Devoir de Programmation Fonctionnelle
Opérations sur des grammaires hors-contextes

Vincent BEUGNET - Adlane LADJAL

8 avril 2019

Enseignant : Christian CODOGNET

Sommaire

1 Introduction **3**

2 Définition des types OCAML **4**
2.1 Le type `lettre` 4
2.2 Le type `regle` 4

3 Les non terminaux accessibles **5**

4 Les non terminaux productifs **6**

5 La suppression des ϵ-règles **7**
5.1 Première méthode 7
5.2 Deuxième méthode 8

6 Conclusion **9**

7 Les tests **10**
7.1 Grammaires de tests 10
7.2 Résultats 10
7.2.1 Non terminaux accessibles 10
7.2.2 Règles productives 11
7.2.3 Non terminaux productifs 11
7.2.4 Suppression des ϵ-regles 11

8 Annexe **13**
8.1 Le fichier `type.ml` 13
8.2 Le fichier `utils.ml` 14
8.3 Le fichier `accessibles.ml` 18
8.4 Le fichier `productifs.ml` 20
8.5 Le fichier `epsilon.ml` 22
8.6 Le fichier `epsilon2.ml` 25

1 Introduction

Le présent devoir a été réalisé par Vincent Beugnet et Adlane Ladjal, au cours de notre deuxième année de formation d'Ingénieur Informatique à Sup Galilée. Il a été réalisé dans le cadre du cours Programmation Fonctionnelle. Le but de ce cours est de nous donner les bases de ce paradigme de programmation, en nous initiant au lambda-calcul, ainsi qu'aux algorithmes de typage et d'unification.

Nous devions concevoir un programme OCAML permettant de réaliser trois opérations sur des grammaires hors-contextes :

- Déterminer les non-terminaux accessibles.
- Déterminer les non-terminaux productibles.
- Eliminer les epsilon-règles.

Une grammaire hors-contexte est une grammaire où toutes les règles de production sont de la forme

$$X \to \alpha$$

où X est un symbole non terminal, et α un mot composé de symboles non terminaux et/ou terminaux. α peut aussi être le mot vide ϵ.

2 Définition des types OCAML

Nous devions commencer tout d'abord par définir les types OCAML.

2.1 Le type `lettre`

Tout d'abord nous avons définis un type `lettre`. Ce type représente à la fois les non terminaux, les terminaux et ϵ.

Nous aurions pu représenter les non terminaux, les terminaux et ϵ séparemment. Mais il nous aurait aussi fallu le type `lettre` pour pouvoir avoir une liste de `lettre` qui correspond à α dans une production $X \to \alpha$.

Dans ce cas-là nous aurions alors quelque chose comme cela :

```
1 type nonTerminal = NT of char ;;
2 type terminal = T of char ;;
3 type epsilon = Epsilon ;;
4 type lettre = terminal | nonTerminal | epsilon ;;
```

Mais une telle définition de `lettre` n'est pas possible en OCAML.

Dès lors nous avons opté pour cette définition :

```
1 type lettre = T of char | NT of char | Epsilon ;;
```

Ici, nous pouvons remarquer que pour les non-terminaux, par exemple, rien ne nous empêche de les représenter par une lettre minuscule, par exemple : `NT('a')`. Alors que d'usage, nous les représentons par une lettre majuscule. Il en va de même pour les terminaux avec `T('A')` par exemple.

Nous n'avons pas considéré ceci comme un problème. Si l'utilisateur utilise `NT('A')` et `NT('a')`, le programme interprétera ces deux symboles comme différents.

Nous ferons nous même attention à bien représenter les non terminaux par des majuscules et les terminaux par des minuscules, comme il est de rigueur.

2.2 Le type `regle`

Nous avons ensuite définis le type des règles de production. Nous avons considéré ce type comme composé de deux champs, le terme de gauche, un symbole non terminal, et le terme de droite une liste de symboles non terminaux et terminaux, ou une liste contenant simplement `Epsilon`.

```
1 type regle = Prod of (lettre * (lettre list)) ;;
```

Nous remarquons qu'avec cette définition, nous pouvons avoir un symbole terminal à gauche, ce qui ne correspond pas à une grammaire hors-contexte.

En revanche notre programme n'accepte en entrée uniquement des grammaires hors-contextes. Nous pouvons garantir le bon fonctionnement du programme uniquement pour des grammaires hors-contextes. Dès lors, si l'utilisateur rentre en entrée une grammaire qui n'est pas hors-contexte, le programme ne pourra fonctionner correctement. Il appartient à l'utilisateur de définir de lui même une grammaire hors-contexte.

3 Les non terminaux accessibles

On note $A \rightarrow^* \alpha$ le fait que α puisse être atteint après 0 (ou plus) dérivation(s) à partir de A. Une dérivation est l'application d'une règle de la grammaire.

Un non terminal B accessible à partir d'un non terminal A est tel que $B \in \alpha$ avec $A \rightarrow^* \alpha$

La première opération de notre programme est donc de récupérer les non-terminaux accessibles à partir d'un non terminal donné en entrée. Notre fonction récursive OCAML prendra alors en entrée deux paramètres :

- Une grammaire sous forme d'une liste de productions.
- Un symbole non terminal.

Nous avons décomposé cette fonction en deux étapes. Nous avons une fonction qui exécute la tâche principale décrite plus haut. Et une autre fonction qui se charge de récupérer tous les non terminaux qui sont produits par un symbole après une seule dérivation.

