Compilation EISE4 – TD

Exercice 1 : Grammaire Hors-Contexte et Ambiguïté

Soit la grammaire suivante :
$G_1 =< \{a, b\}, \{S\}, S, R >$ et $R : S \rightarrow aSb|aS|\varepsilon$

- Quel est le langage engendré par la grammaire ?
- Montrer que cette grammaire est ambigüe
- Donner une grammaire équivalente non-ambigüe

<**correction**>
$L = \{a^n b^p | n \geq p\}$

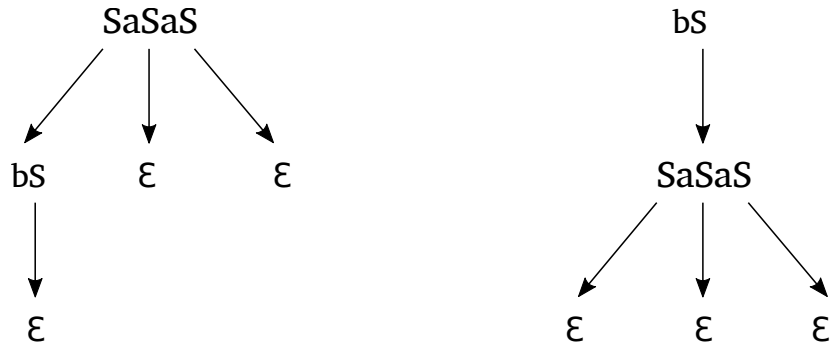
Le mot `aab` peut par exemple être obtenu de deux manières : en prenant d'abord la règle $S \rightarrow aSb$ puis la règle $S \rightarrow aS$, ou les mêmes règles dans l'ordre inverse (à chaque fois suivies de la règle $S \rightarrow \varepsilon$)

Grammaire non ambigüe équivalente : $G_1' =< \{a, b\}, \{S, T\}, S, R >$
R :
$S \rightarrow aSb|T$
$T \rightarrow aT|\varepsilon$
</**correction**>

Mêmes questions avec la grammaire $G_2 =< \{a, b\}, \{S\}, S, R >$ et $R : S \rightarrow SaSaS|bS|\varepsilon$

<**correction**>
$L = \{\{a, b\}^* |$ il y a un nombre pair de a$\}$

Le mot `baa` peut par exemple être obtenu de deux manières :



Grammaire non ambigüe équivalente : $G_1' =< \{a, b\}, \{S, T\}, S, R >$
R :
$S \rightarrow SaSaS|bT|\varepsilon$
$T \rightarrow bT|\varepsilon$

</**correction**>

Exercice 2 : Analyse syntaxique

Soit le programme yacc suivant, qui permet de reconnaitre les mots du langage $x^n a y^n$:

```
1  A : 'x' A 'y'     { printf("A -> x A y\n"); }
2    | 'a'           { printf("A -> a\n"); }
3    ;
```

- Qu'affiche ce programme appliqué à la chaine d'entrée `xxayy` ?
- Afficher les états successifs de la pile de yacc pour la chaine `xxayy`.

<**correction**>
Ce programme affiche :
A -> a
A -> x A y
A -> x A y

On a des "empilement" (transfert, shift) et des "réductions" (reduce).

```
                                  y|
                     a| <==   A   A|          y|
                  x  x        x   x| <== A    A|
               x  x  x        x   x       x    x| <== A
fond de pile -  -  -  -        -   -       -    -       -
```

Les flèches <== correspondent à des réductions. Lorsqu'on réduit, l'action correspondante est effectuée.

L'analyse est ascendante : on "applique" d'abord les règles associées aux feuilles de l'arbre syntaxique, puis on remonte vers l'axiome.
</**correction**>

Soit le langage ab^*c engendré par la grammaire :

```
1  A : 'a' B
2    ;
3  B : 'b' B
4    | 'c'
5    ;
```

- Dessiner les états successifs de la pile de yacc pour la chaine d'entrée `abbbc`
- Quel problème cela peut-il poser ? Donner un grammaire équivalente qui résout le problème.

<**correction**>
États de la pile :

```
                        c| <== |B|
                     b  b       b| <== |B|
                  b  b  b       b       b| <== |B|
               b  b  b  b       b       b       b| <== B|
            a  a  a  a  a       a       a       a       a| <== A
fond de pile -  -  -  -  -       -       -       -       -       -
```

On doit empiler toute la chaîne d'entrée avant de commencer à réduire. Problème : si on doit reconnaitre des chaînes très longue, la pile de yacc peut grossir.

