2 Parsing top-down

Idée : l'analyse commence avec le symbole initiale S et on cherche une dérivation qui permet de générer une ou plusieures analyses pour la phrase en question.

Si pour l'instant on suppose qu'il n'existe qu'au plus une seule analyse pour une phrase, on peut décrire l'algorithme `recognize-td` qui décide si une phrase fait partie de la langue générée par une grammaire donnée comme suit:

Algorithme recognize-td

`top-down / gauche-droit / déterministique`

```
Données globales :  un dictionnaire et une grammaire G = ⟨V,T,S,P⟩
Input :  une phrase w = w_1 ... w_n ,  n ≥ 0.
Output :  true/false
Variables locales :
 word        position du mot actuel
 derivation  suite des symboles de V ∪ T.  Valeur initiale:  [S].
Méthode :
   repeat:
      if derivation = [] ∧ word = n + 1
      then return(true)
      else
         if derivation = [] ∨ first(derivation) ∈ T
         then return(false)
         else derivation := reduce(expand(derivation))
```

Représentaion de la grammaire : les symboles terminales de G sont des préterminaux (det, n, v, ...). Le dictionnaire lie les mots de la langue aux préterminaux.

Pour une suite de symboles comme *derivation*, `first` retourne le premier symbole tandis que `rest` retourne la suite privée de son premier symbole.

`expand` fait un pas de dérivation en remplacant le premier symbole dans *derivation* par la partie droite d'une production.

`reduce` supprime autant de symboles terminaux que possible à gauche de *derivation* si ces symboles sont des préterminaux du mot actuel, mot suivant etc.

Pour tout ce qui suit nous supposons que dans une procédure les variables locales de la fonction d'appel sont accessibles (p.e. *word* et *derivation* sont comme variables gobales dans `expand` et `reduce`).

Procédure expand

```
Input :  une dérivation d
Output :  une nouvelle dérivation
Méthode :
   choisir une production p ∈ P avec left(p)=first(d)
   return(right(p) + rest(d))
```

`left` donne la partie gauche, `right` la partie droite d'une production.

`first` donne le premier élément d'une séquence, `rest` la séquence privée de son premier élément. + appliqué à deux séquences en donne la concaténation.

Procédure reduce

```
Input :  une dérivation d
Output :  une nouvelle dérivation
```

```
Effet supplémentaire :  modification de la variable word
Méthode :
   if first(d) ∈ T ∧ w_word ∈ first(d)
   then word := word + 1
      reduce(rest(d))
   else return(d)
```

Un préterminal est considéré comme un ensemble de mots. $w \in t$ pour un $t \in T$ veut dire que t est une catégorie préterminale de w.

Exemple 2 *G soit une grammaire avec symbole initiale S et les productions suivantes :*

1 $S \rightarrow NP\ VP$
2 $NP \rightarrow det\ n$
3 $VP \rightarrow v$

Analyse top-down avec recognize-td de la phrase un homme rit:

$w_{word} \ldots w_n$	*derivation*	*word*	*action*	*production*
un homme rit	*S*	*1*		
un homme rit	*NP VP*	*1*	*expand*	*1*
un homme rit	*det n VP*	*1*	*expand*	*2*
homme rit	*n VP*	*2*	*reduce*	
rit	*VP*	*3*	*reduce*	
rit	*v*	*3*	*expand*	*3*
–	–	*4*	*reduce*	

Pour faire un algorithme de parsing de recognize-td (c'est-à-dire pour obtenir une analyse), on peut transformer la dérivation *derivation* en analyse. A lieu de supprimer qc. dans `reduce`, on modifie seulement la position actuelle dans l'analyse. Cette position compte les symboles terminaux et non-terminaux mais pas les parenthèses ajoutées à l'analyse.

Exemple 3 *Prenons la grammaire de l'exemple précédant.*

$w_{word} \ldots w_n$	*derivation*	*position*
un homme rit	*S*	*1*
un homme rit	*[S [NP VP]]*	*2*
un homme rit	*[S [[NP [det n]] VP]]*	*3*
homme rit	*[S [[NP [[det un] n]] VP]]*	*5*
rit	*[S [[NP [[det un] [n homme]]] VP]]*	*7*
rit	*[S [[NP [[det un] [n homme]]] [VP [v]]]]*	*8*
–	*[S [[NP [[det un] [n homme]]] [VP [[v rit]]]]]*	*10*

Cette modification nécessite une nouvelle procédure `expand` et une nouvelle procédure `reduce`. Au lieu de *derivation* on va utiliser une variable *structure* (pour dérivation et analyse). Une deuxième variable *position* indique la position actuelle dans *structure*.

