Programmation fonctionnelle

Grammaire de réécriture avec structures de traits

Travail à rendre jusqu'au 10.1.2001

Une grammaire de réécriture consiste en des productions $A \to \alpha$ avec α étant une suite de symboles terminaux (mots) et non-terminaux (catégories). Dans une séquence $A_1 A_2 \ldots A_n$, une catégorie A_i peut être remplacée par une séquence α_i si une production $A_i \to \alpha_i$ existe.

Dans ce travail nous allons considérer des grammaires de réécriture avec des structures de traits comme symboles non-terminaux.

Exemple:

$$\begin{bmatrix} cat & S \end{bmatrix} \to \begin{bmatrix} cat & NP \\ per & 1 \\ num & 2 \end{bmatrix} \begin{bmatrix} cat & VP \\ per & 1 \\ num & 2 \end{bmatrix}$$

$$\begin{bmatrix} cat & NP \\ per & p3 \\ num & sg \end{bmatrix} \to Jean$$

$$\begin{bmatrix} cat & VP \\ per & 1 \\ num & 2 \end{bmatrix} \to \begin{bmatrix} cat & V \\ per & 1 \\ num & 2 \end{bmatrix}$$

$$\begin{bmatrix} cat & V \\ per & p3 \\ num & sg \end{bmatrix} \to rit$$

Les nombres $1, 2, \ldots$ sont des variables.

Pour les grammaires considérées ici, nous supposons que

- les grammaires ne contiennent pas de productions $A \to \epsilon$ avec A étant une structure de traits et ϵ étant le mot vide.
- les valeurs des traits sont atomiques, c'est-à-dire la valeur d'un trait est une valeur atomique (un nom) ou une variable (un nombre).

Deux structures de traits A_1, A_2 avec des ensembles de variables disjoints sont unifiables si

1. pour chaque trait de A_1,
   - ce trait n'existe pas en A_2 ou
   - ce trait a la même valeur dans A_2 ou
   - la valeur de ce trait dans A_2 est une variable, et dans ce cas on garde cette variable et la valeur du trait dans A_1 sur une liste de liaisons, ou
   - la valeur de ce trait dans A_2 est un nom et celle dans A_1 est une variable. Dans ce cas on garde cette variable et la valeur du trait dans A_2 sur la liste de liaisons.
2. après avoir inspecté tous les traits de A_1 de cette façon, la liste des liaisons est telle que
   - aucune variable apparait avec deux valeurs différentes qui sont toutes les deux pas de variables, et
   - si la valeur d'une variable sur la liste est une variable, les deux variables n'apparaissent pas avec deux valeurs differentes qui sont toutes les deux pas de variables.

Dans une dérivation, un pas de dérivation $\alpha_1 X \alpha_2 \Rightarrow \alpha_1^{\sim} A_1^{\sim} A_2^{\sim} \ldots A_n^{\sim} \alpha_2^{\sim}$ (avec X une structure de traits, α_1, α_2 des séquences de structures de traits et symboles terminaux) est possible si la grammaire contient une production $A \to A_1 A_2 \ldots A_n$ tel que

- X et A sont unifiable avec une liste de laisons L, et
- pour $i \in \{1, 2, \ldots, n\}$: si A_i est un symbole terminale, $A_i = A_i^{\sim}$, et si A_i est une structure de trait, $A_i^{\sim}$ est le résultat de remplacer dans A_i toutes les occurrences de variables ayantes une valeur dans L par leur valeur, et
- $\alpha_i^{\sim}$ $(1 \leq i \leq 2)$ est le résultat de remplacer dans α_i toutes les occurrences de variables ayantes une valeur dans L par leur valeur.

Nous supposons que la séquence des structures de traits déjà dérivée et une nouvelle production de la grammaire contiennent des ensembles de variables disjoints.

Exemple d'une dérivation:

$$\begin{bmatrix} cat & S \end{bmatrix} \Rightarrow \begin{bmatrix} cat & NP \\ per & 1 \\ num & 2 \end{bmatrix} \begin{bmatrix} cat & VP \\ per & 1 \\ num & 2 \end{bmatrix} \Rightarrow Jean \begin{bmatrix} cat & VP \\ per & p3 \\ num & sg \end{bmatrix}$$

$$\Rightarrow Jean \begin{bmatrix} cat & V \\ per & p3 \\ num & sg \end{bmatrix} \Rightarrow Jean\ rit$$

Dans le travail à faire il s'agit

- d'écrire les fonctions nécessaires pour structures de traits et
- d'écrire un parser simple pour des grammaires de réécriture avec structures de traits comme décrit ci-dessus.

1. Structures de traits:

   Supposons qu'une structure de traits en LISP consiste en une liste dont chaque élément est une liste qui contient un trait et sa valeur:

   ```
   ((cat NP) (per 1) (num 2))
   ```

   (a) Pour assurer qu'une nouvelle production ne contient pas de variable déjà utilisée, il faut remplacer chaque variable v dans la production par $v + n$ avec n étant la plus grande variable déjà utilisée.

   Ecrivez une fonction `repl-var` qui prend n et une structure de traits comme arguments et qui effectue ce remplacement:

   ```
   > (repl-var 5 '((cat NP) (per 1) (num 2)))
   ((CAT NP) (PER 6) (NUM 7))
   ```

   (b) Le résultat de l'unification de deux structures de traits est `nil` si l'unification n'est pas possible et il est une liste avec la nouvelle structure de traits comme premier élément et la liste des liaisons comme deuxième élément si unification est possible. La nouvelle structure de traits est l'ensemble de tous les traits de les deux structures de traits avec leur nouvelles valeurs.

