

École Normale Supérieure
Langages de programmation et compilation
examen 2024–2025

Jean-Christophe Filliâtre
24 janvier 2025 — 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.

Sauf mention explicite du contraire, les réponses doivent être justifiées.

Les figures 1–4 sont regroupées en fin de sujet, page 7. Suggestion : détacher la dernière feuille.

Dans ce sujet, on considère un petit langage impératif appelé `WHILE` dont la syntaxe abstraite est donnée figure 1. Un programme se réduit à une instruction. Une instruction (notée s) est un affectation, un affichage avec `print`, une conditionnelle, une boucle `while` ou un bloc (une séquence d'instructions, possiblement vide). Une variable (notée x) contient un entier et les affectations sont limitées à des constantes et des soustractions de variables. Voici un exemple de programme :

```
{ a=0  b=1
  while n<>0 do { t=0  t=t-b  b=a-t  a=b-a  t=1  n=n-t }
  print a }
```

Sémantique. On munit `WHILE` d'une sémantique opérationnelle à petits pas, sous la forme d'une relation binaire notée \rightarrow entre deux configurations. Une configuration, notée $\langle E, s \rangle$, est la donnée d'un environnement E et d'une instruction s . Un environnement est une fonction partielle des variables vers les entiers. Son domaine, c'est-à-dire l'ensemble des variables pour lesquelles E est définie, est noté $\text{dom}(E)$. La relation $\langle E, s \rangle \rightarrow \langle E', s' \rangle$ se lit donc comme « dans l'environnement E , l'instruction s exécute un pas de calcul avec succès et conduit à l'environnement E' et à l'instruction s' ».

La figure 2 définit la relation \rightarrow par un ensemble de règles d'inférence. Dans ces règles, la notation $E \oplus \{x \mapsto n\}$ désigne la fonction E' définie par

$$\begin{aligned} E'(x) &= n, \\ E'(y) &= E(y) \text{ pour } y \in \text{dom}(E) \text{ et } y \neq x. \end{aligned}$$

En particulier, on a $\text{dom}(E') = \text{dom}(E) \cup \{x\}$. On notera avec soin comment les règles définissant \rightarrow manipulent les environnements. Si `fib` désigne le programme donné ci-dessus en exemple, on a l'exécution

$$\langle \{n \mapsto 10\}, \text{fib} \rangle \rightarrow^* \langle \{a \mapsto 55, n \mapsto 0, b \mapsto 89, t \mapsto 1, \}, \{ \} \rangle$$

(et le programme a affiché 55).

Une configuration C est dite irréductible s'il n'existe pas de configuration C' telle que $C \rightarrow C'$. On dit que l'exécution d'une configuration *bloque* si elle conduit, après un certain nombre de pas d'exécution, à une configuration irréductible $\langle E, s \rangle$ avec $s \neq \{ \}$. Un exemple trivial de configuration qui bloque est $\langle \emptyset, \text{print } x \rangle$ car la variable x n'est pas définie dans l'environnement vide. Une exécution qui ne bloque pas termine sur l'instruction $\{ \}$ ou bien ne termine pas.

Question 1 Donner la séquence des étapes d'exécution du programme

```
{ x=1 if x=0 then y=1 else z=2 print y }
```

dans l'environnement initial vide.

Question 2 Proposer un programme qui, dans un environnement E définissant au moins les variables a et b , avec $E(b) \geq 0$, termine après avoir affiché la valeur de $E(a) \times E(b)$. Comment relâcher l'hypothèse $E(b) \geq 0$? (On ne demande pas le code cette fois, mais uniquement l'idée.)

Analyse syntaxique. On souhaite réaliser l'analyse syntaxique du langage WHILE.

Question 3 Une grammaire pour un sous-ensemble du langage est la suivante :

$$\begin{aligned} S &::= \text{id} = \text{int} \\ &| \{ L \} \\ L &::= \epsilon \\ &| S L \end{aligned}$$

Montrer que cette grammaire est LL(1).

Question 4 Donner une grammaire pour *tout* le langage WHILE qui soit LL(1). On ne demande pas de justification.

Question 5 Discuter le problème de la constante 0 qui apparaît dans la syntaxe des constructions `if` et `while` mais également dans une affectation de la forme $x = 0$. On indiquera précisément comment le prendre en compte au niveau de l'analyse lexicale et au niveau de l'analyse syntaxique. On essaiera de proposer deux solutions différentes.

