4 Parsing coin-gauche (*left-corner parsing*)

Stratégie mixte : en partie bottom-up, en partie top-down.

Idée : on construit le coin gauche d'un arbre correspondant à une production bottom up, ensuite on construit le reste de cet arbre top-down.

Condition pour l'application d'une production $A \to A_1 \dots A_n$: on a déjà trouvé un sous-arbre pour A_1 (le coin gauche). C'est-à-dire en appliquant une production, on fait une hypothèse à l'égard du noeud père et des noeuds frères.

On traverse l'arbre de dérivation tel que pour chaque noeud n avec noeuds fils $n_1, \dots, n_m$:

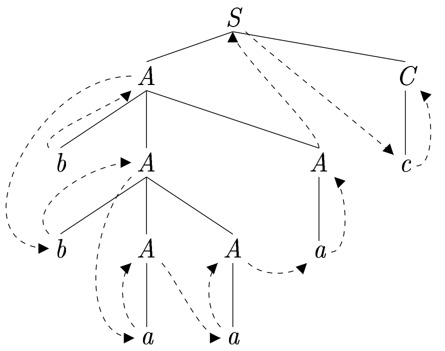
- tous les noeuds dominés par n_1 (ceci inclut n_1) sont visités avant n
- n est visités avant tous les noeuds dominés par un des n_i, $1 < i \leq m$
- pour $1 \leq i < m$: tous les noeuds dominés par n_i sont visités avant tous les noeuds dominés par n_{i+1}.

Exemple 10

Productions :

1	$S \to A\ C$	*3*	$A \to a$
2	$A \to b\ A\ A$	*4*	$C \to c$

Pour bbaaac cela donne



Ordre dans lequel les productions sont appliquées :

Top-down gauche-droite	*1 2 2 3 3 3 4*
Bottom-up gauche-droite	*3 3 2 3 2 4 1*
Left-corner	*2 2 3 3 3 1 4*

Pour un recognizer on a besoin de trois piles :

- $stack1$ pour ce qu'on a déjà reconnu (et qui va être utilisé comme coin gauche ou un des noeuds frères $n_2, dots, n_m$ d'une production)
- $stack2$ pour les catégories qu'il faut encore analyser top-down ($n_2, \dots, n_m$)
- $stack3$ pour les catégories à complêter (n)

Trois procédures :

Procédure `reduce` (on a trouvé un coin gauche) : On peut appliquer une règle $A_0 \to A_1 \dots A_n$ si le top de $stack1$ est A_1. On enlève ce top, on met $A_2 \dots A_n$ avec un symbole $t \notin V \cup T$ sur $stack2$ et on met A_0 sur $stack3$. t marque la fin d'une partie droite sur $stack2$.

Procédure `move` (on est arrivé à la fin d'une partie droite de production) : Si t est le top de $stack2$, on a complètement parcouru un sous-arbre. On peut enlever le top de $stack3$ (la racine de cet arbre) et le mettre sur $stack1$. En plus, t est enlevé de $stack2$.

Procédure `remove` (on a trouvé une des catégories d'une partie droite de production qui ne sont pas coin gauche) : Si *stack*1 et *stack*2 commence par les mêmes symboles, on peut les supprimer.

Exemple 11

Productions :

$S \to NP\ VP$ $NP \to det\ n$ $VP \to v$

*stack*1	*stack*2	*stack*3	*action*
un homme rit	*S*	–	
homme rit	*t S*	*det*	*reduce*
det homme rit	*S*		*move*
homme rit	*n t S*	*NP*	*reduce*
rit	*t n t S*	*n NP*	*reduce*
n rit	*n t S*	*NP*	*move*
rit	*t S*	*NP*	*remove*
NP rit	*S*	–	*move*
rit	*VP t S*	*S*	*reduce*
–	*t VP t S*	*v S*	*reduce*
v	*VP t S*	*S*	*move*
–	*t VP t S*	*VP S*	*reduce*
VP	*VP t S*	*S*	*move*
–	*t S*	*S*	*remove*
S	*S*	–	*move*
–	–	–	*remove*

