3 Parsing bottom-up

Idée : on commence par la phrase. Dans chaque pas on fait une réduction, c'est-à-dire on remplace la partie droite d'une production par sa partie gauche. Le but est d'arriver au symbole initial.

Exemple 6 *G soit un CFG avec les productions suivantes :*

1 $S \to NP\ VP$
2 $NP \to det\ n$
3 $NP \to n$
4 $VP \to v\ NP$

reste de la phrase	*prod.*	
Jean mange une pomme	–	–
mange une pomme	–	*n*
mange une pomme	*3*	*NP*
une pomme	–	*NP v*
pomme	–	*NP v det*
–	–	*NP v det n*
–	*2*	*NP v NP*
–	*4*	*NP VP*
–	*1*	*S*

Propriétés qui posent des problèmes pour ce type de parsing :

1. des productions de la forme $X \to \epsilon$ (on peut les appliquer autant qu'on veut)

2. des productions permettant des cycles $A \stackrel{n}{\Rightarrow} A$ ($n \geq 1$), c'est-à-dire $A \to A_1$, $A_1 \to A_2$, $\ldots, A_n \to A$

Ces deux peuvent être éliminé, c'est-à-dire

- Si $\epsilon \notin L(G)$ pour un CFG G, il existe un CFG G' faiblement équivalent à G qui ne contient pas de productions de la forme $X \to \epsilon$.
- Pour chaque CFG G il existe un CFG G' faiblement équivalent à G qui ne permet pas de cycles $A \stackrel{n}{\Rightarrow} A$ ($n \geq 1$)

Mais comme l'équivalence est seulement faible, 1. et 2. sont des vraies contraintes.

Les parser bottom-up les plus importants sont les parser shift-reduce. Ils fonctionnent avec

1. une pile $stack1$ contenant le reste de la phrase,

2. une pile $stack2$ contenant ce qu'on a déjà réduit,

3. une opération `shift` qui consiste à enlever le premier élément de $stack1$ et l'ajouter à $stack2$

4. et une opération `reduce` qui consiste à remplacer sur $stack2$ la partie droite d'une production par la partie gauche de cette production.

Procédure shift

```
Input :  une pile stack1 avec stack1 ≠ [] et une pile stack2
Output :  deux piles
Méthode :
   stack2 := push(top(stack1), stack2)
   stack1 := pop(stack1)
   return(stack1, stack2)
```

Procédure reduce

```
Données globales : CFG G = ⟨V, T, S, P⟩ et dictionnaire
Input : une pile stack2
Output : une nouvelle pile
Variables locales : séquence right-side
Méthode :
   if il existe un t ∈ T tel que top(stack2) ∈ t
   then return(push(t, pop(stack2)))
   else
      right-side := [top(stack2)]
      while stack2 ≠ []
         if il existe une production A → right-side ∈ P
         then return(push(A, pop(stack2)))
         else
            stack2 := pop(stack2)
            if stack2 ≠ []
            then right-side := [top(stack2)] + right-side
      while right-side ≠ []
         stack2 := push(first(right-side), stack2)
         right-side := rest(right-side)
      return(stack2)
```

Supposons pour l'instant que `reduce` est déterministe, c'est-à-dire qu'on trouve jamais plus qu'une production pour une réduction possible. Pour cela il faut que le CFG soit tel que

1. chaque mot a exactement une catégorie préterminale
2. et aucune partie droite d'une production est préfixe ou suffixe d'une autre partie droite de production

Algorithme recognize-bu

`bottom-up/gauche-droite/déterministe`

```
Données globales : CFG G = ⟨V, T, S, P⟩ et dictionnaire
Input : une pile stack1, valeur initiale [w_1, ..., w_n], n ≥ 1, et une pile stack2,
valeur initiale []
Output : true/false
Variable locale : pile s
Méthode :
   s := reduce(stack2)
   if s ≠ stack2
   then recognize(stack1, s)
   else if stack1 ≠ []
      then recognize(shift(stack1, stack2))
      else if stack2 = [S]
         then return(true)
         else return(false)
```

