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 15. 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 $\to$ entre deux configurations. Un 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 $\mathrm{dom}(E)$. La relation $\langle E, s\rangle \to \langle E', s'\rangle$ 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 $\to$ par un ensemble de règles d'inférence. Dans ces règles, la notation $E \oplus \{x \mapsto n\}$ désigne la fonction E' définie par

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

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

$$\langle \{\mathtt{n} \mapsto 10\}, \mathtt{fib}\rangle \to^\star \langle \{\mathtt{a} \mapsto 55, \mathtt{n} \mapsto 0, \mathtt{b} \mapsto 89, \mathtt{t} \mapsto 1, \}, \mathtt{\{\ \}}\rangle$$

(et le programme a affiché 55).

Une configuration C est dite irréductible s'il n'existe pas de configuration C' telle que $C \to C'$. 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 $\langle \emptyset, \mathtt{print}\ x\rangle$ 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.

Correction :

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

et le programme est bloqué.

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.)

Correction : On boucle sur la valeur de b, en ajoutant a à chaque étape à un accumulateur c.

```
c = 0      // accumulateur
a = c - a  // -a plutôt que a
o = 1
while b <> 0 do {
  c = c - a
  b = b - o
}
print c
```

Pour relâcher l'hypothèse $b \geq 0$, une solution consiste, lorsque $b < 0$, à remplacer a par $-a$ et b par $-b$. Pour déterminer le signe de b, on peut faire une boucle qui explore successivement tous les entiers naturels et, pour chacun, le compare avec b et $-b$ (une soustraction puis un `if`). À la première égalité, on sort de la boucle (en mettant à 0 la variable testée par la boucle).

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

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

$$\begin{array}{rcl} S & ::= & \texttt{id = int} \\ & | & \texttt{\{}\ L\ \texttt{\}} \\ L & ::= & \epsilon \\ & | & S\ L \end{array}$$

Montrer que cette grammaire est LL(1).

Correction : On a NULL(L) = `true` et NULL(S) = `false`. On a FIRST(S) = FIRST(L) = {`id`, `{` }. On a FOLLOW(L) = {`}`}. D'où la table d'expansion

	`id`	`{`	`}`
S	`id = int`	`{` L `}`	
L	$S\ L$	$S\ L$	ϵ

(Il n'y a aucune expansion pour les lexèmes `=`, `int` et `#`.) Elle ne contient pas de conflit.

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

Correction : Le seul problème vient des deux règles `id = int` et `id = id - id` qu'il convient de factoriser, par exemple comme ceci :

$$
\begin{array}{rcl}
S & ::= & \texttt{id =}\ R \\
 & | & \texttt{print id} \\
 & | & \texttt{if id = 0 then}\ S\ \texttt{else}\ S \\
 & | & \texttt{while id <> 0 do}\ S \\
 & | & \texttt{\{}\ L\ \texttt{\}} \\
R & ::= & \texttt{int} \\
 & | & \texttt{id - id} \\
L & ::= & \epsilon \\
 & | & SL
\end{array}
$$

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 essayera de proposer deux solutions différentes.

Correction : Deux solutions au moins :

1. Au niveau de l'analyse lexicale, on construit deux lexèmes différents, un pour `0` et un autre pour les constantes littérales différentes de `0`. (La règle de priorité dans les outils de type `lex` permet de le faire très facilement en reconnaissant le premier avant le second.) Ensuite, on écrit deux règles de grammaire pour l'affectation dans l'analyseur syntaxique :

```
stmt:
| IDENT EQUAL CONST { ... }
| IDENT EQUAL ZERO  { ... }
...
```

2. Au niveau de l'analyse lexicale, on ne fait pas de distinction, avec un unique lexème `CONST` pour les constantes littérales. Dans l'analyseur syntaxique, on introduit alors un non terminal

```
zero:
| n=CONST { if n <> 0 then raise Parsing.Parse_error }
;
```

dont le but est de reconnaître uniquement `0` et on l'utilise ensuite dans les règles de grammaire pour `if` et `while`.

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 $use(s) \subseteq \mathrm{dom}(E)$, 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 $\langle E, s\rangle \to^\star \langle E', \texttt{\{ \}}\rangle$, alors on a $def(s) \subseteq \mathrm{dom}(E')$. Pour le programme ci-dessus, par exemple, on a $def(s) = \{\texttt{y}\}$ 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.