Cela vient du fait que la grammaire est récursive à droite. On peut reconnaitre le même langage avec une grammaire récursive à gauche :

```
1  A : B 'c'
2    ;
3  B : B 'b'
4    | 'a'
5    ;
```

La pile est désormais :

```
                        b|        b|        b|        c|
                  a| <== B  B| <== B  B| <== B  B| <== B  B| <== A
fond de pile -  -      -  -      -  -      -  -      -  -      -
```

Conclusion : on utilise de préférence des règles récursives à gauche, pour des raisons de place mémoire (ex : `liste_inst` → `liste_inst inst`).
</**correction**>

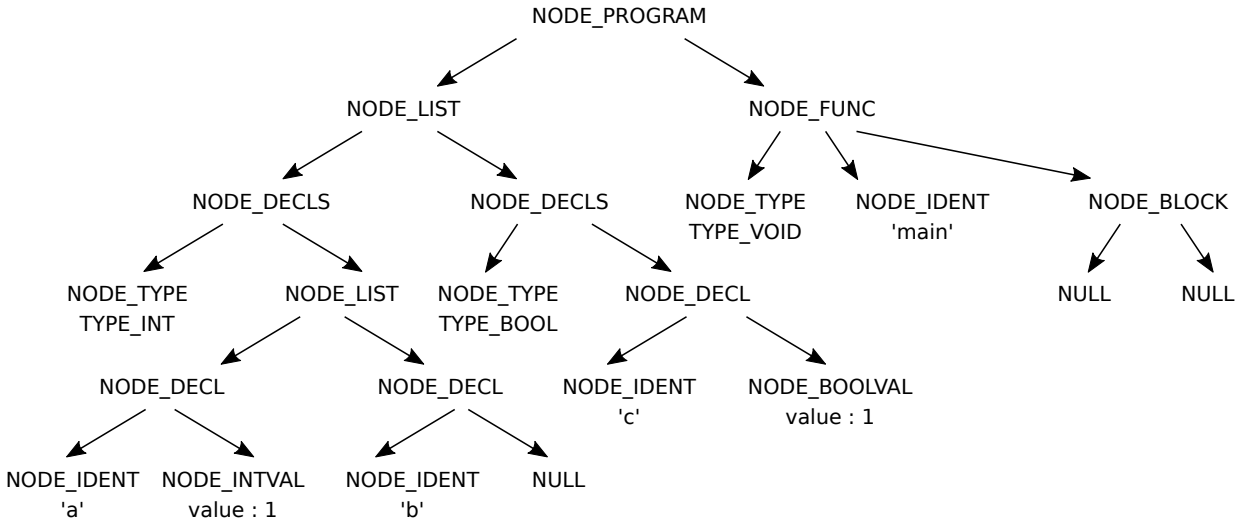
Exercice 3 : MiniC

Soit le programme MiniC suivant :

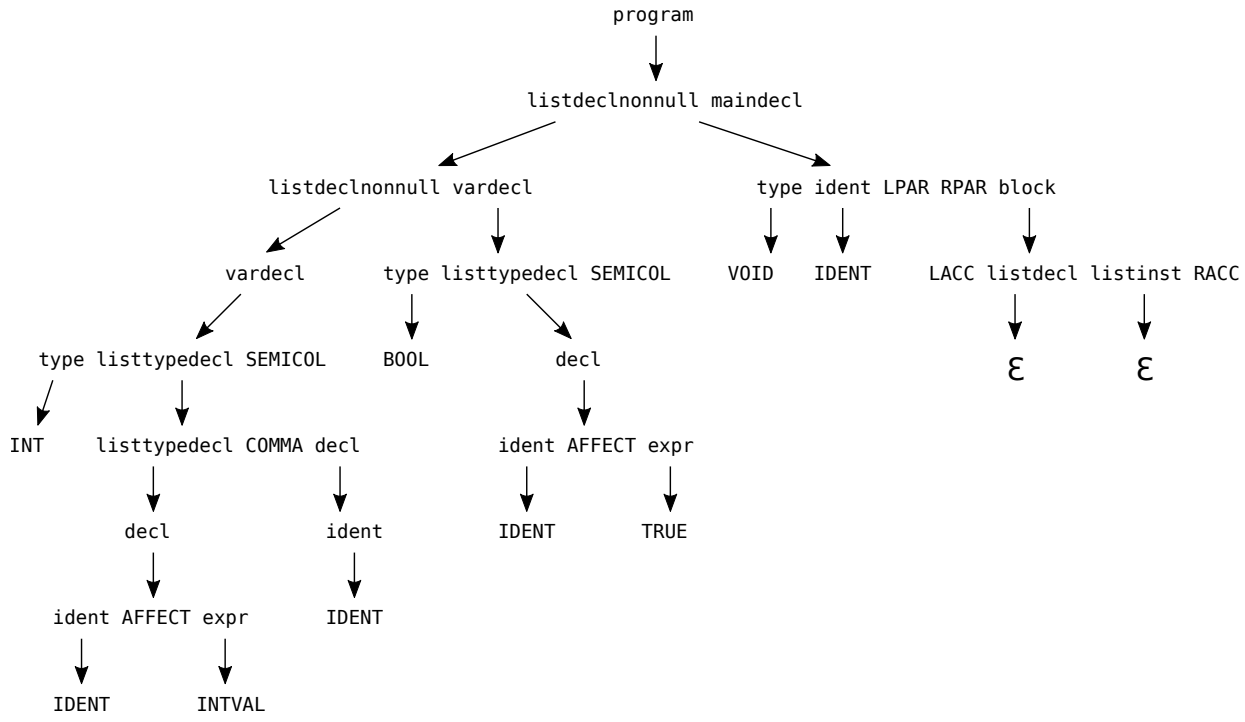
```
1  int a = 1, b;
2  bool c = true;
3  void main() {
4  }
```