Analyse statique des utilisations et définitions. Pour qu'un programme ne bloque pas, il suffit que toute variable utilisée par une instruction (à droite d'une affectation ou en argument d'une instruction `print`, `if` ou `while`) soit présente dans l'environnement au moment de l'exécution de cette instruction. Cette variable peut être présente dans l'environnement depuis le début de l'exécution ou bien avoir été introduite dans l'environnement par une instruction d'affectation précédemment exécutée. Ainsi, le programme

```
{ if x = 0 then y = 1 else y = 2
  print y }
```

s'exécute correctement dans un environnement initial qui définit uniquement x .

On se propose de déterminer statiquement, pour une instruction s , un ensemble de variables noté $use(s)$ suffisant pour sa bonne exécution, dans le sens suivant : pour tout environnement E tel que $use(s) \subseteq \text{dom}(E)$, l'exécution de $\langle E, s \rangle$ ne bloque pas. Bien entendu, il suffirait de prendre pour $use(s)$ l'ensemble des variables qui apparaissent dans s pour qu'une exécution soit toujours possible, mais c'est là une solution très grossière. Pour le programme ci-dessus, par exemple, l'ensemble $\{x\}$ est suffisant.

Pour réaliser une analyse plus fine de $use(s)$, on va calculer en même temps un second ensemble de variables, noté $def(s)$, correspondant à des variables nécessairement définies par l'exécution de s , dans le sens suivant : pour toute exécution $\langle E, s \rangle \rightarrow^* \langle E', \{ \} \rangle$, alors on a $def(s) \subseteq \text{dom}(E')$. Pour le

programme ci-dessus, par exemple, on a $def(s) = \{y\}$ car toute exécution va donner une valeur à la variable y .

On se propose de calculer les ensembles $def(s)$ et $use(s)$ par récurrence sur la structure de s . Voici comment effectuer le calcul pour les trois premières instructions :

	$def(s)$	$use(s)$
$x = n$	$\{x\}$	\emptyset
$x_1 = x_2 - x_3$	$\{x_1\}$	$\{x_2, x_3\}$
print x	\emptyset	$\{x\}$

Question 6 Compléter ce tableau en proposant un calcul de $def(s)$ et $use(s)$ pour les trois autres instructions de WHILE : conditionnelle, boucle et bloc.

Question 7 Montrer la correction de votre définition de $use(s)$. On s'attachera à énoncer clairement la ou les propriétés prouvées et à indiquer clairement sur quoi portent les récurrences, le cas échéant.

Question 8 Un algorithme (le vôtre ou tout autre) peut-il toujours donner des ensembles $def(s)$ et $use(s)$ minimaux pour l'inclusion ? Si oui, justifier. Si non, expliquer.

Langage RTL. On considère maintenant un langage de type RTL (pour *Register Transfer Language*) dont la syntaxe abstraite est donnée figure 3. Un programme RTL est un ensemble fini de *blocs de base*. Un bloc de base b est la donnée d'une étiquette L , d'une séquence d'instructions et d'un saut. Une instruction i est une affectation ou un affichage, comme dans le langage WHILE. Un saut j est soit la terminaison du programme (**halt**), un branchement conditionnel (**ifz** x L_1 L_2) ou un branchement inconditionnel (**goto** L). Un programme RTL est bien formé si

- toute étiquette mentionnée par **ifz** ou **goto** correspond bien à un bloc du programme ;
- il n'y a pas de saut **goto** L vers un bloc n'ayant qu'un seul prédécesseur (c'est-à-dire un bloc autre que le premier bloc et dont la seule référence est ce saut **goto** L).

Voici un exemple de programme RTL bien formé avec quatre blocs de base :

```
L0: n=10  r=0  one=1  one=r-one  goto L1
L1: t=r-n  ifz t L2 L3
L2: halt
L3: r=r-one  print r  goto L1
```

(R_0)

On note en particulier que le bloc L1 a deux prédécesseurs.

L'exécution d'un programme RTL se fait dans un environnement E (du même type que pour WHILE) et démarre à son premier bloc (L0 dans l'exemple ci-dessus). L'exécution d'un bloc est l'exécution de ses instructions, en séquence, avec la même sémantique que pour WHILE. Si l'exécution des instructions bloque, le programme RTL bloque également. Sinon, on obtient un nouvel environnement E' puis on effectue le saut. Le saut **halt** termine le programme RTL. Le saut conditionnel **ifz** x L_1 L_2 poursuit l'exécution sur le bloc L_1 si $E'(x) = 0$ et sur le bloc L_2 sinon. Le saut inconditionnel **goto** L poursuit l'exécution sur le bloc L . En cas de saut, l'exécution se poursuit avec l'environnement E' . Ainsi, le programme ci-dessus imprime les entiers de 1 à 10 puis termine.