Procédure expand

```
Input :  -
Output :  true
Effet supplémentaire :  modification des variables globales structure et position
Méthode :
   choisir une production p ∈ P avec left(p)= s_position
   remplacer structure_position par [structure_position[right(p)]]
   position := position + 1
   return(true)
```

Pour une structure s, s_n donne l'élément dans s à la position n, autrement dit le n-ième élément (sans compter les parenthèes).

Procédure reduce

```
Input :  -
Output :  true
Effet supplémentaire :  modification des variables structure, word et position
Méthode :
   repeat:
      if structure_position ∈ T ∧ w_word ∈ structure_position
      then remplacer structure_position par [structure_position w_word]
        position := position + 2
        word := word + 1
      else return(true)
```

Le nouvel algorithme principal :

Algorithme parse-td

`top-down / gauche-droit / déterministique`

```
Données globales :  un dictionnaire et une CFG G = ⟨V, T, S, P⟩
Input :  une phrase w = w_1 ... w_n ,  n ≥ 0.
Output :  une structure pour w/false
Variables locales :
 word        position du mot actuel.  Valeur initiale :  1.
 structure   suite des symboles de N ∪ T ∪ {[,]}.  Valeur initiale :  [S].
 position    un pointeur vers un symbole dans structure.  Valeur initiale :  1.
Méthode :
   repeat:
      if position = |structure| + 1  ∧  word = n + 1
      then return(structure)
      else
         if position = |structure| + 1  ∨  structure_position ∈ T
         then return(false)
         else expand()
            reduce()
```

$|structure|$ est la longueur de *structure* sans compter les parenthèses.

Pour prendre en compte le fait qu'il peut y avoir plusieures productions avec les mêmes parties gauches, il y a deux stratégies possibles:

1. depth-first avec backtracking
2. breadth-first

Dans ce qui suit nous supposons qu'on veut ontenir toutes les analyses possibles. Mais on peut aussi modifier les algorithmes tel qu'ils s'arrêtent à la première analyse trouvée.

Idée d'un parser top-down depth-first avec backtracking : Quand on a pour le symbole non-terminal actuel plusieures productions possibles, on choisit d'abord la première, suit cette dérivation jusqu'au bout (échec ou analyse trouvée), retourne ensuite et reprend le parsing avec la deuxième production etc.

Au moment de choisir entre plusieures alternatives, il faut mettre les alternatives pas encore choisies (qui sont à poursuivre plus tard) dans une pile. En plus, il faut y mettre la structure obtenue jusque là, la position actuelle dans cette structure et la position actuel dans la phrase. Une entrée de la pile a alors la forme $\langle \mathit{structure}, \mathit{position}, \mathit{word}, \mathit{expansions} \rangle$ ou *expansions* est une séquence d'à moins une partie droite de production.

Nous supposons les opérations traditionnelles `push` et `pop` pour des piles. Une nouvelle procédure `backtrack` sortit de la pile la prochaine partie droite à prendre. Il y a une variable globale *stack* pour la pile.

Pour les entrées de la pile, nous supposons des opérations `structure`, `position`, `word` et `expansions` pour extraire les éléments d'une entrée.

Procédure backtrack

```
Input :  -
Output :  true
Effet supplémentaire :  modification des variables globales word, structure,
position et stack
Méthode :
   word := word(top(stack))
   position := position(top(stack))
   structure := structure(top(stack))
   remplacer structure_position par [structure_position[first(expansions(top(stack)))]]
   position := position + 1
   expansions(top(stack)) := rest(expansions(top(stack)))
   if expansions(top(stack)) = []
   then pop(stack)
   return(true)
```

Il faut aussi modifier `expand` tel que cette procédure ajoute une nouvelle entrée sur la pile dans le cas de plusieures productions possibles.