Exemple 7 *G soit un CFG avec les productions suivantes :*

1 $S \to NP\ VP$
2 $NP \to det\ cn$
3 $NP \to pn$
4 $VP \to v\ NP$

stack1	*stack2*	*action*
Jean mange une pomme	–	–
mange une pomme	*Jean*	*shift*
mange une pomme	*pn*	*reduce*
mange une pomme	*NP*	*reduce*
une pomme	*NP mange*	*shift*
une pomme	*NP v*	*reduce*
...	...	...
–	*S*	*reduce*

Dans ce qui suit on va introduire une variante non-dététerministe de l'algorithme `recognize-bu`, c'est-à-dire la grammaire peut

(a) affecter plusieurs préterminaux à un mot et

(b) contenir des production $A \to \alpha$ et $A \to \alpha'\alpha$ ou des production $A \to \alpha$ et $A \to \alpha\alpha'$

Conditions pour la CFG (comme dans le cas déterministe) :

1. pas de cycles
2. pas de production $A \to \epsilon$

Comme dans le cas top-down on a les alternatives d'une stratégie depth-first avec backtracking ou d'une stratégie breadth-first.

Dans ce qui suit on va réaliser la première stratégie.

Supposons que les productions et les entrées du dictionnaire sont ordonnées et qu'elles vont toujours être testées dans leur ordre dans la grammaire. P_i donne la production avec index i, D_i l'entrée lexicale avec index i.

A part de $stack1$ et $stack2$ il faut une troisième pile $stack3$ sur laquelle on met les informations dont on a besoin pour faire du backtracking. Chaque entrée de $stack3$ contient soit un 0 (pour une opération `shift`), soit l'index d'une production/d'une entrée lexicale.

Procédure shift

```
Données globales : une pile stack1 avec stack1 ≠ [], une pile stack2 et une pile stack3
Input : --
Output : true
Effet supplémentaire : modification de stack1, stack2 et stack3
Méthode :
   stack2 := push(top(stack1), stack2)
   stack1 := pop(stack1)
   stack3 := push(0, stack3)
   return(true)
```

Une entrée lexicale est représentée comme suit : $\langle Jean, n \rangle$.

Procédure reduce

```
Données globales : CFG G = ⟨V,T,S,P⟩ et dictionnaire D, stack2, stack3
Input : index i
Output : nouvel index
Effet supplémentaire : modification de stack2 et stack3
Variables locales : séquence right-side, index m
Méthode :
   m := i
   while m ≤ |P| + |D| :
      if il existe un t ∈ T tel que ⟨top(stack2),t⟩ = D_m
```

```
        then
            stack2 := push(t, pop(stack2))
            stack3 := push(m, stack3)
            return(m)
        else if P_m ∈ P
            then
                right-side := []
                while stack2 ≠ [] ∧ |right-side| < |right(P_m)|
                    right-side := [top(stack2)] + right-side
                    stack2 := pop(stack2)
                if right(P_m) = right-side
                then
                    stack2 := push(left(P_m), stack2)
                    stack3 := push(m, stack3)
                    return(m)
                else while right-side ≠ []
                    push(first(right-side), stack2)
                    right-side := rest(right-side)
        m := m + 1
    return(m)
```

Au cours d'un bracktracking, on va chercher l'index de la dernière opération (production, entrée lexicale ou `shift`) sur $stack3$, annuler cette opération et continuer.

Exemple 8

Productions :

1	$S \rightarrow NP\ VP$	*3*	$NP \rightarrow det\ n$	*5*	$VP \rightarrow v$
2	$NP \rightarrow n$	*4*	$VP \rightarrow v\ NP$		

Dictionnaire :

