-1} - 1 & \text{par H.R.} \\ & = & 2^{R(e)-1} - 1. & \end{array}$$

— si $R(e_1) \neq R(e_2)$, supposons sans perte de généralité que $R(e_1) > R(e_2)$. Alors

$$\begin{array}{rcll} T(e) & = & 1 + T(e_1) + T(e_2) & \\ & \geq & 1 + 2^{R(e_1)-1} - 1 + 2^{R(e_2)-1} - 1 & \text{par H.R.} \\ & > & 2^{R(e_1)-1} - 1 & \text{car } 2^{R(e_2)-1} > 0 \\ & = & 2^{R(e)-1} - 1. & \end{array}$$

On en déduit que s'il faut au moins 6 registres, c'est-à-dire $n \geq 6$, la taille est au moins égale à 31.

Compilation. On suppose qu'on dispose d'un nombre $K \geq 2$ de registres, notés $r_1, \ldots, r_K$. On cherche à compiler une expression en utilisant les registres au maximum, et la pile sinon.

Question 18 Définir la compilation d'une expression e. Le résultat doit être un code assembleur qui place la valeur de e dans le registre r_1 au final. La pile ne doit pas être utilisée lorsque $R(e) \leq K$.

Correction : On commence par écrire une fonction plus générale, prenant en arguments une expression e et un registre k, plaçant la valeur de e dans r_k. L'idée est que les registres inférieurs à k ne sont plus disponibles. On maintient l'invariant $1 \leq k \wedge k + \min(2, R(e)) \leq K + 1$.

$$
\begin{array}{rcl}
C(n,k) & = & \texttt{movq}\ \$n, r_k \\
C(x,k) & = & \texttt{movq}\ \$n(\texttt{\%rbp}), r_k \text{ si argument} \\
C(x,k) & = & \texttt{leaq}\ \$n(\texttt{\%rbp}), r_k \text{ sinon} \\
C(*e,k) & = & C(e,k);\ \texttt{movq}\ (r_k), r_k \\
C(e_1+e_2,k) & = & \text{si } R(e_1) \geq R(e_2) \text{ alors} \\
 & & \quad \text{si } k+1+\min(2,R(e_2)) \leq K+1 \text{ alors} \\
 & & \quad\quad C(e_1,k);\ C(e_2,k+1);\ \texttt{addq}\ r_{k+1}, r_k \\
 & & \quad \text{sinon} \\
 & & \quad\quad C(e_1,k);\ \texttt{pushq}\ r_k;\ C(e_2,k);\ \texttt{popq}\ r_{k+1};\ \texttt{addq}\ r_{k+1}, r_k \\
 & & \text{sinon} \\
 & & \quad C(e_2+e_1,k)
\end{array}
$$

La fonction demandée est alors $C(e, 1)$.

Annexe : aide-mémoire x86-64

Dans ce qui suit, L désigne une étiquette, r 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
`add` o`,` r	calcule $r + o$ et l'affecte à r
`add` r`,` o	calcule $o + r$ et l'affecte à o
`add` `$`n`,` o	calcule $o + n$ et l'affecte à o
`shl` `$`n`,` r	décale les bits de r vers la gauche, n fois
`lea` n`(`r_1`),` r_2	affecte à r_2 l'adresse représentée par n`(`r_1`)`, c'est-à-dire $r_1 + n$
`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

On pourra utiliser librement toute autre instruction x86-64. Mais les instructions ci-dessus suffisent pour répondre aux questions de ce sujet.