Question 9 Quel est l'effet du programme suivant

```
L0: zero=0  one=1  one=zero-one  u=1  r=0  goto L1
L1: t=r-n  ifz t L6 L2
L2: v=u-zero  s=0  goto L3
L3: t=s-r  ifz t L5 L4
L4: t=zero-v  u=u-t  s=s-one  goto L3
L5: r=r-one  goto L1
L6: print u  halt
```

dans un environnement qui définit une variable n avec une valeur positive ou nulle ?

Question 10 Proposer des types OCaml pour la syntaxe abstraite d'un programme RTL. Proposer un code OCaml pour exécuter un programme RTL (à partir d'un environnement vide).

Question 11 On souhaite calculer l'ensemble *use* d'un programme RTL, avec le même sens que pour un programme WHILE (*i.e.*, l'exécution ne bloque pas sur un environnement qui définit au moins les variables de l'ensemble *use*). Proposer un algorithme pour calculer cet ensemble.

Question 12 Proposer un algorithme pour compiler le langage WHILE vers le langage RTL. On veillera à ce que la bonne formation du programme RTL soit assurée par construction.

Compilation vers l'assembleur x86-64. On se propose de compiler le langage RTL vers l'assembleur x86-64. (Un aide-mémoire est donné en annexe.) On fait l'hypothèse que les entiers de notre langage sont limités à des entiers 64 bits signés. On suppose donnée une fonction assembleur `print` obéissant aux conventions d'appel (son argument est dans `%rdi` et elle écrase potentiellement tout registre *caller-saved*) mais n'exigeant pas l'alignement de la pile lors de son appel.

Comme notre langage RTL est très simple, on peut faire une allocation de registres par coloration de graphe en construisant directement un graphe d'interférence à partir du code RTL. Pour les deux questions suivantes, on prend en exemple le programme RTL suivant :

```
A: zero=0  s=0  one=1  n=10  goto B
B: t=zero-n  s=s-t  n=n-one  ifz n H B
H: print s  halt
```

(R_1)

Question 13 Donner le graphe d'interférence du programme (R_1). On essayera de proposer une ou plusieurs arêtes de préférence, en les justifiant.

Question 14 Proposer un code assembleur pour le programme (R_1).

Question 15 Proposer un code assembleur pour le programme (R_0) de la page 4.

Question 16 De manière générale, quelles arêtes de préférence peut-on induire d'un programme RTL ?

Question 17 Proposer un schéma de compilation général pour le langage RTL, c'est-à-dire expliquer comment chacune de ses constructions (instruction ou saut) peut être compilée vers x86-64.

Langage SSA. Pour optimiser nos programmes RTL, on va temporairement les mettre en forme SSA (*Static Single Assignment*), plus adaptée aux optimisations, avant de revenir dans le langage RTL. Dit autrement, on adopte une architecture de compilateur de la forme suivante :

$$\text{WHILE} \rightarrow \text{RTL} \rightarrow \text{SSA} \xrightarrow{\text{optim}} \text{SSA} \rightarrow \text{RTL} \rightarrow \text{x86-64}$$

Le langage SSA est une variante du langage RTL. Sa syntaxe abstraite est donné figure 4. Comme dans RTL, un programme est un ensemble de blocs (avec les mêmes conditions de bonne formation). Les instructions et les sauts sont les mêmes que dans RTL. À la différence de RTL, cependant,

- un bloc peut commencer par une séquence (possiblement vide) de « fonctions ϕ » de la forme $x = \text{phi}(x_1, \dots, x_n)$. Une telle instruction affecte à la variable x la valeur de la dernière variable x_i qui a été affectée.
- toute variable n'est affectée qu'à *un seul endroit du programme* (c'est là le sens de SSA), que ce soit par une instruction ou par une fonction ϕ .

Voici un exemple de programme SSA dont l'exécution est identique à celle du programme (R_0) :

```
L0: n=10 r1=0 one1=1 one2=r1-one1 goto L1
L1: r2=phi(r1,r3) t=r2-n ifz t L2 L3
L2: halt
L3: r3=r2-one2 print r3 goto L1
```

Au début du bloc L1, la fonction ϕ donne à la variable $r2$ la valeur de $r1$ si on vient du bloc L0 et la valeur de $r3$ si on vient du bloc L3. Pour être bien formé, un programme SSA doit vérifier la propriété que, pour toute instruction $x = \text{phi}(x_1, \dots, x_n)$ atteinte par l'exécution, au moins une des variables x_i est bien définie.

Question 18 Proposer un code SSA pour le programme (R_1).