Correction :

	$def(s)$	$use(s)$
`if` x `= 0 then` s_1 `else` s_2	$def(s_1) \cap def(s_2)$	$\{x\} \cup use(s_1) \cup use(s_2)$
`while` x `<> 0 do` s	$\emptyset$	$\{x\} \cup use(s)$
`{ }`	$\emptyset$	$\emptyset$
`{` s_1 $s_2 \ldots$ `}`	$def(s_1) \cup def(\texttt{\{}\ s_2 \ldots\ \texttt{\}})$	$use(s_1) \cup (use(\texttt{\{}\ s_2 \ldots\ \texttt{\}}) \setminus def(s_1))$

Noter en particulier que la boucle `while` ne définit aucune variable, car on ne peut pas savoir, de manière générale, si son corps va être exécuté une seule fois.

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.

Correction : On commence par un lemme de monotonie : si $\langle E, s\rangle \to^\star \langle E', s'\rangle$ alors $\mathrm{dom}(E) \subseteq \mathrm{dom}(E')$. (L'environnement ne peut que croître.) Récurrence triviale sur le nombre de pas.

On poursuit par un lemme de passage au contexte : on a $\langle E, s_1\rangle \to^\star \langle E', \texttt{\{ \}}\rangle$ si et seulement si $\langle E, \texttt{\{}\, s_1\ s_2 \ldots \,\texttt{\}}\rangle \to^\star \langle E', \texttt{\{ \{ \}}\ s_2 \ldots \,\texttt{\}}\rangle$. Récurrence triviale sur le nombre de pas.

Enfin, on montre par récurrence structurelle sur l'évaluation la propriété suivante : Si $use(s) \subseteq \mathrm{dom}(E)$ alors l'exécution de $\langle E, s\rangle$ ne bloque pas et, pour toute exécution $\langle E, s\rangle \to^\star \langle E', \texttt{\{ \}}\rangle$ on a $def(s) \subseteq \mathrm{dom}(E')$.

- x `=` n : immédiat
- x_1 `=` x_2`-`x_3 : immédiat
- `print` x : immédiat
- `if` x `= 0 then` s_1 `else` s_2 : On a $x \in use(s)$ et on peut donc faire un pas, vers $\langle E, s_1\rangle$ ou $\langle E, s_2\rangle$. Dans les deux cas, l'exécution ne bloque pas par HR car $use(s_1), use(s_2) \subseteq use(s) \subseteq E$. Si $E(x) = 0$ et $\langle E, s_1\rangle \to^\star \langle E', \texttt{\{ \}}\rangle$ alors $def(s_1) \subseteq \mathrm{dom}(E')$ par HR, et $def(s) \subseteq def(s_1)$. Même chose si $E(x) \neq 0$.
- `while` x `<> 0 do` s_1 : On a $x \in use(s)$ et on peut donc faire un pas. Si $E(x) = 0$, c'est immédiat. Sinon, l'exécution ne bloque pas par HR car $use(s_1) \subseteq use(s) \subseteq E$. Et par ailleurs $def(s) = \emptyset$ donc rien à prouver pour la seconde partie.
- `{ }` : immédiat
- `{`$\,s_1\ s_2 \ldots$`}` : On a $use(s_1) \subseteq use(s)$ et donc l'exécution de $\langle E, s_1\rangle$ ne bloque pas par HR. Si elle ne termine pas, l'exécution de s non plus et elle ne bloque pas. Sinon, le lemme de passage au contexte nous dit que $\langle E, \texttt{\{}\, s_1\ s_2 \ldots \,\texttt{\}}\rangle \to^\star \langle E_1, \texttt{\{ \{ \}}\ s_2 \ldots \,\texttt{\}}\rangle \to \langle E_1, \texttt{\{}\, s_2 \ldots \,\texttt{\}}\rangle$ pour un certain E_1. Par HR on a alors $def(s_1) \subseteq \mathrm{dom}(E_1)$ et donc $use(\texttt{\{}\, s_2 \ldots \,\texttt{\}}) \subseteq \mathrm{dom}(E_1)$ en utilisant la définition de use et le lemme de monotonie. Dès lors, l'exécution de `{`$\,s_2 \ldots$`}` ne bloque pas. Si elle termine, alors $def(\texttt{\{}\, s_2 \ldots \,\texttt{\}}) \subseteq \mathrm{dom}(E')$ mais également $def(s_1) \subseteq \mathrm{dom}(E')$ par monotonie, d'où le résultat.