L'opération de récupération des non terminaux accessibles se réalise comme suit :

- La fonction récursive est initialisée avec une liste de non terminaux `parcours` qui contient uniquement dans un premier lieu le non terminal de départ. Et une liste de tous les non terminaux que contient la grammaire. Nous l'appelons `alphabet`.
- `parcours` est parcouru. Pour chaque non terminal T de `parcours`, nous récupérons tous les non terminaux que peux produire T avec une seule dérivation. Ceci nous fournit une liste de non terminaux `acc`.
- Ensuite nous concaténons cette liste dans le résultat retourné par la fonction. Nous concaténons aussi `acc` à `parcours` pour pouvoir calculer les non terminaux accessibles directement de ces nouveaux symboles. Puis nous supprimons le non terminal T de `alphabet` dont nous venons de réaliser le calcul pour éviter de boucler indéfiniment.

Le code est fourni en annexe.

4 Les non terminaux productifs

On dit que A est un non-terminal productif si on peut dériver au moins un mot terminal à partir de A. C'est à dire $A \rightarrow^* \alpha$ avec α un mot uniquement constitué de symboles terminaux.

L'algorithme communément utilisé pour déduire les non-terminaux productifs d'une grammaire est le suivant :

Algorithm 1 Calcul des non-terminaux productifs d'une grammaire

Soit $G = (V, T, P, S)$ une grammaire hors-contexte.

$i = 0$
$V_0 = \{X \in V | X \rightarrow \alpha, \alpha \in T^*\}$
repeat
  $i = i + 1$
  $V_i = V_{i-1} \cup \{X \in V | X \leftarrow \alpha, \alpha \in (T \cup V_{i-1})^*\}$
until $V_i = V_{i-1}$
return V_i

Nous avons adapté cet algorithme afin d'effectuer un chaînage arrière recursif en OCAML. Notre choix s'est posé sur un chaînage arrière car il permet à la fois une économie en place (il est récursif terminal) et une économie de calcul par rapport à un chaînage avant.

Pour récupérer les non-terminaux d'une grammaire, nous parcourons une par une les règles de production de la grammaire. Pour chaque règle, si tous les non-terminaux de son membre droit ont déjà été marqués comme productifs ou que le membre droit ne contient pas de non-terminaux, alors le membre gauche est marqué comme productif.

La fonction récursive `non_terminaux_productibles_rec grammaire regles prec acc` présente deux cas d'action différents :

Cas 1 La liste des `regles` n'est pas vide. On récupère alors la première règle et on verifie si le membre de gauche n'est pas déjà marqué et si tous les non-terminaux du membre droit sont déjà marqués (tous les éléments d'une liste vide sont marqués). Si tel est le cas, on marque le membre de gauche. Dans tous les cas, on passe ensuite récursivement à la règle suivante.

Cas 2 La liste des `regles` est vide. On récupère la liste des productions dont le membre gauche n'est pas marqué. Si la liste est vide ou qu'elle est égale aux règles de l'itération précédente, on renvoit les non-terminaux marqués. Sinon, on fait un appel récursif avec cette liste comme règles à vérifier.

Un exemple avec la grammaire [Prod(NT('A'), [NT('F'),T('g')]) ; Prod(NT('B'), [T('j') ;NT('E')]) ; Prod(NT('F'), [T('b')])] correspondant à $(A \rightarrow Fg, B \rightarrow jE, F \rightarrow b)$ nous renverra la liste de non-terminaux [NT('A'), NT('F')].

5 La suppression des ϵ-règles

Les ϵ-règles sont des règles de la forme $A \to \epsilon$ où A est un symbole non terminal, et ϵ le mot vide.

Le but de l'opération de suppression des ϵ-règles est de passer d'une grammaire hors-contexte G quelconque à une grammaire ϵ-libre G', c'est-à-dire ne contenant pas d'ϵ-règles.

La grammaire G' devra dériver exactement le même langage que la grammaire G. Seule différence : la grammaire G' ne pourra générer le mot vide.

Pour cette opération nous avons réalisé deux méthodes. Après avoir terminé la première méthode, nous nous sommes rendus compte que nous l'avons pensé de manière itérative. Nous nous sommes lancés alors dans une autre implémentation qui nécessite moins de calculs intermédiaires.

Nous n'avons pas amélioré la première et nous nous sommes concentrés sur la deuxième méthode. Néanmoins nous présentons ici les deux méhodes.

5.1 Première méthode

Notre fonction récursive OCAML prendra alors en paramètre la grammaire hors-contexte que l'on souhaite transformer en une grammaire ϵ-libre.

Elle se déroule comme suit :

- En premier lieu nous récupérons dans la liste `ntpe` les symboles non terminaux qui produisent le mot vide.
- Nous parcourons la liste `ntpe`. Puis pour chacun de ces symboles nous supprimons les ϵ-règles associées au symbole.

Lorsque nous supprimons les ϵ-règles associées à un symbole T, nous devons distinguer deux cas :

Cas 1 Le cas où seuls des règles de la forme $T \to \epsilon$ et $T \to \theta$ (avec θ contenant un nombre indéfini de fois le symbole T uniquement) apparaissent. Dans ce cas, il faut tout simplement supprimer le symbole T de la grammaire et ne réaliser aucune autre opération. En effet, T ne produit rien, et est donc inutile à la grammaire.