   Ecrivez une fonction `unify` qui fait l'unification de deux structures de traits:

   ```
   > (unify '((cat 1) (per 2) (num sg)) '((cat 3) (per p3) (num 4)))
   (((CAT 1) (PER P3) (NUM SG)) ((4 SG) (2 P3) (3 1)))
   > (unify '((cat NP)) '((cat VP)))
   NIL
   ```

   (c) Ecrivez une fonction `apply-var` qui prend une structure de traits et une liste de liaisons comme arguments et qui remplace dans la structure de traits chaque variable qui apparait (comme premier élément d'un de ses éléments) dans la liste des liaisons par sa valeur:

   ```
   > (apply-var '((cat 2) (per 3) (num sg)) '((2 1) (3 p3)))
   ((CAT 1) (PER P3) (NUM SG))
   ```

2. La grammaire de réécriture:

   (a) L'arbre de dérivation doit être rendu dans un format

   ```
   (<pos1> <cat1> ... <posn> <catn>)
   ```

   <posi> est une liste de nombres qui marque la position d'un noeud ((0) pour la racine, (0 0) pour le noeud fils le plus à gauche de la racine, (0 1) pour le deuxième noeud fils etc.).

   <cati> est l'étiquette du noeud à la position <posi>.

Ecrivez une fonction `tree-out` qui prend un tel arbre comme argument et qui imprime le même arbre un noeud (`<posi>` suivi de `<cati>`) par ligne. Comme valeur de retour la fonction rend `T`:

```
> (tree-out '((0) ((cat S)) (0 0) ((cat NP) (per p3) (num sg))
              (0 0 0) Jean (0 1) ((cat VP) (per p3) (num sg))
              (0 1 0) ((cat V) (per p3) (num sg)) (0 1 0 0) rit))
(0) ((cat S))
(0 0) ((cat NP) (per p3) (num sg))
(0 0 0) JEAN
(0 1) ((cat VP) (per p3) (num sg))
(0 1 0) ((cat V) (per p3) (num sg))
(0 1 0 0) RIT
T
```

(b) La grammaire va être notée en une liste dont chaque élément représente une production. Une production est une liste dont le `car` est la partie gauche de la production et le `cdr` est la liste des éléments de la partie droite.
Exemple:

```
((((cat S)) ((cat NP)(per 1)(num 2)) ((cat VP)(per 1)(num 2)))
 (((cat NP)(per p3)(num sg)) Jean)
 (((cat VP)(per 1)(num 2)) ((cat V)(per 1)(num 2)))
 (((cat V)(per p3)(num sg)) rit))
```

Elargissez cette grammaire telle qu'elle peut analyser les phrases suivantes:

(1) a. Jean mange une pomme
  b. Jean voit un vieil ami
  c. un vieux type arrive
  d. Marie aime les pommes rouges

La grammaire ne doit pas accepter les phrase suivantes:

(2) a. *Jean mange un pomme
  b. *Jean mangent une pomme
  c. *Jean voit un vieux ami
  d. *Marie aime les rouges pommes

Donnez un nom (avec `setq`) à votre nouvelle grammaire.

(c) Parsing : le parser va procéder d'une façon top-down.
Dans chaque pas, on a la séquence des symboles (structures de traits et mots terminaux) déjà dérivée `derived` et la liste des mots qu'il faut encore générer `goal`.
Le parser compare le premier symbole de `derived` avec le premier mot dans `goal`. Si les deux sont égaux, il peut supprimer les deux.
Si le premier symbole de `derived` est un symbol terminal qui n'est pas égal au premier mot de `goal`, le parser s'est trompé, c'est-à-dire il a choisi les mauvaises productions. Il doit poursuivre d'autres alternatives.
Si le premier symbole dans `derived` est une structure de trait, le parser essaie d'appliquer une production à `derived` en remplaçant le premier symbole de `derived`.
Exemple:

```
derived: ((cat S))
goal: Jean rit

derived: ((cat NP) (per 1) (num 2)) ((cat VP) (per 1) (num 2))
goal: Jean rit

derived: Jean ((cat VP) (per p3) (num sg))
goal: Jean rit
```

```
derived: ((cat VP) (per p3) (num sg))
goal: rit

derived: ((cat V) (per p3) (num sg))
goal: rit

derived: rit
goal: rit
```

Ecrivez une fonction `parse` qui prend une grammaire, une liste des mots et une valeur initiale pour `derived` comme arguments. La valeur de default de `derived` est `(((cat S)))`. `parse` retourne l'arbre de dérivation pour chaque analyse de la liste des mots possible avec la grammaire. Si aucune analyse n'est possible, `parse` retourne `nil`.

Exemple: Supposons que la grammaire sous (b) s'appelle `gram`.

```
> (parse gram '(Jean rit))
(0) ((cat S))
(0 0) ((cat NP) (per p3) (num sg))
(0 0 0) JEAN
(0 1) ((cat VP) (per p3) (num sg))
(0 1 0) ((cat V) (per p3) (num sg))
(0 1 0 0) RIT
T
> (parse gram '(Jean rient))
NIL
```