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.

Correction : Calculer des ensembles minimaux reviendrait à pouvoir déterminer quand une variable prend une valeur nulle, ce qui n'est pas décidable en toute généralité. Nos ensembles $def(s)$ et $use(s)$ ont donc vocation à être des approximations, comme dans l'exemple trivial suivant

```
{ if x = 0 then x = 1 else {}
  while x <> 0 do { y = 2 x = 0 } }
```

où on ne détermine pas que `y` est définie. Bien entendu, on pourrait ici le déterminer, en propageant simplement la valeur de `x` (ou plutôt sa nullité), mais un code arbitraire aboutissant à une valeur de `x` non nul peut être substitué à la première instruction.

De manière plus formelle, calculer un ensemble *use* minimal pour le programme

```
{ ...p... print x }
```

où le programme `p` n'utilise pas la variable `x` revient à déterminer le problème de l'arrêt du programme `p`.

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 ?

Correction : Ce programme calcule et affiche la factorielle de `n`. En effet, il correspond à la double boucle

```
u=1
for r=0 to n-1 do  // invariant u = fact(r)
  v = u
  for s=0 to r-1 do  // invariant u = (s+1) * fact(r)
    u = u+v
print u  // avec donc u = fact(n)
```

Note : c'est le programme de *Checking a Large Routine* (Turing, 1949).

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).

Correction : Une syntaxe abstraite naturelle est la suivante :

```
type var = string
type label = string
module M = Map.Make(String)

type instr =
| Iprint of var
| Icst of var * int
| Isub of var * var * var

type branch =
| Halt
| Ifz of var * label * label
| Goto of label

type block = { code: instr list; branch: branch; }
type rlt = { entry: label; blocks: block M.t; }
```

On a choisi de représenter l'ensemble des blocs par un dictionnaire, afin de faciliter l'exécution. (Une autre solution serait d'utiliser des entiers $0, \ldots, n-1$ pour les étiquettes et directement un tableau pour l'ensemble des blocs.)

Pour l'exécution, le plus simple est d'utiliser une table de hachage (module `H` ci-dessous) pour représenter l'environnement :

```
let exec {entry; blocks} =
  let vars = H.create 16 in
  let var x = try H.find vars x
    with Not_found -> eprintf "unbound variable %s@." x; exit 1 in
  let rec instr = function
    | Iprint x          -> printf "  %d@." (var x)
    | Icst (x, n)       -> H.replace vars x n
    | Isub (x1, x2, x3) -> H.replace vars x1 (var x2 - var x3) in
  let rec exec l =
    let b = M.find l blocks in
    List.iter instr b.code;
    match b.branch with
    | Halt            -> ()
    | Goto l          -> exec l
    | Ifz (x, l1, l2) -> exec (if var x = 0 then l1 else l2) in
  exec entry
```

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.

Correction : On remarque qu'on peut calculer les ensembles *def* et *use* d'un bloc de base une fois pour toutes, indépendamment des autres blocs. On procède exactement comme pour un bloc du langage WHILE.

Ensuite, on peut calculer les variables vivantes en entrée et en sortie de chaque bloc avec un calcul de point fixe, exactement comme on l'a fait en cours, avec les équations

$$\begin{cases} in(l) &= use(l) \cup (out(l) \backslash def(l)) \\ out(l) &= \bigcup_{s \in succ(l)} in(s) \end{cases}$$

et par exemple l'algorithme de Kildall. L'ensemble cherché sera alors donné par in pour le premier bloc.

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.

Correction : Écrivons la compilation sous la forme d'une fonction $C(s, j)$ qui prend en paramètres une instruction s du langage WHILE et un branchement (à effectuer à l'issue), et qui renvoie l'étiquette du bloc correspondant à ce calcul. Pour répondre à la question, il suffit de calculer $C(s, \texttt{halt})$ et de considérer le résultat comme le premier bloc.

On se donne une fonction auxiliaire $B(i_1 \dots i_n\ j)$ qui construit le bloc $L : i_1 \dots i_n\ j$ pour une étiquette fraîche L et la renvoie, lorsque $n > 0$ ou $j \neq \texttt{goto}$, et renvoie directement L lorsque $n = 0$ et $j = \texttt{goto}\ L$.

