Corrigé de l'examen de compilation

Énsiie, semestre 3

18 janvier 2011

Exercice 1 : Analyse syntaxique *(8 points)*

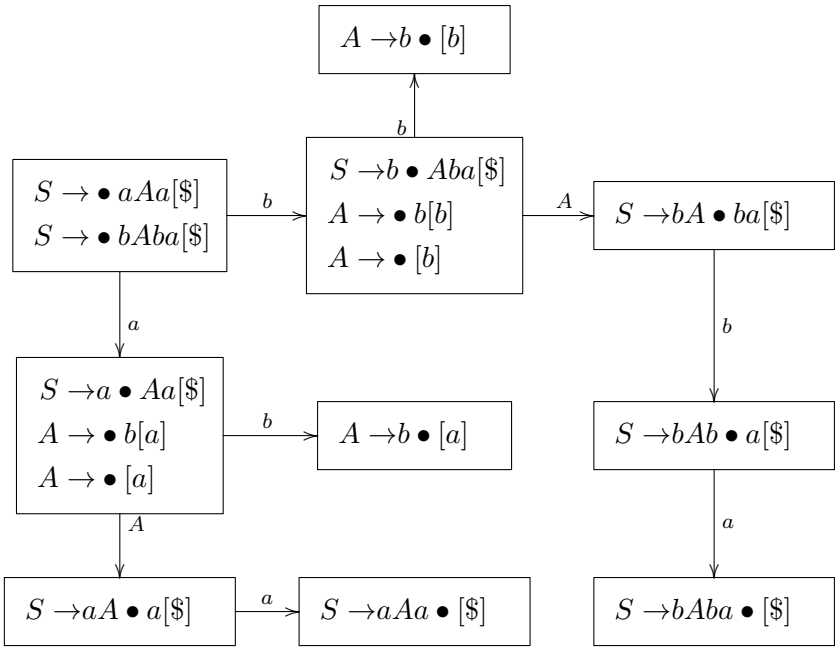
1.

w	$First(w)$
aAa	a
$bAba$	b
b	b

w	$Follow(w)$
A	$a\ b$

$First(b) \cap Follow(A) \neq \emptyset$ donc la grammaire G_1 n'est pas dans LL(1).

Automate déterministe LR(1)



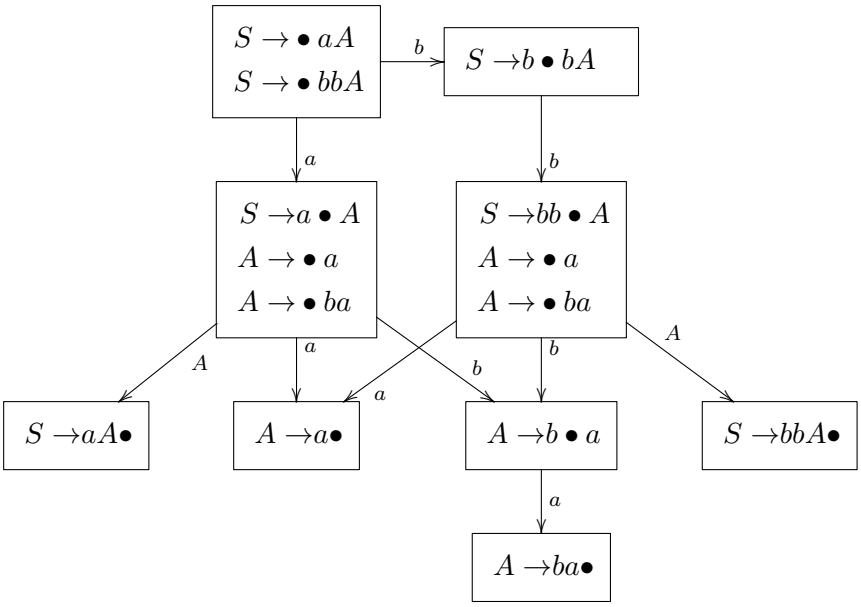
On voit un conflit shift/reduce dans l'état $\boxed{\begin{array}{l} S \to b \bullet Aba[\$] \\ A \to \bullet\, b[b] \\ A \to \bullet\, [b] \end{array}}$: on peut décaler b ou

réduire $A \to \epsilon$. La grammaire n'est donc pas LR(1). A fortiori, elle n'est pas LR(0), ni LL(1).