Procédure expand

```
Input :  -
Output :  true
Effet supplémentaire :  modification des variables structure, position et stack
Variable locale :  exp, une séquence d'expansions
Méthode :
   exp := ⟨α | X → α ∈ P ∧ X = structure_position⟩
   if rest(exp) ≠ []
   then stack := push(⟨structure, position, word, rest(exp)⟩, stack)
   remplacer structure_position par [structure_position[first(exp)]]
   position := position + 1
   return(true)
```

Le nouvel algorithme principal :

Algorithme parse-td

`top-down / gauche-droit / depth-first`

```
Données globales :  un dictionnaire et une CFG G = ⟨V, T, S, P⟩
Input :  une phrase w = w_1 ... w_n ,  n ≥ 0.
Output :  une séquence de structures pour w/false
Variables locales :
 word        position du mot actuel.  Valeur initiale :  1.
 structure   suite des symboles de N ∪ T ∪ {[,]}.  Valeur initiale :  [S].
 position    un pointeur vers un symbole dans structure.  Valeur initiale :  1.
 stack       une pile avec entrées de la forme ⟨structure, position, word, expansions⟩.
             Valeur initiale :  [].
 analyses    une séquence de structures.  Valeur initiale :  [].
```

```
Méthode :
   repeat:
      if position = |structure| + 1 ∧ word = n + 1
      then
         ajoute structure à analyses
         if stack = []
         then return(analyses)
         else backtrack()
      else
         if position = |structure| + 1 ∨ structure_position ∈ T
         then
            if stack = []
            then if analyses = []
               then return(false)
               else return(analyses)
            else backtrack()
         else
            expand()
            reduce()
```

Idée d'un parsing top-down breadth-first : au moment d'avoir plusieures alternatives, on lance (indépendant l'un de l'autre) un procès pour chacune de ces alternatives. Autrement dit, on fait un boucle dans lequel on poursuit le parsing pour chaque production possible.

Algorithme parse-td

top-down / gauche-droit / breadth-first

Données globales : un dictionnaire, une CFG $G = \langle V, T, S, P\rangle$ et une variable *analyses* avec valeur initiale $[\,]$

Input : une phrase $w = w_1 \ldots w_n$, $n \geq 0$, une structure s (valeur initiale $[S]$) et une position p dans s avec valeur initiale 1.

Output : -

Effet supplémentaire : mettre toutes les structures trouvées dans *analyses*

Variables locales :

expansions	une séquence de parties droites de productions
s-temp	une structure

```
Méthode :
   if p = |s| + 1 ∧ w = []
   then ajouter s à analyses
   else
      if p ≠ |s| + 1 ∧ s_p ∉ T
      then
         expansions := ⟨α | X → α ∈ P ∧ X = s_p⟩
         for all x ∈ expansions:
            s-temp := s
            remplacer s-temp_p par [s-temp_p[x]]
            parse(reduce(w, s-temp, p+1))
```

A la fin, les analyses possibles sont mises dans la variable *analyses*.

Procédure reduce

Input : une phrase w, une structure s et une position p

Output : une nouvelle phrase, nouvelle structure et nouvelle position

```
Méthode :
   if s_p ∈ T ∧ first(w) ∈ s_p
```

```
    then
        remplacer s_p par [s_p first(w)]
        reduce(rest(w), s, p+2)
    else return(w,s,p)
```

Contrainte pour les CFG pour lesquels les algorithmes top-down introduits ici marchent : pas de récursions à gauche.

Exemple 4 *G soit une grammaire avec symbole initiale S et les productions suivantes :*

S → *S coord S*
S → *NP VP*
NP → *det n*
VP → *v*

Avec un algorithme top-down depth-first avec backtracking on aura un boucle infini:

$w_{word} \ldots w_n$	*structure*
un homme rit	*S*
un homme rit	[*S* [*S coord S*]]
un homme rit	[*S* [[*S* [*S coord S*]]*coord S*]]
	...

Même si la récursion n'est pas immédiate, on va avoir le même problème, p.e. NP → det A n, A → AP A, AP → A.

Solutions:

1. Si la grammaire ne contient pas de productions de la forme $X \rightarrow \epsilon$, $X \in V$, on peut s'arrêter au moment ou le reste de la structure (à partir de la position actuelle) est plus longue que le reste de la phrase.
2. Chaque CFG G_1 avec récursions à gauche peut être transformé en une CFG sans récursions à gauche G_2 tel que G_1 est G_2 sont faiblement équivalentes. C'est-à-dire les deux grammaires génèrent les mêmes langues au niveau des phrases. Mais en général, les ensembles d'arbes de dérivation ne sont pas les mêmes. Or, la structure syntaxique d'une phrase est linguistiquement motivée, par conséquent G_2 n'est pas toujours linguistiquement adéquate.

Exemple 5 *G soit la grammaire de l'exemple précédant.*

La transformation en CFG sans récursion à gauche donne les productions suivantes :

S → *NP VP S'*
S → *NP VP*
S' → *coord S*
S' → *coord S S'*
NP → *det n*
VP → *v*