- Dessiner l'arbre de ce programme à la fin de l'analyse syntaxique
- Dessiner l'arbre de dérivation correspondant à ce programme dans la grammaire hors-contexte de MiniC (règles qui sont prise depuis l'axiome pour arriver à ce programme)
- Donner le code assembleur mips correspondant à ce programme

<**correction**>
Arbre du programme après l'analyse syntaxique :



Arbre de dérivation dans la grammaire hors-contexte (le préfixe `TOK` des tokens est omis) :



Code assembleur :

```
.data

a: .word 1
b: .word 0
c: .word 1

.text

main:
    # Les 2 premieres instructions ne sont pas necessaires :
    # on peut tester si la valeur a allouer est 0 et dans ce cas
    # ne pas produire ces instructions
    addiu $29, $29, 0
    addiu $29, $29, 0
    ori   $2, $0, 10
    syscall
```

</**correction**>

Exercice 4 : MiniC

Soit le programme MiniC suivant :

```
1  void main() {
2    int a = 120, b = 80;
3    if (a > b) {
4      a = a - b;
5    }
6    print("a = ", a, " - b = ", b);
7  }
```

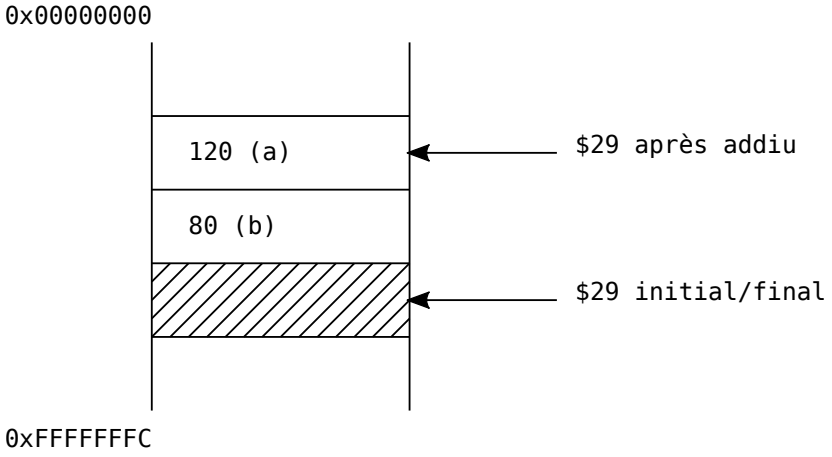
- Dessiner l'arbre de ce programme après la première passe
- Donner toutes les conditions, explicites ou implicites, vérifiées par la grammaire attribuée de MiniC pour ce programme
- Dessiner l'état de la pile au moment du `if`
- Donner le code assembleur mips correspondant à ce programme

<correction>
Arbre du programme après la passe de vérifications (ignorer le champ `nb_ops`) :

Voir Figure 1.