2. On a $\mathcal{L}(G_1) = \{aa; aba; bba; bbba\}$. La grammaire

$$\begin{aligned} S &\to aA \\ S &\to bbA \\ A &\to a \\ A &\to ba \end{aligned}$$

reconnaît le même langage et est LL(1) (le calcul des ensembles $First$ est trivial) et LR(0) : l'automate déterministe est

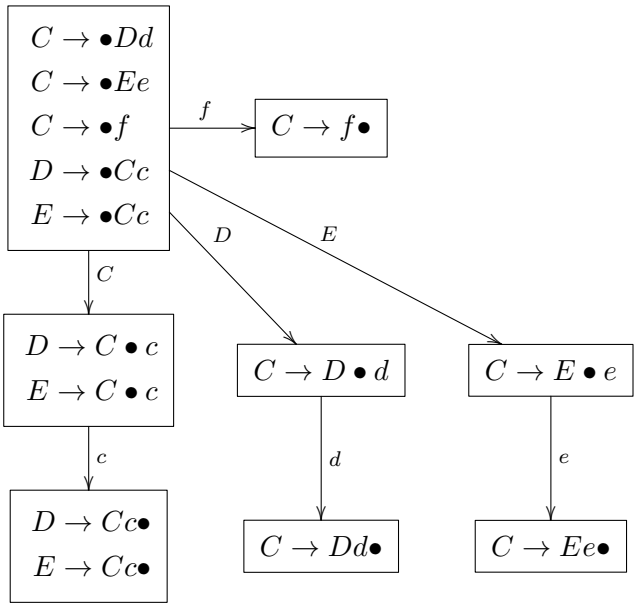


3.

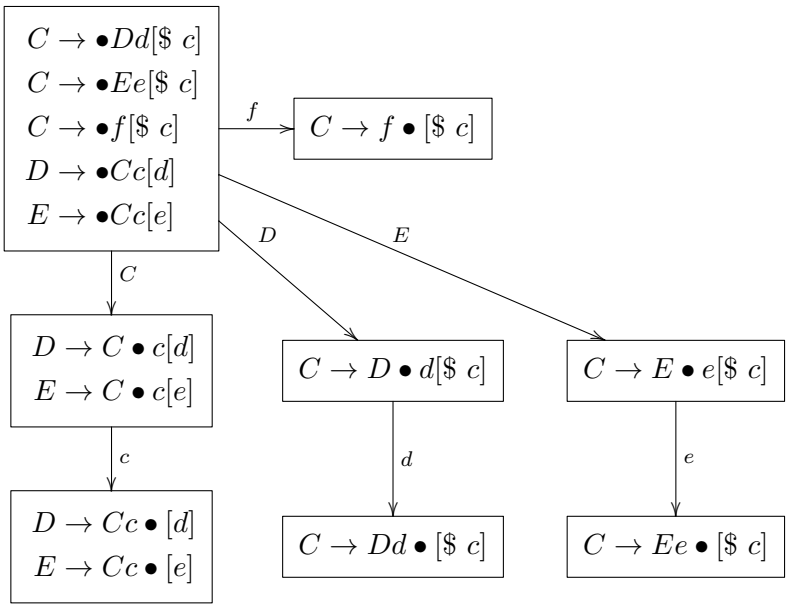
w	$First(w)$
Dd	f
Ee	f
f	f
Cc	f
C	f

On voit que les ensembles $First$ ne sont pas deux-à-deux disjoint, la grammaire n'est pas LL(1).

Automate déterministe LR(0) :



On voit un conflit reduce/reduce dans l'état $\boxed{\begin{array}{l} D \to Cc\bullet \\ E \to Cc\bullet \end{array}}$ où on peut réduire par $D \to Cc$ ou par $E \to Cc$. Par conséquent la grammaire n'est pas LR(0).
Automate déterministe LR(1) :



Ici, il n'y a pas de conflit ; la grammaire est LR(1).