Cas 2 Le cas où la règle $T \to \epsilon$ apparaît avec d'autres règles de la forme $T \to \alpha$, α contenant à la fois des non terminaux et des terminaux. Dans ce cas nous supprimons la règle $T \to \epsilon$, et pour toutes les autres règles de la grammaire contenant le symbole T dans le membre de droite nous appliquons cette procédure :

Si nous avons une règle de la forme $A \to \alpha T \beta$, avec α et β contenant à la fois des non terminaux et des terminaux (ils peuvent aussi contenir le symbole T), alors il faut rajouter la règle $A \to \alpha\beta$ dans la grammaire.

Dans le cas 2, pour rajouter les règles lorsque l'on trouve le symbole à traiter dans le membre de droite d'une production, nous procédons de la sorte :

Nous générons une liste des positions dans le membre de droite que prend le symbole à traiter. Par exemple dans la règle $S \to aSbScS$, nous aurons la liste suivante :

`[1;3;5]`

Puis à partir de cette liste nous générons une liste de toutes les combinaisons possibles.

Avec l'exemple nous aurons : `[[1]; [3]; [5]; [1; 3]; [1; 5]; [3; 5]; [1; 3; 5]]`

Puis nous parcourons cette dernière liste et nous ajoutons les règles sans les symboles non terminaux aux positions données.

Donc toujours avec le même exemple, lorsque nous tombons sur la liste `[1; 5]` à traiter, il faut ajouter la règle $S \rightarrow abSc$.

5.2 Deuxième méthode

Pour cette méthode, plus simple, nous appliquons à la lettre l'algorithme dans la partie de la méthode 1, que nous remettons ici :

Lorsque nous supprimons les ϵ-règles associées à un symbole T, nous devons distinguer deux cas :

Cas 1 Le cas où seuls des règles de la forme $T \rightarrow \epsilon$ et $T \rightarrow \theta$ (avec θ contenant un nombre indéfini de fois le symbole T uniquement) apparaissent. Dans ce cas, il faut tout simplement supprimer le symbole T de la grammaire et ne réaliser aucune autre opération. En effet, T ne produit rien, et est donc inutile à la grammaire.

Cas 2 Le cas où la règle $T \rightarrow \epsilon$ apparaît avec d'autres règles de la forme $T \rightarrow \alpha$, α contenant à la fois des non terminaux et des terminaux. Dans ce cas nous supprimons la règle $T \rightarrow \epsilon$, et pour toutes les autres règles de la grammaire contenant le symbole T dans le membre de droite nous appliquons cette procédure :

Si nous avons une règle de la forme $A \rightarrow \alpha T \beta$, avec α et β contenant à la fois des non terminaux et des terminaux (ils peuvent aussi contenir le symbole T), alors il faut rajouter la règle $A \rightarrow \alpha\beta$ dans la grammaire.

Si après ce calcul nous avons une règle de la forme $Prod(NT(\_), [])$, nous la remplaçons par $Prod(NT(\_), [Epsilon])$.

Nous appliquons cet algorithme tant qu'il y a des $\epsilon - regles$ dans la grammaire.

Avant de réaliser cela, nous récupérons bien évidemment l'ensemble des non terminaux qui produisent ϵ, mais nous réalisons surtout un pré-traitement : nous supprimons les règles de la forme S -> S.

6 Conclusion

Ce devoir nous a permis de nous familiariser un peu plus avec la programmation fonctionnelle au travers du langage fonctionnel OCAML. Il nous a permis d'utiliser les notions que nous avons apprises en Théorie des Langages, mais aussi en Compilation, pour les appliquer dans ce projet. Il nous a aussi permis d'utiliser une méthode de chainage, que nous avons pû découvrir en Prolog. Finalement, il nous a aussi permis de nous améliorer dans notre méthode de travail en groupe, à parallèliser le travail et à corriger mutuellement nos erreurs.

7 Les tests

7.1 Grammaires de tests

Voici nos grammaires de tests :

```
#use "tests.ml";;

let accTest1 = non_terminaux_accessibles grammaireTest1 (NT('A'));;
let accTest2 = non_terminaux_accessibles grammaireTest1 (NT('F'));;
let accTest3 = non_terminaux_accessibles grammaireTest2 (NT('S'));;
let accTest4 = non_terminaux_accessibles grammaireTest3 (NT('U'));;

let prodTest1 = regles_productives grammaireTest1;;
let prodTest2 = regles_productives grammaireTest2;;
let prodTest3 = regles_productives grammaireTest3;;
let prodTestEpsilon = regles_productives grammaireEpsilon;;

let termesProdTest1 = non_terminaux_productifs grammaireTest1;;
let termesProdTest2 = non_terminaux_productifs grammaireTest2;;
let termesProdTest3 = non_terminaux_productifs grammaireTest3;;
let termesProdTestEpsilon = non_terminaux_productifs grammaireEpsilon;;

let epsTest1 = supprimer_toutes_epsilon_regles2 grammaireTest1;;
let epsTest2 = supprimer_toutes_epsilon_regles2 grammaireTest2;;
let epsTest3 = regles_productives grammaireTest3;;
let epsTestEpsilon = supprimer_toutes_epsilon_regles2 grammaireEpsilon;;
```