Conditions vérifiées par la grammaire attribuée :
- Règle 1.4 (6 occurrences)
- Règle 1.7, les 2 conditions (1 occurrence)
- Règle 1.12 (1 occurrence)
- Règle 1.17 (2 occurrences)
- Règle 1.24 (1 occurrence)
- Règle 1.36, implicite à travers `type_op_binaire` (2 occurrences : `>` et `-`)
- Règle 1.38 (1 occurrence)

État de la pile au moment du `if` :



Code assembleur mips :

```
.data

.asciiz "a = "
.asciiz " - b = "

.text

main:
    addiu $29, $29, -8
    ori   $8, $0, 120
    sw    $8, 0($29)
    ori   $8, $0, 80
    sw    $8, 4($29)
    lw    $8, 0($29)
    lw    $9, 4($29)
    slt   $8, $9, $8
    beq   $8, $0, _L1
```

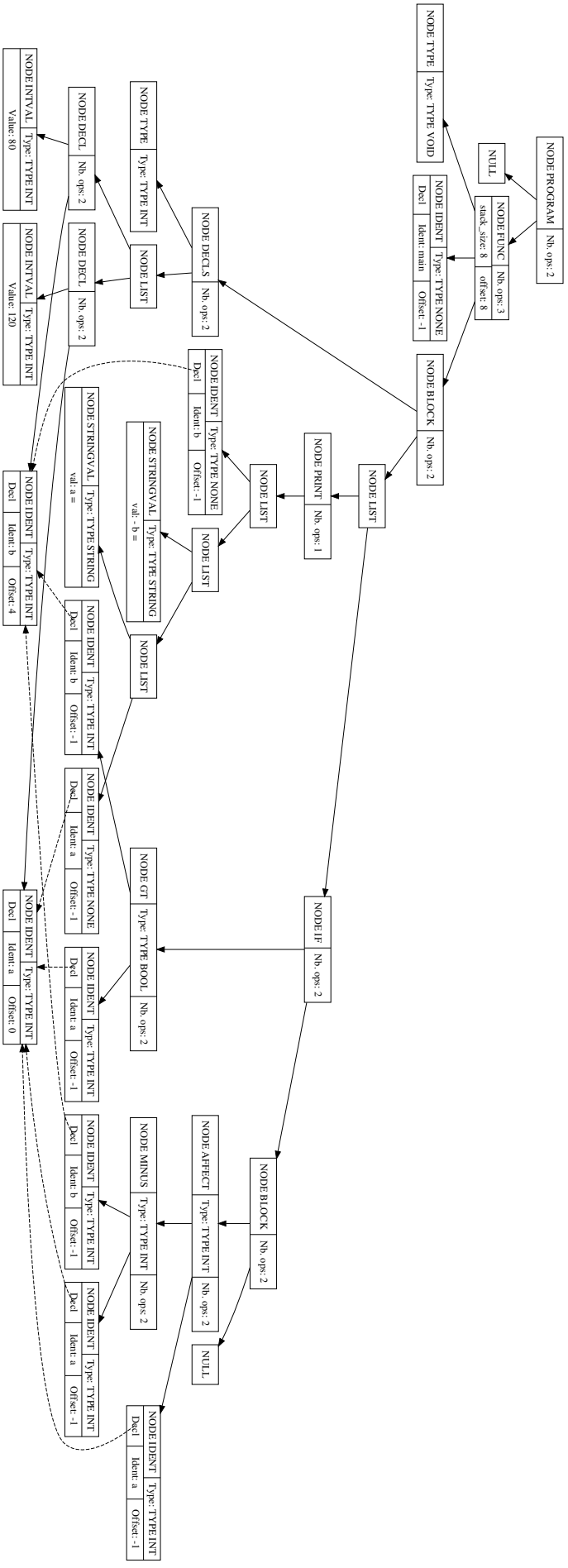


FIGURE 1 – Arbre du programme de l'exercice 4 après la passe de vérifications. Attention : l'ordre gauche / droite des fils n'est pas respecté sur le schéma (généré automatiquement), alors qu'il l'est pour l'analyse ; par exemple, les positions des noeuds `if` et `print` sont inversées

```
    lw    $8, 0($29)
    lw    $9, 4($29)
    subu  $8, $8, $9
    sw    $8, 0($29)
_L1:
    lui   $4, 0x1001
    ori   $4, $4, 0
    ori   $2, $0, 4
    syscall
    lw    $4, 0($29)
    ori   $2, $0, 1
    syscall
    lui   $4, 0x1001
    ori   $4, $4, 5
    ori   $2, $0, 4
    syscall
    lw    $4, 4($29)
    ori   $2, $0, 1
    syscall
    addiu $29, $29, 8
    ori   $2, $0, 10
    syscall
```

</**correction**>

Exercice 5 : Grammaire attribuée de MiniC

Donner, pour chacune des règles suivantes de la grammaire attribuée, un programme MiniC minimal ne vérifiant pas la condition de la règle :