Pour rendre la grammaire LL(1), on applique les techniques habituelles :

– mise en forme récursive directe :

$$C \to Ccd$$
$$C \to Cce$$
$$C \to f$$

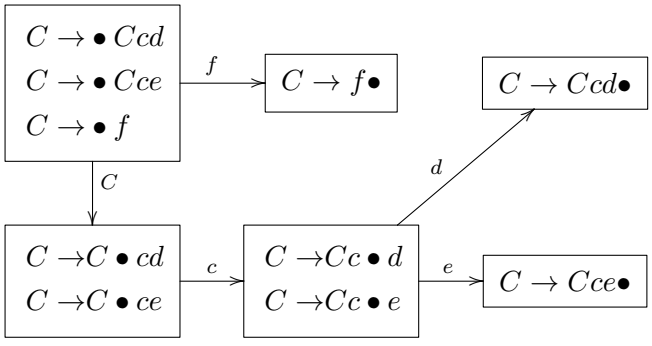
– élimination de la récursivité à gauche :

$$C \to fC'$$
$$C' \to cdC'$$
$$C' \to ceC'$$
$$C' \to \epsilon$$

– factorisation à gauche :

$$C \to fC'$$
$$C' \to cC"$$
$$C' \to \epsilon$$
$$C" \to dC'$$
$$C" \to eC'$$

On peut vérifier que la grammaire obtenue est bien LL(1).
Pour obtenir une grammaire LR(0), seule la mise en forme récursive directe est nécessaire, on a alors l'automate

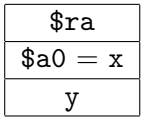


Exercice 2 : Pile d'appel *(8 points)*

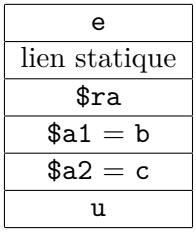
1. – `f` : il faut sauvegarder `$ra` en début d'appel car il est écrasé par les appels à `i` et `g` ; il faut sauvegarder `$a0` contenant `x` avant l'appel à `g` car il est utilisé par la suite ; il n'est pas nécessaire de sauvegarder `$a1`, qui contient le lien statique, car il n'est pas utilisé.

– `g` : il faut sauvegarder `$ra` en début d'appel car il est écrasé par les appels à `h` et `i` ; il faut sauvegarder `$a1` et `$a2` contenant `b` et `c` avant l'appel à `h` car ils sont utilisés par la suite ; il n'est pas nécessaire de sauvegarder `$a0`, qui contient `a`, car il n'est pas utilisé après l'appel à `h`.
– `h` : il ne faut pas sauvegarder `$ra` car il n'y a pas d'appel qui l'écraserait ; de même, les registres `$a0`–`$a3` n'ont pas besoin d'être sauvegardés.
– `i` : idem.

2. `f` :

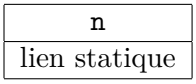


`g` :



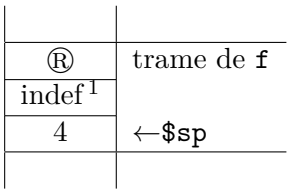
`h` :

`i` :

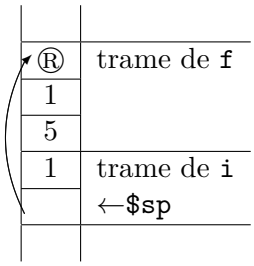


On peut choisir de sauvegarder les registres dans un ordre différent.

