

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.

$e ::=$	<ul style="list-style-type: none"> <li>  <math>n</math></li> <li>  <math>x</math></li> <li>  <math>e + e</math></li> <li>  <math>*e</math></li> </ul>	<b>expression</b> constante entière variable addition déréférencement de pointeur
$s ::=$	<ul style="list-style-type: none"> <li>  <math>*e = e</math></li> <li>  <math>f(e, \dots, e)</math></li> <li>  <math>\text{if } (e) s \text{ else } s</math></li> <li>  <math>\{ s; \dots; s \}</math></li> </ul>	<b>instruction</b> affectation appel de fonction conditionnelle bloc
$d ::=$	<ul style="list-style-type: none"> <li>  <math>\text{void } f(\text{int } *x, \dots, \text{int } *x)</math></li> <li>  <math>\{ \text{int } x[n], \dots, x[n]; s; \dots; s \}</math></li> </ul>	<b>définition de fonction</b>
$p ::=$	<ul style="list-style-type: none"> <li>  <math>d \dots d</math></li> </ul>	<b>programme</b>

FIGURE 1 – Syntaxe abstraite.

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

$$E, M, e \rightarrow v$$

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 :

$v ::=$	<ul style="list-style-type: none"> <li>  <math>n</math></li> <li>  <math>(a, n)</math></li> </ul>	<b>valeur</b> constante entière 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  $E, M, e \rightarrow v$ .

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

**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 :

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

Ici, les valeurs  $n_{j,l}$  désignent des valeurs entières quelconques utilisées pour initialiser les cases des tableaux  $y_1, \dots, y_m$  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.)

## 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 ( $\{\dots\}$ ).

**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.

## 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  $\Gamma$ , l'expression  $e$  est bien typée de type  $\tau$  ». L'environnement  $\Gamma$  est l'ensemble des variables pouvant apparaître dans  $e$  (toutes de type `int*`). Un type est noté  $\tau$  et peut être de deux sortes :

$$\tau ::= \begin{array}{l} \text{type} \\ | \text{int} \quad \text{type d'un entier} \\ | \text{int*} \quad \text{type d'un pointeur} \end{array}$$

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

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

**Question 8** Montrer la propriété de préservation du typage, à savoir que si  $\Gamma \vdash e : \text{int}$  et si  $E, M, e \rightarrow v$  alors la valeur  $v$  est un entier et, de même, si  $\Gamma \vdash e : \text{int*}$  et si  $E, M, e \rightarrow v$  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  $n$ .

**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.

**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  $\Delta$  et  $\Gamma$ , l'instruction  $s$  est bien typée ». L'environnement  $\Delta$  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) s1 else s2`.

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

## 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)`.

argument $n$
⋮
argument 1
adresse de retour
sauvegarde <code>%rbp</code>
tableau 1
⋮
tableau $m$

`%rbp` →

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

**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.

**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.

## 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.  $*x + ((*y + *z) + *t)$

**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  $e$ .

**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.

**Compilation.** On suppose qu'on dispose d'un nombre  $K \geq 2$  de registres, notés  $r_1, \dots, r_K$ . 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  $R(e) \leq K$ .

## 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  $n(r)$ .

<code>mov <math>r, o</math></code>	copie le registre $r$ dans l'opérande $o$
<code>mov <math>o, r</math></code>	copie l'opérande $o$ dans le registre $r$
<code>mov <math>\\$n, r</math></code>	charge la constante $n$ dans le registre $r$
<code>add <math>o, r</math></code>	calcule $r + o$ et l'affecte à $r$
<code>add <math>r, o</math></code>	calcule $o + r$ et l'affecte à $o$
<code>add <math>\\$n, o</math></code>	calcule $o + n$ et l'affecte à $o$
<code>shl <math>\\$n, r</math></code>	décale les bits de $r$ vers la gauche, $n$ fois
<code>lea <math>n(r_1), r_2</math></code>	affecte à $r_2$ l'adresse représentée par $n(r_1)$ , c'est-à-dire $r_1 + n$
<code>push <math>r</math></code>	empile la valeur de $r$
<code>pop <math>r</math></code>	dépile une valeur dans le registre $r$
<code>test <math>r_2, r_1</math></code>	positionne les drapeaux en fonction de la valeur de $r_1$ ET $r_2$
<code>jz <math>L</math></code>	saute à l'adresse désignée par l'étiquette $L$ si les drapeaux signalent un résultat nul
<code>jmp <math>L</math></code>	saute à l'adresse désignée par l'étiquette $L$
<code>call <math>L</math></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 pourra utiliser librement toute autre instruction x86-64. Mais les instructions ci-dessus suffisent pour répondre aux questions de ce sujet.