Procédure reduce

```
Données globales :
 CFG G = ⟨V,T,S,P⟩
 un dictionnaire D
 une pile stack1
 une pile stack2
 une pile stack3
Input :  --
Output :  true/false
Effet supplémentaire :  modification de stack1, stack2 et stack3
Variable locale :  séquence right-side
Méthode :
   if stack1 ≠ [] ∧ stack2 ≠ [] ∧ top(stack2) ≠ t
   then
      if il existe un x ∈ T tel que ⟨top(stack1), x⟩ ∈ D
      then
         stack2 := push(t, stack2)
         stack3 := push(x, stack3)
         stack1 := pop(stack1)
         return(true)
      else if il existe un p ∈ P avec first(right(p)) = top(stack1)
         then
            stack2 := push(t, stack2)
            right-side := rest(right(p))
            while right-side ≠ []
               stack2 := push(last(right-side), stack2)
               right-side := without-last(right-side)
            stack3 := push(left(p), stack3)
            stack1 := pop(stack1)
```

```
            return(true)
         else return(false)
   else return(false)
```

(On a trouvé un coin gauche et on applique une production/une entrée lexicale.)

`last`(x) retourne le dernier élément de x, `without-last`(x) retourne x privé de son dernier élément.

Procédure remove

```
Données globales :
 une pile stack1,
 une pile stack2
Input :  --
Output :  true/false
Effet supplémentaire :  modification de stack1 et stack2
Méthode :
   if stack1 ≠ [] ∧ stack2 ≠ [] ∧ top(stack1) = top(stack2)
   then
      stack1 := pop(stack1)
      stack2 := pop(stack2)
      return(true)
   else return(false)
```

(On a trouvé un sous-arbre qui correspond à la catégorie qu'on était en train de chercher.)

Procédure move

```
Données globales :
 une pile stack1,
 une pile stack2,
 une pile stack3
Input :  --
Output :  true/false
Effet supplémentaire :  modification de stack1, stack2 et stack3
Méthode :
   if stack3 ≠ [] ∧ top(stack2) = t
   then
      stack2 := pop(stack2)
      stack1 := push(top(stack3), stack1)
      stack3 := pop(stack3)
      return(true)
   else return(false)
```

(On est arrivé à la fin d'une production. Autrement dit, on vient de complêter un sous-arbre.)

Procédure recognize-lc

`coin-gauche / non-déterministe`

```
Données globales :  CFG G = ⟨V,T,S,P⟩, dictionnaire D
Input :
 une pile stack1  valeur initiale [w_1 ... w_n]
 une pile stack2  valeur initiale [S]
 une pile stack3  valeur initiale []
Output :  true/false
Méthode :
```

```
if stack1 = stack2 = stack3 = []
then return (true)
else if move()
   then recognize-lc(stack1, stack2, stack3)
   else if reduce()
      then recognize-lc(stack1, stack2, stack3)
      else if remove()
         then recognize-lc(stack1, stack2, stack3)
         else return(false)
```

Cet algorithme est non-déterministe parce qu'il peut y avoir plusieures productions avec les mêmes coins gauches (⇒ deux applications différentes de `reduce`).

Exemple 12

Les mêmes coins gauches :

NP → det n, NP → det AP n

Le préterminal det est coin gauche des deux productions. Dans une situation

$stack1$	$stack2$	$stack3$
det homme rit	*S*	–

on a le choix entre les deux productions. La première donnerait [n t S], la deuxième [n AP t S] comme nouvelle valeur de $stack2$.

Un autre problème est qu'un sous-arbre reconnu peut être soit un des sous-arbres non-gauches d'une production, soit le coin gauche d'une autre production (⇒ une application de `remove` ou de `reduce`). Dans ce cas-là la version en-dessus ne fait que le `reduce`.

Exemple 13

Le coin gauche d'une production est un des symboles non-gauche de la partie droite d'une autre production :

NP → det n, NP → n

Dans une situation (causé par une application antérieure de la première production)

$stack1$	$stack2$	$stack3$
n rit	*n t S*	*NP*

on a le choix entre un `remove` *ou un* `reduce` *avec la deuxième production. Dans le premier cas, on aurait*

$stack1$	$stack2$	$stack3$
rit	*t S*	*NP*

et dans le deuxième

$stack1$	$stack2$	$stack3$
rit	*t n t S*	*NP NP*

comme nouvelles valeurs de $stack1$, $stack2$ *et* $stack3$.

La version d'en-dessus va seulement poursuivre la deuxième possibilité.