Question 19 Proposer un algorithme pour convertir un programme RTL en un programme SSA équivalent (au sens où la séquence d'appels à `print` est identique). On pourra supposer qu'on a préalablement calculé les variables vivantes en entrée et en sortie de chaque bloc du programme RTL.

Question 20 On considère le programme SSA suivant :

```
I: n1=42 a1=0 b1=1 goto T
T: n2=phi(n1,n3) a2=phi(a1,a3) b2=phi(b1,b3) ifz n2 E B
B: t1=0 t2=t1-b2 b3=a2-t2 a3=b3-a2 t3=1 n3=n2-t3 goto T
E: print a2 halt
```

Quelles sont les instructions que l'on peut avantageusement déplacer du bloc B vers le bloc I? Expliquer pourquoi un tel déplacement est plus simple à réaliser sur un programme SSA que sur un programme RTL. De manière générale, quelle propriété doit être vérifiée lorsqu'une instruction est déplacée d'un bloc vers un autre?

Question 21 Proposer un algorithme pour convertir un programme SSA en un programme RTL équivalent (au sens où la séquence d'appels à `print` est identique), c'est-à-dire un moyen d'éliminer les fonctions ϕ .

$s ::= x = n$	<i>constante</i> $n \in \mathbb{N}$
$x = x - x$	<i>soustraction</i>
print x	<i>affichage</i>
if $x = 0$ then s else s	<i>conditionnelle</i>
while $x <> 0$ do s	<i>boucle</i>
$\{s \dots s\}$	<i>bloc</i>

FIGURE 1 – Syntaxe abstraite de WHILE.

$\frac{}{\langle E, x = n \rangle \rightarrow \langle E \oplus \{x \mapsto n\}, \{ \} \rangle}$	$\frac{x_2 \in \text{dom}(E) \quad x_3 \in \text{dom}(E)}{\langle E, x_1 = x_2 - x_3 \rangle \rightarrow \langle E \oplus \{x_1 \mapsto E(x_2) - E(x_3)\}, \{ \} \rangle}$
$\frac{x \in \text{dom}(E)}{\langle E, \text{print } x \rangle \rightarrow \langle E, \{ \} \rangle}$	
$\frac{x \in \text{dom}(E) \quad E(x) = 0}{\langle E, \text{if } x = 0 \text{ then } s_1 \text{ else } s_2 \rangle \rightarrow \langle E, s_1 \rangle}$	$\frac{x \in \text{dom}(E) \quad E(x) \neq 0}{\langle E, \text{if } x = 0 \text{ then } s_1 \text{ else } s_2 \rangle \rightarrow \langle E, s_2 \rangle}$
$\frac{x \in \text{dom}(E) \quad E(x) = 0}{\langle E, \text{while } x <> 0 \text{ do } s \rangle \rightarrow \langle E, \{ \} \rangle}$	$\frac{x \in \text{dom}(E) \quad E(x) \neq 0}{\langle E, \text{while } x <> 0 \text{ do } s \rangle \rightarrow \langle E, \{s \text{ while } x <> 0 \text{ do } s\} \rangle}$
$\frac{}{\langle E, \{ \{ \} s \dots \} \rangle \rightarrow \langle E, \{s \dots\} \rangle}$	$\frac{\langle E, s_1 \rangle \rightarrow \langle E_1, s'_1 \rangle}{\langle E, \{s_1 s_2 \dots\} \rangle \rightarrow \langle E_1, \{s'_1 s_2 \dots\} \rangle}$

FIGURE 2 – Sémantique opérationnelle de WHILE.

$p ::= b \dots b$	<i>programme RTL</i>
$b ::= L : i \dots i j$	<i>bloc de base</i>
$i ::= x = n$	<i>constante</i> $n \in \mathbb{N}$
$x = x - x$	<i>soustraction</i>
print x	<i>affichage</i>
$j ::= \text{halt}$	<i>fin du programme</i>
ifz $x L L$	<i>saut conditionnel</i>
goto L	<i>saut inconditionnel</i>

FIGURE 3 – Langage RTL.

$p' ::= b' \dots b'$	<i>programme SSA</i>
$b' ::= L : \phi \dots \phi i \dots i j$	<i>bloc de base</i>
$\phi ::= x = \text{phi}(x, \dots, x)$	<i>fonction</i> ϕ

FIGURE 4 – Langage SSA.

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>sub r_2, r_1</code>	calcule $r_1 - r_2$ dans r_1
<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

Les registres `rsp`, `rbp`, `rbx`, `r12`, `r13`, `r14` et `r15` sont *callee-saved* ; les autres sont *caller-saved*.