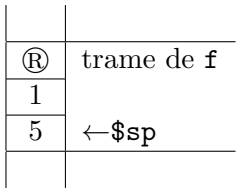
3. ① `$a0` 1
`$a1` indef (lien statique)
`$a2` indef
`$a3` indef
`$v0` indef
`$ra` Ⓡ



② `$a0` 1
`$a1` 4
`$a2` 1
`$a3` 4
`$v0` indef
`$ra` ③

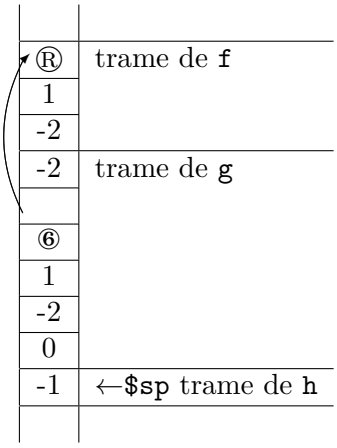


③ `$a0` 1 (restauré)
`$a1` 4
`$a2` 1
`$a3` 4
`$v0` -2
`$ra` ③

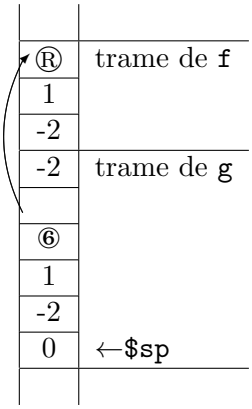


1. sauvegardé uniquement avant l'appel à `g`

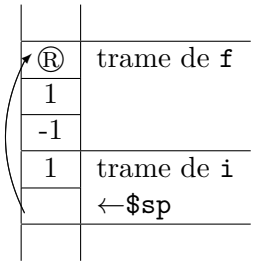
④ `$a0` -2
`$a1` -2
`$a2` début de trame de `g`
`$a3` 1
`$v0` -2
`$ra` ⑤



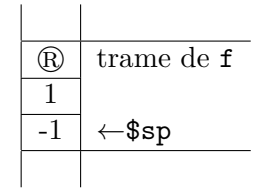
⑤ `$a0` -2
`$a1` 1 (restauré après appel)
`$a2` -2 (restauré après appel)
`$a3` 1
`$v0` -3
`$ra` ⑤



② `$a0` 1
`$a1` -2
`$a2` -2
`$a3` 1
`$v0` -3
`$ra` ⑥



⑥ `$a0` 1
`$a1` -2
`$a2` -2
`$a3` 1
`$v0` -2
`$ra` ⑥

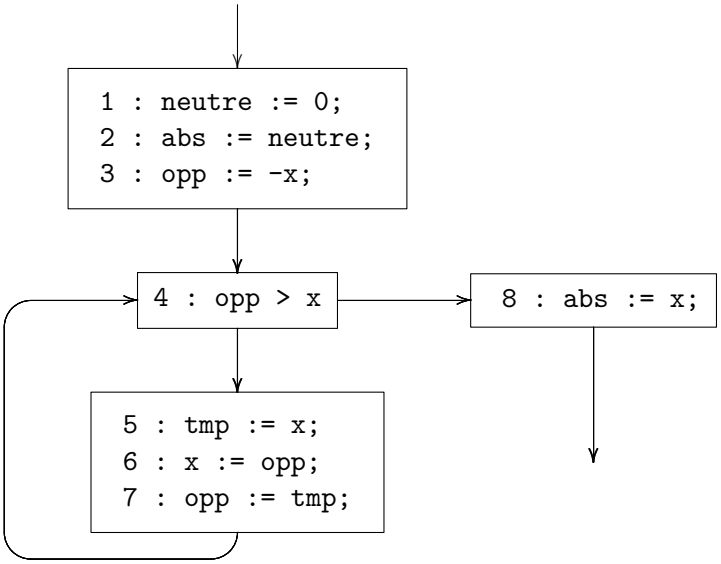


4.
```
addi $sp, $sp, -4 # création de la trame
lw %0, -4($a2)    # debut de trame de f dans %0
lw %1, -4(%0)     # valeur de x dans %1
add %2, %1, $a1   # calcul de la valeur de t
sw %2, 0($sp)     # stockage de t sur la pile
```

```
add $v0, %2, $a0  # valeur de retour
addi $sp, $sp, 4  # destruction de la trame
jr $ra            # retour à l'appelant
```

Exercice 3 : Allocation de registres *(5 points)*

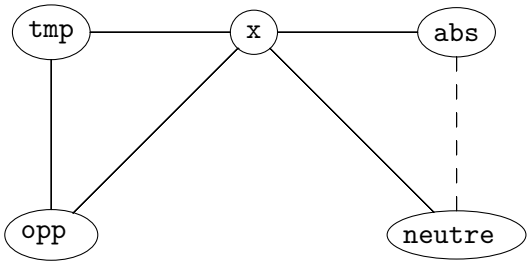
1.