Pour calculer $C(s, j)$, on commence par collecter la séquence $i_1 \dots i_n$ de toutes les instructions élémentaires « au début » de s, jusqu'au premier `if` ou `while`. S'il n'y en a pas, c'est que s est de la forme

$$\texttt{\{}\ i_1 \dots i_n\ \texttt{\}}$$

et on renvoie simplement le bloc $B(i_1 \dots i_n\ j)$.

Si s est de la forme

$$\texttt{\{}\ i_1 \dots i_n\ \texttt{if}\ x\ \texttt{=}\ \texttt{0}\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ s_3 \dots\ \texttt{\}}$$

on pose $L = C(\texttt{\{}\ s_3 \dots\ \texttt{\}}, j)$, puis $L_1 = C(s_1, \texttt{goto}\ L)$ et $L_2 = C(s_2, \texttt{goto}\ L)$, et on renvoie $B(i_1 \dots i_n, \texttt{ifz}\ x\ L_1\ L_2)$. (Il y a bien deux `goto` vers L.)

Enfin, si s est de la forme

$$\texttt{\{}\ i_1 \dots i_n\ \texttt{while}\ x\ \texttt{<>}\ \texttt{0}\ \texttt{do}\ s_1\ s_2 \dots\ \texttt{\}}$$

on pose $L = C(\texttt{\{}\ s_2 \dots\ \texttt{\}}, j)$, puis on choisit une étiquette fraîche L_t et on pose $L_b = C(s_1, \texttt{goto}\ L_t)$, on déclare le bloc $L_t : \texttt{ifz}\ x\ L\ L_b$, et enfin on renvoie $B(i_1 \dots i_n, \texttt{goto}\ L_t)$. (Il y a bien deux `goto` vers L_t.)

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                    (R1)
H: print s  halt
```

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

Correction : On a cinq variables (`n`, `s`, `zero`, `one` et `t`) qui interfèrent toutes deux à deux et forment donc une clique. On peut avantageusement ajouter une arête de préférence entre `s` et `%rdi` (paramètre de `print`).

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

Correction : Le plus simple est ici d'utiliser des registres *caller-saved*, car l'instruction `print` est utilisée tout à la fin du programme.

```
        ##    s %rdi
        ##  one %rsi
        ## zero %rdx
        ##    n %rcx
        ##    t %r8
main:   mov $10, %rcx
        mov $0, %rdi
        mov $1, %rsi
        mov $0, %rdx
B:      mov %rdx, %r8
        subq %rcx, %r8
        subq %r8, %rdi
        subq %rsi, %rcx
        jnz  B
H:      call print
        xorq %rax, %rax
        ret
```

Question 15 Proposer un code assembleur pour le programme (R_0) de la page 7.

Correction : Cette fois, on a au contraire intérêt à prendre des registres *callee-saved* pour les variables `n`, `r` et `one`. La variable `t`, en revanche, peut rester dans un *caller-saved*.

```
        ##    n %rbx
        ##    r %r12
        ##  one %r13
        ##    t %rcx
main:   pushq %rbx
        pushq %r12
        pushq %r13
        mov $10, %rbx
        mov $0, %r12
        mov $1, %r13
        subq %r12, %r13         # one = r - one
        negq %r13               # cf plus bas
L1:     mov %r12, %r8
        subq %rbx, %r8
        jz  L2
L3:     subq %r13, %r12
        movq %r12, %rdi
        call print
        jmp L1
L2:     popq %r13
        popq %r12
        popq %rbx
        xorq %rax, %rax
        ret
```

(Bien entendu, le code pourrait être plus simple mais on le compile ici comme si c'était fait de façon mécanique, à l'instar de ce qui est demandé dans la question 17.)

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

Correction : Il y a deux endroits où une préférence peut être exprimée :

- dans une instruction $x = y - z$, entre les variables x et y. En effet, la soustraction $x = x - z$ pourra alors être directement compilée par une instruction assembleur `sub`.
- dans une instruction `print` x, entre la variable x et le registre `%rdi`.

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.

Correction : On fait une linéarisation comme vu en cours. On réserve un registre comme temporaire (par exemple `%r10`). En début de programme, on sauvegarde sur la pile les registres *callee-saved* qui vont être utilisés.

x `=` n `:` C'est directement une instruction `mov`.

x `=` y `-` z `:` Comme la soustraction de x86-64 est à deux opérandes seulement, on peut distinguer trois cas :

- x `=` x `-` z `:` C'est le cas favorable discuté dans la question précédente, qui se traduit directement par `sub`.
- x `=` y `-` z **avec** $y \neq x$ **et** $z \neq x$ `:` On copie y dans x puis on lui soustrait z.
- x `=` y `-` x `:` On soustrait y de x, puis on fait `neg` sur x. (C'est le cas `one = r - one` dans le code assembleur de la question 15.)

Bien entendu, dans tous les cas, il faut tenir compte de la possibilité que plusieurs variables soient vidées en mémoire. Il faut alors utiliser un temporaire.

`print` x `:` On copie x dans `%rdi`, si nécessaire, puis on fait `call print`.

`ifz` x L_1 L_2 `:` Si le saut est immédiatement précédé (dans le même bloc de base) d'une soustraction x `=` y `-` z, alors on peut directement faire un saut conditionnel (`jz` ou `jnz` au choix). Sinon, il faut tester x explicitement (avec `test` x, x) avant le saut.

`goto` L `:` C'est directement une instruction `jmp`.

`halt :` On restaure les *callee-saved*, le cas échéant, on met `%rax` à zéro puis on termine le programme avec `ret`.

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 :

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

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 `= phi(`$x_1, \ldots, x_n$`)`. 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 `= phi(`$x_1, \ldots, x_n$`)` 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).

Correction :