Gramamire de tests

7.2 Résultats

7.2.1 Non terminaux accessibles

accTest1

```
[NT 'D'; NT 'E'; NT 'B'; NT 'A']
```

accTest2

```
[NT 'F']
```

accTest3

```
[NT 'B'; NT 'S']
```

accTest4

```
[NT 'T'; NT 'U']
```

7.2.2 Règles productives

prodTest1

```
[Prod (NT 'Z', [Epsilon]); Prod (NT 'A', [NT 'B'; T 'c']);
 Prod (NT 'B', [T 'd']); Prod (NT 'F', [T 'g'; T 'a'])]
```

prodTest2

```
[Prod (NT 'S', [T 'a'; NT 'S'; NT 'B'; NT 'S'; T 'b'; T 'd'; NT 'S']);
 Prod (NT 'A', [T 'c'; NT 'S']); Prod (NT 'S', [NT 'S']);
 Prod (NT 'B', [NT 'S']); Prod (NT 'B', [Epsilon]);
 Prod (NT 'S', [Epsilon]); Prod (NT 'D', [Epsilon]);
 Prod (NT 'F', [T 'g'; T 'a'])]
```

prodTest3

```
[Prod (NT 'U', [NT 'T'; T 'a'; NT 'U']);
 Prod (NT 'U', [NT 'T'; T 'a'; NT 'T']);
 Prod (NT 'V', [NT 'T'; T 'b'; NT 'V']);
 Prod (NT 'V', [NT 'T'; T 'b'; NT 'T']);
 Prod (NT 'T', [T 'a'; NT 'T'; T 'b'; NT 'T']);
 Prod (NT 'T', [T 'b'; NT 'T'; T 'a'; NT 'T']); Prod (NT 'T', [Epsilon])]
```

prodTestEpsilon

```
[Prod (NT 'S', [Epsilon])]
```

7.2.3 Non terminaux productifs

termesProdTest1

```
[NT 'A'; NT 'F'; NT 'B'; NT 'Z']
```

termesProdTest2

```
[NT 'A'; NT 'F'; NT 'D'; NT 'S'; NT 'B']
```

termesProdTest3

```
[NT 'S'; NT 'V'; NT 'U'; NT 'T']
```

termesProdTestEpsilon

```
[NT 'S']
```

7.2.4 Suppression des ϵ-regles

epsTest1

```
[Prod (NT 'A', [NT 'B'; T 'c']); Prod (NT 'B', [T 'c'; T 'c'; NT 'D']);
 Prod (NT 'B', [T 'd']); Prod (NT 'D', [NT 'E']);
 Prod (NT 'F', [T 'g'; T 'a'])]
```

epsTest2

```
[Prod (NT 'A', [T 'c']);
 Prod (NT 'S', [T 'a'; NT 'S'; NT 'B'; NT 'S'; T 'b'; T 'd']);
 Prod (NT 'S', [T 'a'; NT 'S'; NT 'B'; T 'b'; T 'd']);
 Prod (NT 'S', [T 'a'; NT 'S'; NT 'B'; T 'b'; T 'd'; NT 'S']);
 Prod (NT 'S', [T 'a'; NT 'B'; NT 'S'; T 'b'; T 'd']);
 Prod (NT 'S', [T 'a'; NT 'B'; T 'b'; T 'd']);
 Prod (NT 'S', [T 'a'; NT 'B'; T 'b'; T 'd'; NT 'S']);
 Prod (NT 'S', [T 'a'; NT 'B'; NT 'S'; T 'b'; T 'd'; NT 'S']);
 Prod (NT 'S', [T 'a'; NT 'S'; NT 'S'; T 'b'; T 'd']);
 Prod (NT 'S', [T 'a'; NT 'S'; T 'b'; T 'd']);
 Prod (NT 'S', [T 'a'; T 'b'; T 'd']);
 Prod (NT 'S', [T 'a'; T 'b'; T 'd'; NT 'S']);
 Prod (NT 'S', [T 'a'; NT 'S'; T 'b'; T 'd'; NT 'S']);
 Prod (NT 'S', [T 'a'; NT 'S'; NT 'S'; T 'b'; T 'd'; NT 'S']);
 Prod (NT 'S', [T 'a'; NT 'S'; NT 'B'; NT 'S'; T 'b'; T 'd'; NT 'S']);
 Prod (NT 'A', [T 'c'; NT 'S']); Prod (NT 'B', [NT 'S']);
 Prod (NT 'D', [NT 'E']); Prod (NT 'F', [T 'g'; T 'a'])]
```

epsTest3

```
[Prod (NT 'U', [NT 'T'; T 'a'; NT 'U']);
 Prod (NT 'U', [NT 'T'; T 'a'; NT 'T']);
 Prod (NT 'V', [NT 'T'; T 'b'; NT 'V']);
 Prod (NT 'V', [NT 'T'; T 'b'; NT 'T']);
 Prod (NT 'T', [T 'a'; NT 'T'; T 'b'; NT 'T']);
 Prod (NT 'T', [T 'b'; NT 'T'; T 'a'; NT 'T']); Prod (NT 'T', [Epsilon])]
```

8 Annexe

8.1 Le fichier `type.ml`

```
1 type nonTerminal = NT of char;;
2 type lettre = T of char | NT of char | Epsilon ;;
3 type regle = Prod of (lettre * (lettre list)) ;;
```