Deux stratégies pour gérer de telles ambiguités : depth-first avec backtracking ou breadth-first.

Version breadth-first : nous utilisons une variable locale $configurations$, un ensemble dans lequel nous mettons les nouveaux triples $\langle stack1, stack2, stack3\rangle$ obtenus par les appels de `move`, `remove` et `reduce`. Ensuite, on continue le parsing parallèlement pour chacun de ces triples.

Supposons que les entrées lexicales et les productions sont numérotées et `reduce` est modifié tel qu'il prend un argument i, commence la recherche d'une entrée lexicale/production adéquate à partir de l'index i et retourne soit (a) l'index de l'entrée lexicale/production appliquée si une réduction a été possible soit (b) 0 si une réduction n'a pas été possible.

Procédure recognize-lc

`coin-gauche / breadth-first`

```
Données globales :  CFG G = ⟨V,T,S,P⟩, dictionnaire D
Input :
 une pile stack1  valeur initiale [w_1 ... w_n]
 une pile stack2  valeur initiale [S]
 une pile stack3  valeur initiale []
Output :  true/false
Variables locales :
 ensemble de triples configurations  valeur initiale ∅
 piles s1, s2, s3
 index n
Méthode :
   if stack1 = stack2 = stack3 = []
   then return (true)
   else
      s1 := stack1
      s2 := stack2
      s3 := stack3
      if move()
      then
         configurations := configurations ∪ {⟨stack1, stack2, stack3⟩}
         stack1 := s1
         stack2 := s2
         stack3 := s3
      n := reduce(1)
      while n > 0
         configurations := configurations ∪ {⟨stack1, stack2, stack3⟩}
         stack1 := s1
         stack2 := s2
         stack3 := s3
         n := reduce(n + 1)
      if remove()
      then
         configurations := configurations ∪ {⟨stack1, stack2, stack3⟩}
         stack1 := s1
         stack2 := s2
         stack3 := s3
      if configurations = ∅
      then return(false)
      else for all ⟨s1', s2', s3'⟩ from configurations:
         if recognize-lc(s1', s2', s3')
         then return(true)
      return(false)
```

Comme dans le cas d'un parsing bottom-up, on a toujours des problèmes avec

- des productions de la forme $A \rightarrow \epsilon$
- et des cycles $A \stackrel{n}{\Rightarrow} A$, $n \geq 1$

parce que pour une production $A \rightarrow \alpha$ avec $|\alpha| \leq 1$, parsing coin-gauche est comme parsing bottom-up.

Des récursions à gauche, qui sont problématiques pour le parsing top-down, ne posent pas de problème pour un parsing coin-gauche parce que les coins gauches sont parsé bottom-up :

Dans $A \rightarrow A\alpha$ on peut faire un `reduce` à partir de A comme coin gauche, mais ensuite il faut trouver α avant d'utiliser le noeud père de cette production à nouveau comme coin gauche avec la même production.

Pour obtenir une analyse, on peut utiliser une quatrième pile $stack4$ dans laquelle on construit l'analyse au cours du parsing.

Exemple 14

Productions :

$S \rightarrow NP\ VP$ $\qquad$ $NP \rightarrow det\ n$ $\qquad$ $VP \rightarrow v$

$stack1$	$stack2$	$stack3$	$stack4$	$action$
un homme rit	S	–	–	
homme rit	$t\ S$	det	[*det un*]	*reduce*
det homme rit	S		[*det un*]	*move*
homme rit	$n\ t\ S$	NP	[*NP* [*det un*]]	*reduce*
rit	$t\ n\ t\ S$	$n\ NP$	[*n homme*] [*NP* [*det un*]]	*reduce*
n rit	$n\ t\ S$	NP	[*n homme*] [*NP* [*det un*]]	*move*
rit	$t\ S$	NP	[*NP* [*det un*] [*n homme*]]	*remove*
NP rit	S	–	[*NP* [*det un*] [*n homme*]]	*move*
rit	$VP\ t\ S$	S	[*S* [*NP* [*det un*] [*n homme*]]]	*reduce*
–	$t\ VP\ t\ S$	$v\ S$	[*v rit*] [*S* [*NP* [*det un*] [*n homme*]]]	*reduce*
v	$VP\ t\ S$	S	[*v rit*] [*S* [*NP* [*det un*] [*n homme*]]]	*move*
–	$t\ VP\ t\ S$	$VP\ S$	[*VP* [*v rit*]] [*S* [*NP* [*det un*] [*n homme*]]]	*reduce*
VP	$VP\ t\ S$	S	[*VP* [*v rit*]] [*S* [*NP* [*det un*] [*n homme*]]]	*move*
–	$t\ S$	S	[*S* [*NP* [*det un*] [*n homme*]] [*VP* [*v rit*]]]	*remove*
S	S	–	[*S* [*NP* [*det un*] [*n homme*]] [*VP* [*v rit*]]]	*move*
–	–	–	[*S* [*NP* [*det un*] [*n homme*]] [*VP* [*v rit*]]]	*remove*