```
A: zero=0  s1=0  one=1  n1=10  goto B
B: n2=phi(n1, n3)  s2=phi(s1,s3)  t=zero-n2  s3=s2-t  n3=n2-one  ifz n3 H B
H: print s3  halt
```

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.

Correction : Voici un algorithme simple.

1. Pour chaque bloc B, dans un ordre arbitraire :
   (a) Pour toutes ses variables vivantes en entrée $x, y, z \ldots$ on choisit des noms frais $x_i, y_j, z_k \ldots$.
   (b) On traduit le code du bloc, instruction par instruction, en partant de ces variables, et en introduisant de nouvelles variables à chaque affectation.
   (c) On n'oublie pas de traduire la variable dans le saut `ifz`, le cas échéant.

(d) On retient les variables auxquelles on est arrivé pour toutes les variables vivantes en sortie de B.

2. On examine de nouveau tous les blocs, et pour chaque bloc B on insère une fonction ϕ pour chaque variable x vivante en entrée de B, de la forme $x = \texttt{phi}(x_1, \ldots, x_n)$ où les x_i sont les variables (pour x) en sortie de tous les blocs prédécesseurs de B, que l'on connaît à l'issue de la phase 1.

Note : Ce n'est pas optimal, au sens où certaines fonctions ϕ sont inutiles car n'ayant qu'un seul argument, mais c'est correct. On pourrait les supprimer dans une seconde passe, ou même ne pas les produire initialement mais cela demande un effort supplémentaire.

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 ?

Correction : On peut sortir les instruction `t1=0` et `t3=1` de la boucle. En effet, la forme SSA nous garantit qu'une variable x définie par $x = n$ aura toujours la valeur n, quelles que soit ses utilisations. (Ici, leurs utilisations de `t1` et `t3` sont uniques et situées immédiatement dans l'instruction suivante, mais ce serait vraie de manière générale.)

Dans un programme RTL, il pourrait y avoir d'autres définitions de x entre $x = n$ et une utilisation de x, ce qui rendrait l'analyse plus complexe.

Pour que le déplacement soit correct de manière générale, il peut déplacer une instruction d'un bloc B uniquement vers un bloc A qui *domine* B, au sens où toute exécution atteignant B est passée par A. (Il existe dans la littérature des algorithmes pour calculer les dominateurs dans un graphe, que les compilateurs peuvent alors utiliser dans leurs optimisations.)

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.

Correction : Pour chaque fonction $x = \texttt{phi}(x_1, \ldots, x_n)$ au début d'un bloc B, il suffit d'ajouter une affectation $x{=}x_i$ à la fin du bloc prédécesseur de B qui définit la variable x_i.

Note : On n'a pas d'instruction de copie dans notre langage RTL mais ce serait évident de l'ajouter et par ailleurs on peut la réaliser trivialement avec $x = y - z$ où z est une variable qui vaut toujours 0.

$$
\begin{array}{rcll}
s & ::= & x\texttt{=}n & \textit{constante } n \in \mathbb{N} \\
& | & x\texttt{=}x\texttt{-}x & \textit{soustraction} \\