Listing 1: Fichier de définition des types

8.2 Le fichier `utils.ml`

```
#use "type.ml"


(*
 * Renvoie la liste 'liste' sans doublons
 *)
let  list_to_set liste =
    let rec list_to_set_rec liste ensemble =
        match liste with
        | [] -> ensemble
        | head::tail ->   if List.mem head ensemble then
                        list_to_set_rec tail ensemble
                    else
                        list_to_set_rec tail (head::ensemble)
    in list_to_set_rec liste []
    ;;


(*
 * Verifie si une liste est sous-liste d'une autre liste
 * @param deux listes
 * @return un booleen
 *)
let rec subset sub set =
    match sub with
    | [] -> true
    | h::t -> if (List.mem h set)
              then subset t set
              else false
    ;;


(*
 * Renvoie les non terminaux presents dans la regle 'production'
 * @param regle dont on souhaite determiner les non terminaux
 * @return une liste de non terminaux
 *)
let rec recuperer_non_terminaux_regle production =
    let rec recuperer_non_terminaux_regle_rec production res =
        match production with
        | Prod(nt, liste) ->    match liste with
                                | T(_)::tail | Epsilon::tail ->
                                ↪  recuperer_non_terminaux_regle_rec
                                ↪  (Prod(nt, tail)) res
                                | NT(x)::tail ->
                                ↪  recuperer_non_terminaux_regle_rec
                                ↪  (Prod(nt, tail)) (NT(x)::res)
                                | _ -> nt::res
```

```
45      in (list_to_set (recuperer_non_terminaux_regle_rec production []))
46      ;;
47
48
49  (*
50   * Renvoie les non terminaux produits par la regle 'production'
51   * @param regle dont on souhaite determiner les non terminaux produits
52   * @return une liste de non terminaux
53   *)
54  let rec non_terminaux_produits production =
55      match production with
56      | Prod(nt, h::t) -> begin
57          match h with
58          | NT(x) -> (NT(x))::(non_terminaux_produits (Prod(nt, t)))
59          | _ -> non_terminaux_produits (Prod(nt, t))
60          end
61      | _ -> []
62      ;;
63
64
65  (*
66   * Renvoie les non terminaux d'une grammaire
67   * @param grammaire une liste de regles
68   * @return une liste de non terminaux
69   *)
70  let rec recuperer_non_terminaux_grammaire grammaire =
71      match grammaire with
72      | [] -> []
73      | head::tail -> list_to_set
74                      ((recuperer_non_terminaux_regle head)
75                      @
76                      (recuperer_non_terminaux_grammaire tail))
77      ;;
78
79
80  (*
81   * Retire un symbole d'une liste de symboles donnee en parametre
82   * @param alphabet une liste de symboles non terminaux et/ou terminaux
83   * @return une liste de symboles non terminaux et/ou terminaux
84   *)
85  let retirer_terme terme alphabet =
86      let rec retirer_terme_rec terme alphabet res =
87          match alphabet with
88          | [] -> res
89          | head::tail -> if head = terme
90                          then retirer_terme_rec terme tail res
91                          else retirer_terme_rec terme tail (head::res)
92      in List.rev (retirer_terme_rec terme alphabet [])
93      ;;
```

```
94
95
96  (*
97   * Compte le nombre d'occurences d'un symbole 'terme' dans 'alphabet'
98   *)
99  let rec nombre_occurences terme alphabet =
100     match alphabet with
101     | [] -> 0
102     | head::tail -> if head = terme
103                     then 1 + (nombre_occurences terme tail)
104                     else nombre_occurences terme tail
105     ;;
106
107
108 (*
109  * Renvoie une liste contenant la position de chaque occurence de 'n' d
110  * dans 'liste'.
111  *)
112 let positions_valeurs n liste =
113     let rec positions_valeurs_rec n liste i =
114         match liste with
115         | [] -> []
116         | head::tail -> if head = n
117                         then i::(positions_valeurs_rec n tail (i + 1))
118                         else positions_valeurs_rec n tail (i + 1)
119     in positions_valeurs_rec n liste 0
120     ;;
121
122
123 (*
124  * Retire l'element d'indice 'n' de la liste 'liste'.
125  *)
126 let rec retirer_indice n liste =
127     match liste with
128     | [] -> []
129     | head::tail -> if n = 0
130                     then tail
131                     else head::(retirer_indice (n - 1) tail)
132     ;;
133
134
135 (*
136  * Retire les elements d'indice contenu dans la liste 'n' de la liste
    ↪  'liste'.
137  *)
138 let rec retirer_indices ns liste =
139     match ns with
140     | [] -> liste
141     | head::tail -> let moinsun x = x - 1 in
```

```
142                     retirer_indices (List.map moinsun tail) (retirer_indice
                        ↪  head liste)
143     ;;
144
145
146 (*
147  * Genere toutes les combinaisons de 'n' elements de la liste 'liste'.
148  *)
149 let rec combinaisons n liste =
150     match n with
151     | 0 -> [[]]
152     | _ ->  match liste with
153             | [] -> []
154             | head::tail -> let inserer_tete suite = head::suite in
155                                 (List.map inserer_tete (combinaisons (n -
                                    ↪  1) tail))
156                                 @
157                                 (combinaisons n tail)
158     ;;
159
160
161 (*
162  * Genere toutes les combinaisons possibles de la liste 'liste'.
163  *)
164 let toutes_combinaisons liste =
165     let rec toutes_combinaisons_rec n liste =
166         match n with
167         | 0 -> []
168         | _ -> (toutes_combinaisons_rec (n - 1) liste) @ (combinaisons n
                    ↪  liste)
169     in toutes_combinaisons_rec (List.length liste) liste
170     ;;
```

Fichier de définitions de fonctions d'opérations élémentaire sur les grammaires hors-contextes.