6 $\langle un,det\rangle$ *7* $\langle homme,n\rangle$ *8* $\langle rit,v\rangle$

stack1	*stack2*	*stack3*	*action*
un homme rit	–	–	–
homme rit	*un*	*0*	*shift*
homme rit	*det*	*6 0*	*reduce*
rit	*homme det*	*0 6 0*	*shift*
rit	*n det*	*7 0 6 0*	*reduce*
rit	*NP det*	*2 7 0 6 0*	*reduce*
–	*rit NP det*	*0 2 7 0 6 0*	*shift*
–	*v NP det*	*8 0 2 7 0 6 0*	*reduce*
–	*VP NP det*	*5 8 0 2 7 0 6 0*	*reduce*
–	*S det*	*1 5 8 0 2 7 0 6 0*	*reduce*
–	*VP NP det*	*5 8 0 2 7 0 6 0*	*backtrack*
...	...	...	...
rit	*n det*	*7 0 6 0*	*backtrack*
rit	*NP*	*3 7 0 6 0*	*reduce*
–	*rit NP*	*0 3 7 0 6 0*	*shift*
–	*v NP*	*8 0 3 7 0 6 0*	*reduce*
...	...	...	...
–	*S*	*1 5 8 0 3 7 0 6 0*	*reduce*

Procédure backtrack

```
Données globales :  CFG G = ⟨V,T,S,P⟩ et dictionnaire D
    trois piles stack1, stack2, stack3, stack3 ≠ []
```

```
Output :  index m
Variables locales :  séquence right-side, nombre m
Effet supplémentaire :  modification de stack1, stack2 et stack3
Méthode :
   m := top(stack3)
   if m = 0
   then
      stack1 := push(top(stack2, stack1)
      stack2 := pop(stack2)
   else if D_m ∈ D ∧ D_m = ⟨w, t⟩
      then stack2 := push(w, pop(stack2))
      else
         right-side := right(P_m)
         stack2 := pop(stack2)
         while right-side ≠ []
            stack2 := push(first(right-side), stack2)
            right-side := rest(right-side)
   stack3 := pop(stack3)
   return(m)
```

Algorithme recognize-bu

`bottom-up/gauche-droite`
`depth-first/backtracking`

```
Données globales :  CFG G = ⟨V, T, S, P⟩ et dictionnaire
Input :
 une pile stack1  valeur initiale [w_2, ..., w_n], n ≥ 1,
 une pile stack2  valeur initiale [w_1]
 une pile stack3  valeur initiale []
 index n          valeur initiale 0
Output :  true/false
Variable locale :  index m
Méthode :
   m := reduce(n)
   if m = |P| + |D| + 1 ∧ stack1 ≠ [] ∧ n ≠ 1
   then
      shift()
      recognize(stack1, stack2, stack3, 0)
   else if m ≤ |P| + |D|
      then recognize(stack1, stack2, stack3, 0)
      else if stack2 = [S] ∧ stack1 = []
         then return(true)
         else if stack3 ≠ []
            then m := backtrack()
               recognize(stack1, stack2, stack3, m + 1)
           else return(false)
```

Pour obtenir une analyse (transformer le recognizer en parser), on a deux possibilités :

1. en fonction des indexes de productions/entrées lexicales sur *stack*3, on peut construire un arbre de dérivation. *stack*3 représente une dérivation de droite de w.
2. on peut construire une analyse parallèllement au parsing : les éléments de *stack*2 seraient des arbres représentant une séquence des structures pour des parties de w. `reduce` construisait un nouvel arbre à partir de plusieurs arbres dans *stack*2.

Exemple pour la deuxième méthode :

Exemple 9

Productions :

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

$stack1$	$stack2$	$action$
un homme rit	–	–
homme rit	*un*	*shift*
homme rit	[*det un*]	*reduce*
rit	*homme* [*det un*]	*shift*
rit	[*n homme*] [*det un*]	*reduce*
rit	[*NP* [*n homme*] [*det un*]]	*reduce*
–	*rit* [*NP* [*n homme*] [*det un*]]	*shift*
–	[*v rit*] [*NP* [*n homme*] [*det un*]]	*reduce*
–	[*VP* [*v rit*]] [*NP* [*n homme*] [*det un*]]	*reduce*
–	[*S* [*VP* [*v rit*]] [*NP* [*n homme*] [*det un*]]]	*reduce*