& | & \texttt{print}\ x & \textit{affichage} \\
& | & \texttt{if}\ x\ \texttt{=}\ \texttt{0}\ \texttt{then}\ s\ \texttt{else}\ s & \textit{conditionnelle} \\
& | & \texttt{while}\ x\ \texttt{<>}\ \texttt{0}\ \texttt{do}\ s & \textit{boucle} \\
& | & \texttt{\{}\, s \ldots s\, \texttt{\}} & \textit{bloc}
\end{array}
$$

FIGURE 1 – Syntaxe abstraite de WHILE.

$$
\frac{}{\langle E, x\texttt{=}n\rangle \to \langle E \oplus \{x \mapsto n\}, \texttt{\{ \}}\rangle}
\qquad
\frac{x_2 \in \mathrm{dom}(E) \quad x_3 \in \mathrm{dom}(E)}{\langle E, x_1\texttt{=}x_2\texttt{-}x_3\rangle \to \langle E \oplus \{x_1 \mapsto E(x_2) - E(x_3)\}, \texttt{\{ \}}\rangle}
$$

$$
\frac{x \in \mathrm{dom}(E)}{\langle E, \texttt{print}\ x\rangle \to \langle E, \texttt{\{ \}}\rangle}
$$

$$
\frac{x \in \mathrm{dom}(E) \quad E(x) = 0}{\langle E, \texttt{if}\ x\ \texttt{=}\ \texttt{0}\ \texttt{then}\ s_1\ \texttt{else}\ s_2\rangle \to \langle E, s_1\rangle}
\qquad
\frac{x \in \mathrm{dom}(E) \quad E(x) \neq 0}{\langle E, \texttt{if}\ x\ \texttt{=}\ \texttt{0}\ \texttt{then}\ s_1\ \texttt{else}\ s_2\rangle \to \langle E, s_2\rangle}
$$

$$
\frac{x \in \mathrm{dom}(E) \quad E(x) = 0}{\langle E, \texttt{while}\ x\ \texttt{<>}\ \texttt{0}\ \texttt{do}\ s\rangle \to \langle E, \texttt{\{ \}}\rangle}
\qquad
\frac{x \in \mathrm{dom}(E) \quad E(x) \neq 0}{\langle E, \texttt{while}\ x\ \texttt{<>}\ \texttt{0}\ \texttt{do}\ s\rangle \to \langle E, \texttt{\{}\, s\ \texttt{while}\ x\ \texttt{<>}\ \texttt{0}\ \texttt{do}\ s\, \texttt{\}}\rangle}
$$

$$
\frac{}{\langle E, \texttt{\{\{ \}}\ s \ldots \texttt{\}}\rangle \to \langle E, \texttt{\{}\, s \ldots \texttt{\}}\rangle}
\qquad
\frac{\langle E, s_1\rangle \to \langle E_1, s_1'\rangle}{\langle E, \texttt{\{}\, s_1\ s_2 \ldots \texttt{\}}\rangle \to \langle E_1, \texttt{\{}\, s_1'\ s_2 \ldots \texttt{\}}\rangle}
$$

FIGURE 2 – Sémantique opérationnelle de WHILE.

$$
\begin{array}{rcll}
p & ::= & b \ldots b & \textit{programme RTL} \\
b & ::= & L : i \ldots i\ j & \textit{bloc de base} \\
i & ::= & x\texttt{=}n & \textit{constante } n \in \mathbb{N} \\
& | & x\texttt{=}x\texttt{-}x & \textit{soustraction} \\
& | & \texttt{print}\ x & \textit{affichage} \\
j & ::= & \texttt{halt} & \textit{fin du programme} \\
& | & \texttt{ifz}\ x\ L\ L & \textit{saut conditionnel} \\
& | & \texttt{goto}\ L & \textit{saut inconditionnel}
\end{array}
$$

FIGURE 3 – Langage RTL.

$$
\begin{array}{rcll}
p' & ::= & b' \ldots b' & \textit{programme SSA} \\
b' & ::= & L : \phi \ldots \phi\ i \ldots i\ j & \textit{bloc de base} \\
\phi & ::= & x\texttt{=}\texttt{phi(}x, \ldots, x\texttt{)} & \textit{fonction } \phi
\end{array}
$$

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.

`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
`sub` r_2`,` r_1	calcule $r_1 - r_2$ dans r_1
`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

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