8.3 Le fichier accessibles.ml

```
#use "utils.ml";;


(*
 * Recupere tous les non terminaux
 * qui sont produits par le symbole 'terme' apres une seule dérivation
 * dans la liste de regles 'gram' (une grammaire).
 *)
let rec non_terminaux_accessibles_direct terme gram =
    match gram with
    | [] -> []
    | Prod(nt, droite)::tail -> if nt = terme
                                then
                                    list_to_set (
                                    recuperer_non_terminaux_regle (Prod(nt,
                                    ↪ droite))
                                    @
                                    (non_terminaux_accessibles_direct terme
                                    ↪ tail)
                                    )
                                else
                                    (non_terminaux_accessibles_direct terme
                                    ↪ tail)
;;


(*
 * Recupere les non-terminaux accessibles a
 * partir d'un non terminal 'depart' dans une
 * grammaire, une liste de regles, 'gram'.
 *)
let non_terminaux_accessibles gram depart =
    let rec non_terminaux_accessibles_rec gram parcours alphabet =
        match parcours with
        | [] -> []
        | head::tail -> let acc = (non_terminaux_accessibles_direct head
        ↪ gram) in
                            if (List.mem head alphabet)
                            then
                                list_to_set (
                                acc
                                @
                                non_terminaux_accessibles_rec gram (tail @
                                ↪ acc) (retirer_terme head alphabet) )
                            else
                                non_terminaux_accessibles_rec gram tail
                                ↪ alphabet
```

```
42      in non_terminaux_accessibles_rec gram [depart]
        ↪  (recuperer_non_terminaux_grammaire gram)
43  ;;
```

Récupération des non terminaux accessibles.

8.4 Le fichier `productifs.ml`

```
#use "accessibles.ml";;


(*
 * Renvoie toutes les regles d'une grammaire dont le membre gauche ne fait pas
 * partie d'une liste de non-terminaux donnés
 * @param une grammaire et une liste de non-terminaux
 * @return une liste de productions
 *)
let rec regles_restantes nonterminaux grammaire =
    match grammaire with
    | [] -> []
    | Prod(nt, liste)::tail -> if (List.mem nt nonterminaux)
                               then regles_restantes nonterminaux tail
                               else Prod(nt, liste)::(regles_restantes nonterminaux tail)
    ;;


(*
 * Recupere tous les non-terminaux productifs d'une grammaire
 * @param une grammaire
 * @return une liste de non-terminaux
 *)
let non_terminaux_productifs grammaire =
    let rec non_terminaux_productifs_rec grammaire regles prec acc =
        match regles with
        | Prod(nt, liste)::suite -> let non_terminaux = non_terminaux_produits (Prod(nt, liste)) in
                if ((not (List.mem nt acc)) && subset non_terminaux acc)
                then non_terminaux_productifs_rec grammaire suite prec (nt::acc)
                else non_terminaux_productifs_rec grammaire suite prec acc
        | [] -> let reste = (regles_restantes acc grammaire) in begin
                match reste with
                | [] -> acc
                | reste -> if (subset prec reste)
                           then acc
                           else non_terminaux_productifs_rec grammaire reste regles acc
                end
    in non_terminaux_productifs_rec grammaire grammaire grammaire []
    ;;


(*
 * Recupere toutes les regles productives d'une grammaire
```

```
44   * @param une grammaire
45   * @return une grammaire ne contenant que des règles productives
46   *)
47  let rec regles_productives grammaire =
48      match grammaire with
49      | [] -> []
50      | Prod(nt, liste)::suite -> let non_terminaux =
        ↪  non_terminaux_productifs grammaire in
51                                      if (subset
                                        ↪  (recuperer_non_terminaux_regle
                                        ↪  (Prod(nt, liste))) non_terminaux)
52                                      then (Prod(nt,
                                        ↪  liste))::(regles_productives suite)
53                                      else regles_productives suite
54      ;;
```

Récupération des non-terminaux productifs.