2.

instruction	vivantes avant	vivantes apres
1	x	x neutre
2	x neutre	x
3	x	x opp
4	x opp	x opp
5	x opp	opp tmp
6	opp tmp	x tmp
7	x tmp	x opp
8	x	

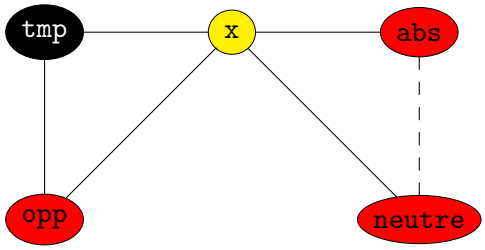
3.



4. Quelque soit le critère choisi :
   – fusionner `abs` et `neutre`

– simplifier `abs/neutre`
– spiller `tmp`
– simplifier `x`
– simplifier `opp`

Le seul choix possible concerne l'ordre des deux dernières étapes. On obtient le coloriage :



Exercice 4 : *Reaching definitions (9 points)*

1.

i	$Reach((avant, i), \texttt{x})$	$Reach((après, i), \texttt{x})$
`i1`		`i1`
`i2`	`i1`	`i2`
`i3`	`i1 i2`	`i1 i2`
`i4`	`i1 i2`	`i4`

2. $$Reach((avant, i), \texttt{x}) = \bigcup_{j \to i} Reach((après, j), \texttt{x})$$

3. $$Reach((après, i), \texttt{\%v}) = \{i\}$$

   `i` redéfinit `%v` donc de tous les chemins qui arrivent après `i`, seul celui partant de `i` ne redéfinit pas `%v`.

   $$Reach((après, i), w) = Reach((avant, i), w) \qquad \text{pour } w \neq \texttt{\%v}$$

   `i` ne redéfinit pas w. Si on a un chemin d'une instruction `j` au point avant `i` qui ne redéfinit pas w, alors en complétant ce chemin pour aller après `i` on ne redéfinit toujours pas w.

4. On pose $\mathcal{T} = (\{avant; après\} \times \mathcal{I}nstr) \times \mathcal{V}ar \to \mathcal{P}(\mathcal{I}nstr)$. On a $Reach \in \mathcal{T}$. Comme ensemble des parties, $(\mathcal{P}(\mathcal{I}nstr), \subseteq)$ est un treillis complet, donc $(\mathcal{T}, \dot{\subseteq})$ (ordre point-à-point) l'est aussi. On définit une fonction $F : \mathcal{T} \to \mathcal{T}$ telle que

$$\begin{aligned}
F(f)((avant, i), w) &= \bigcup_{j \to i} f((après, j), w) \\
F(f)((après, i), w) &= \{i\} && \text{si } i \text{ définit } w \\
F(f)((après, i), w) &= f((avant, i), w) && \text{si } i \text{ ne définit pas } w
\end{aligned}$$

On peut vérifier que F est monotone dans $(\mathcal{T}, \dot{\subseteq})$, donc par le théorème de Knaster-Tarski, elle admet des points fixes. On vérifie facilement que *Reach* est un point fixe de F. Il s'agit du plus petit car on peut montrer que tout point fixe de F surapproxime les ensembles *Reach*.

5.
```
pour toute instruction i
  pour toute variable v
    Reach((avant,i),v) ← ∅
    Reach((apres,i),v) ← ∅
W ← Instr
tant que W ≠ ∅
  retirer i de W, si possible sans prédécesseur dans W
  pour tout v
    pour tout j tel que j→i
      Reach((avant,i),v) ← Reach((avant,i),v) ∪ Reach((apres,j),v)
    si definit(i, v)
      tmp ← i
    sinon
      tmp ← Reach((avant,i),v)
    si tmp ≠ Reach((apres,i),v)
      ajouter à W les successeurs de i
      Reach((apres,i),v) ← tmp
```