- 1.4
- 1.12
- 1.16 (partie de la condition : $type = type_1$)
- 1.26

<**correction**>

Règle 1.4 :

```
1 void main() {
2   a;
3 }
```

Règle 1.12 :

```
1 void main() {
2   void a;
3 }
```

Règle 1.16 :

```
1 int a = false;
2 void main() {}
```

Règle 1.26 :

```
1  void main() {
2    while (1) {
3    }
4  }
```

</**correction**>

Exercice 6 : Génération de code assembleur Mips

Soit le programme MiniC suivant :

```
1  void main() {
2     int a = 136;
3     int b = 80;
4     while (a != b) {
5        if (a > b) {
6           a = a - b;
7        }
8        else {
9           b = b - a;
10       }
11    }
12    print(a);
13 }
```

- Que calcule ce programme ?
- Écrivez un programme assembleur mips correspondant

<**correction**>
Ce programme calcule et affiche le pgcd de `a` et `b`.

Code :

```
.data
.text

main:
    addiu $29, $29, -8
    ori   $8, $0, 136
    sw    $8, 0($29)
    ori   $8, $0, 80
    sw    $8, 4($29)
_L1:
    lw    $8, 0($29)
    lw    $9, 4($29)
    beq   $8, $9, _L2 # optimisation
    lw    $8, 0($29)
    lw    $9, 4($29)
    slt   $8, $9, $8
    beq   $8, $0, _L3
    lw    $8, 0($29)
    lw    $9, 4($29)
    subu  $8, $8, $9
    sw    $8, 0($29)
    j     _L4
```

```
_L3:
    lw    $8, 4($29)
    lw    $9, 0($29)
    subu  $8, $8, $9
    sw    $8, 4($29)
_L4:
    j     _L1
_L2:
    lw    $4, 0($29)
    ori   $2, $0, 1
    syscall
    addiu $29, $29, 8
    ori   $2, $0, 10
    syscall
```

</**correction**>

Exercice 7 : Génération de code assembleur Mips

Soit le programme MiniC suivant :

```
1  void main() {
2     int i;
3     int masque = 1;
4     int mot = 0x46;
5     int res = 0;
6     int temp = 0;
7
8     for (i = 0; i < 32; i = i + 1) {
9        temp = mot & masque;
10       temp = temp >>> i;
11       res = res + temp;
12       masque = masque << 1;
13    }
14    print(res);
15 }
```

- Qu'affiche ce programme ?
- Écrivez un programme assembleur mips correspondant

<**correction**>
Ce programme calcule le nombre de bits à 1 dans le mot `mot`.

Code :

```
.data
.text

main:
    addiu $29, $29, -20
    ori   $8, $0, 1
    sw    $8, 4($29)
    ori   $8, $0, 0x46
    sw    $8, 8($29)
```

```
    ori   $8, $0, 0
    sw    $8, 12($29)
    ori   $8, $0, 0
    sw    $8, 16($29)
    ori   $8, $0, 0
    sw    $8, 0($29)
_L1:
    lw    $8, 0($29)
    ori   $9, $0, 32
    slt   $8, $8, $9
    beq   $8, $0, _L2
    lw    $8, 8($29)
    lw    $9, 4($29)
    and   $8, $8, $9
    sw    $8, 16($29)
    lw    $8, 16($29)
    lw    $9, 0($29)
    srlv  $8, $8, $9
    sw    $8, 16($29)
    lw    $8, 12($29)
    lw    $9, 16($29)
    addu  $8, $8, $9
    sw    $8, 12($29)
    lw    $8, 4($29)
    ori   $9, $0, 1
    sllv  $8, $8, $9
    sw    $8, 4($29)
    lw    $8, 0($29)
    ori   $9, $0, 1
    addu  $8, $8, $9
    sw    $8, 0($29)
    j     _L1
_L2:
    lw    $4, 12($29)
    ori   $2, $0, 1
    syscall
    addiu $29, $29, 20
    ori   $2, $0, 10
    syscall
```

</**correction**>

Exercice 8 : Langages sous-contexte (totalement optionnel – for the fun of it)

Rappel : Une grammaire est dite sous-contexte si toutes ses règles sont de la forme $\alpha A\beta \to \alpha\omega\beta$, avec $\alpha, \beta \in V^*, A \in V_N, \omega \in V^+$.

Montrer que la définition au-dessus est équivalente à la définition suivante : une grammaire sous-contexte est une grammaire dans laquelle toutes les règles sont de la forme $\alpha \to \beta$, où $|\alpha| \leq |\beta|$ ($|.|$ désigne la longueur d'un mot, c'est-à-dire son nombre d'éléments terminaux et non terminaux)

<**correction**>
La preuve est dans le polycopié de référence de théorie des langages.
</**correction**>