8.5 Le fichier `epsilon.ml`

```
#use "productifs.ml";;

(* Fichier de la premiere methode *)

(*
 * Renvoie true si la grammaire 'gram' contient uniquement des
 * regles de la forme A -> epsilon et/ou A -> A..A (uniquement des A à droite)
 *)
let rec epsilon_seul nt gram =
    match gram with
    | [] -> true
    | Prod(x, [Epsilon])::tail when x = nt -> epsilon_seul nt tail
    | Prod(x, droite)::tail when (x = nt && List.length droite = nombre_occurences x droite) -> epsilon_seul nt tail
    | Prod(x, _)::tail when x = nt -> false
    | _::tail -> epsilon_seul nt tail
;;


(*
 * Renvoie une liste des symboles non terminaux qui produisent
 * epsilon dans la grammaire 'gram'.
 * La liste renvoyee contient des elements de la forme '(nt, cas)'
 * avec 'nt' le symbole non terminal et cas la valeur renvoye par 'epsilon_seul'
 * avec comme parametre 'nt'.
 *)
let non_terminaux_produisent_epsilon gram =
    let rec non_terminaux_produisent_epsilon_rec gram gramCheck =
        match gram with
        | [] -> []
        | Prod(nt, [Epsilon])::tail -> (nt, (epsilon_seul nt gramCheck))::(non_terminaux_produisent_epsilon_rec tail gramCheck)
        | Prod(nt, _)::tail -> (non_terminaux_produisent_epsilon_rec tail gramCheck)
    in non_terminaux_produisent_epsilon_rec gram gram
;;


(*
 * Cas 1.
 * Retire un symbole 'terme' de toute la grammaire 'gram'.
 *)
let rec retirer_lettre_grammaire terme gram =
    match gram with
    | [] -> []
```

```
43      | Prod(nt, droite)::tail when (List.mem terme droite) -> (Prod(nt,
        ↪  retirer_terme terme droite))::(retirer_lettre_grammaire terme tail)
44      | head::tail -> head::(retirer_lettre_grammaire terme tail)
45  ;;
46
47
48  (*
49   * Cas 2.
50   * Rajoute les regles necessaires lorsque l'on a supprime une
    ↪  epsilon-regle.
51   * Par exemple pour la regle S -> aSTbScS, avec la
52   * suppression de S -> epsilon, nous ajouterons les regles,  :
53   * S -> aTBScS
54   * S -> aSTBcS
55   * S -> aSTBSc
56   * S -> aTBcS
57   * S -> aTBSc
58   * S -> aSTBc
59   * S -> aTBc
60   *)
61  let dupliquer terme regle =
62      match regle with
63      | Prod(nt, droite) ->   let rec dupliquer_rec positions regle =
64                                  match positions with
65                                  | [] -> [Prod(nt, droite)]
66                                  | head::tail -> Prod(nt, (retirer_indices
                                    ↪  head droite))::(dupliquer_rec tail
                                    ↪  regle)
67                              in dupliquer_rec (toutes_combinaisons
                                ↪  (positions_valeurs terme droite)) regle
68  ;;
69
70
71  (*
72   * Retire un symbole 'terme' d'une regle 'production'.
73   *)
74  let rec retirer_lettre_production terme production =
75      match production with
76      | Prod(nt, droite) -> Prod(nt, retirer_terme terme droite)
77      ;;
78
79
80  (*
81   * Retire d'une grammaire 'gram' les regles qui ne produisent rien,
82   * c'est-a-dire de la forme 'Prod(nt, [])'.
83   *)
84  let rec retirer_production_vide gram =
85      match gram with
86      | [] -> []
```

```
87      | Prod(nt, [])::tail -> retirer_production_vide tail
88      | head::tail -> head::(retirer_production_vide tail)
89      ;;
90
91
92  (*
93   * Retire l'epsilon-regle associe terme 'terme'
94   * (qui se trouve a gauche dans l'epsilon-regle) de la grammaire 'gram',
95   * en indiquant le cas 1 (true) ou 2 (false) avec 'eps_seul'.
96   * Si on est dans le cas 2, on ajoute les regles necessaires avec
     ↪  'dupliquer'.
97   *)
98  let rec epsilon_iteration (terme, eps_seul) gram =
99      match gram with
100     | [] -> []
101     | Prod(nt, [Epsilon])::tail when nt = terme -> epsilon_iteration
        ↪  (terme, eps_seul) tail
102     | Prod(nt, droite)::tail when (List.mem terme droite) ->
103         (if (eps_seul)
104         then
105             [retirer_lettre_production terme (Prod(nt, droite))]
106         else dupliquer terme (Prod(nt, droite)))
107         @
108         (epsilon_iteration (terme, eps_seul) tail)
109     | head::tail -> head::(epsilon_iteration (terme, eps_seul) tail)
110 ;;
111
112
113 (*
114  * Retire toutes les epsilon-regles de la grammaire 'gram'
115  *)
116 let supprimer_toutes_epsilon_regles gram =
117     let rec supprimer_toutes_epsilon_regles_rec ntpe gram =
118         match ntpe with
119         | [] -> gram
120         | head::tail -> supprimer_toutes_epsilon_regles_rec tail
            ↪  (epsilon_iteration head gram)
121     in list_to_set (retirer_production_vide
        ↪  (supprimer_toutes_epsilon_regles_rec
        ↪  (non_terminaux_produisent_epsilon gram) gram))
122 ;;
```

Elimination des epsilon-regles avec la première méthode..

8.6 Le fichier epsilon2.ml

```
#use "productifs.ml";;

(* Fichier de la deuxieme methode *)

(*
 * Renvoie true si la grammaire 'gram' contient uniquement des
 * regles de la forme A -> epsilon et/ou A -> A..A (uniquement des A à droite)
 *)
let rec epsilon_seul nt gram =
    match gram with
    | [] -> true
    | Prod(x, [Epsilon])::tail when x = nt -> epsilon_seul nt tail
    | Prod(x, droite)::tail when (x = nt && List.length droite = nombre_occurences x droite) -> epsilon_seul nt tail
    | Prod(x, _)::tail when x = nt -> false
    | _::tail -> epsilon_seul nt tail
;;


(*
 * Renvoie une liste des symboles non terminaux qui produisent
 * epsilon dans la grammaire 'gram'.
 * La liste renvoyee contient des elements de la forme '(nt, cas)'
 * avec 'nt' le symbole non terminal et cas la valeur renvoye par 'epsilon_seul'
 * avec comme parametre 'nt'.
 *)
let non_terminaux_produisent_epsilon gram =
    let rec non_terminaux_produisent_epsilon_rec gram gramCheck =
        match gram with
        | [] -> []
        | Prod(nt, [Epsilon])::tail -> (nt, (epsilon_seul nt gramCheck))::(non_terminaux_produisent_epsilon_rec tail gramCheck)
        | Prod(nt, _)::tail -> (non_terminaux_produisent_epsilon_rec tail gramCheck)
    in non_terminaux_produisent_epsilon_rec gram gram
;;


(*
 * Remplace toutes les regles de la forme 'Prod(x, [])' par 'Prod(x, [Epsilon])'.
 *)
let rec remplacer_vide_epsilon gram =
    let remplacer (Prod(nt, droite)) =
        if droite = []
```

```
42          then (Prod(nt, [Epsilon]))
43          else (Prod(nt, droite))
44      in List.map remplacer gram
45  ;;
46
47  (*
48   * Supprime les regles de la forme 'Prod(terme, [Epsilon])'.
49   *)
50  let rec supprimer_epsilon_regle terme gram =
51      match gram with
52      | Prod(nt, [Epsilon])::tail when nt = terme -> supprimer_epsilon_regle
        ↪  terme tail
53      | head::tail -> head::(supprimer_epsilon_regle terme tail)
54      | [] -> []
55  ;;
56
57  (*
58   * Supprime les regles de la forme 'Prod(terme, [terme])'.
59   *)
60  let rec nettoyer gram =
61      match gram with
62      | Prod(nt, liste)::tail when liste = [nt] -> nettoyer tail
63      | Prod(nt, liste)::tail -> (Prod(nt, liste))::(nettoyer tail)
64      | [] -> []
65  ;;
66
67  (*
68   * Renvoie vrai s'il existe une epsilon regle dans la grammaire gram.
69   *)
70  let rec existe_epsilon_regle gram =
71      match gram with
72      | Prod(_, [Epsilon])::_ -> true
73      | _::tail -> existe_epsilon_regle tail
74      | [] -> false
75  ;;
76
77  (*
78   * Cas 1.
79   * Retire un symbole 'terme' de toute la grammaire 'gram'.
80   *)
81  let rec retirer_lettre_grammaire terme gram =
82      match gram with
83      | [] -> []
84      | Prod(nt, droite)::tail when (List.mem terme droite) -> (Prod(nt,
        ↪  retirer_terme terme droite))::(retirer_lettre_grammaire terme tail)
85      | head::tail -> head::(retirer_lettre_grammaire terme tail)
86  ;;
87
88  (*
```

```
89    * Cas 2.
90    * Rajoute les regles necessaires lorsque l'on a supprime une
      ↪  epsilon-regle.
91    * Par exemple pour la regle S -> aSTbScS, avec la
92    * suppression de S -> epsilon, nous ajouterons les regles,  :
93    * S -> aTBScS
94    * S -> aSTBcS
95    * S -> aSTBSc
96    * S -> aTBcS
97    * S -> aTBSc
98    * S -> aSTBc
99    * S -> aTBc
100   *)
101   let dupliquer2 terme gram =
102       let rec dupliquer_rec2 terme gram nouvelle_gram =
103           match gram with
104           | [] -> remplacer_vide_epsilon nouvelle_gram
105           | Prod(nt, liste)::suite when (List.mem terme liste) ->
106               let nouvelle_regle =
107                   let positions = positions_valeurs terme liste in
108                       let rec regle_a_ajouter pos =
109                           match pos with
110                           | [] -> liste
111                           | head::tail ->
112                               let nouvelle_liste = retirer_indice head liste
                                  ↪  in
113                                   if List.mem (Prod(nt, nouvelle_liste))
                                      ↪  nouvelle_gram
114                                   then regle_a_ajouter tail
115                                   else nouvelle_liste
116                       in Prod(nt, (regle_a_ajouter positions))
117               in
118                   if nouvelle_regle = (Prod(nt, liste))
119                   then dupliquer_rec2 terme suite nouvelle_gram
120                   else dupliquer_rec2 terme (nouvelle_regle::gram)
                      ↪  (nouvelle_regle::nouvelle_gram)
121           | _::suite -> dupliquer_rec2 terme suite nouvelle_gram
122       in dupliquer_rec2 terme gram gram
123   ;;
124
125   let rec epsilon_iteration2 (terme, eps_seul) gram =
126       let gram_sans_epsilon = supprimer_epsilon_regle terme gram in
127           if eps_seul
128           then retirer_lettre_grammaire terme gram_sans_epsilon
129           else dupliquer2 terme gram_sans_epsilon
130   ;;
131
132
133   let rec supprimer_toutes_epsilon_regles2 gram =
```

```
134     let resultat =
135         let rec supprimer_toutes_epsilon_regles_rec2 ntpe gram =
136             match ntpe with
137             | [] -> gram
138             | head::tail -> supprimer_toutes_epsilon_regles_rec2 tail
                ↪  (epsilon_iteration2 head gram)
139         in supprimer_toutes_epsilon_regles_rec2
            ↪  (non_terminaux_produisent_epsilon gram) (nettoyer gram)
140     in
141         if (existe_epsilon_regle resultat)
142         then supprimer_toutes_epsilon_regles2 resultat
143         else resultat
144 ;;
```

Elimination des epsilon-regles avec la deuxième méthode.