Le contenu de $stack4$ est modifié dans `reduce` et `remove`, dans `move` il reste inchangé.

Dans `reduce` il faut

- mettre un nouvel élément sur $stack4$ si on a trouvé une entrée lexicale
- ajouter un noeud père au premier élément de $stack4$ si on a trouvé le coin gauche d'une production

(Dans le nouveau `reduce` seul les lignes concernant $stack4$ sont ajoutées, le reste est inchangé par rapport à la version avec indexes.)

Procédure reduce

Données globales :
 CFG $G = \langle V, T, S, P\rangle$
 un dictionnaire D
 une pile $stack1$
 une pile $stack2$
 une pile $stack3$
 une pile $stack4$
Input : index i
Output : un index
Effet supplémentaire : modification de $stack1$, $stack2$, $stack3$ et $stack4$
Variable locale : séquence $right\text{-}side$
Méthode :

```
if stack1 ≠ [] ∧ stack2 ≠ [] ∧ top(stack2) ≠ t ∧ i ≤ |P| + |D|
then
   if il existe un x ∈ T tel que ⟨top(stack1), x⟩ = D_i
   then
      stack2 := push(t, stack2)
      stack3 := push(top(stack1), stack3)
      stack4 := push([x top(stack1)], stack4)
      stack1 := pop(stack1)
      return(i)
   else if il existe un p = P_i avec first(right(p)) = top(stack1)
      then
         stack2 := push(t, stack2)
         right-side := rest(right(p))
         while right-side ≠ []
            stack2 := push(last(right-side), stack2)
            right-side := without-last(right-side)
         stack3 := push(top(stack1), stack3)
         stack4 := push([top(stack1)] + [top(stack4)], pop(stack4))
         stack1 := pop(stack1)
         return(i)
      else reduce(i + 1)
else return(0)
```

Dans `remove` il faut mettre le premier élément de *stack*4 comme noeud fils à droite dans le deuxième élément de *stack*4 :

Procédure remove

```
Données globales :
 une pile stack1, |stack1| ≥ 1,
 une pile stack2, |stack2| ≥ 1,
 une pile stack4
Input :  --
Output :  true/false
Variable locale :  arbre
Effet supplémentaire :  modification de stack1 et stack2
Méthode :
   if stack1 ≠ [] ∧ stack2 ≠ [] ∧ top(stack1) = top(stack2)
   then
      stack1 := pop(stack1)
      stack2 := pop(stack2)
      if |stack4| > 1
      then
         arbre := top(stack4)
         stack4 := pop(stack4)
         stack4 := push(top(stack4) + arbre, pop(stack4))
      return(true)
   else return(false)
```

Pour transformer `recognize` en `parse` on ajoute la quatrième pile comme argument et on retourne *stack*4 au moment d'avoir trouvé une analyse. Dans *configurations* on a des quadruples $\langle stack1, \ldots, stack4 \rangle$.

Look-ahead On peut réduire le nombre de règles possibles à un point en utilisant un *look-ahead* : On détermine pour chaque symbole non-terminal X tous les symboles $A \in V \cup T$ qu'on peut dériver comme symbole à gauche à partir de X. (C'est-àdire $X \stackrel{*}{\Rightarrow} A\gamma$ pour quelque $\gamma \in (V \cup T)^*$.)

Si dans un `reduce` on peut appliquer une production $A \to A'\alpha$ parce qu'on a trouvé un coin gauche

A', et si on a un symbole X sur $stack2$, on sait que : si on trouve une dérivation (une analyse) en appliquant cette production, dans cette analyse on va dériver $A\gamma$ à partir de X.

Autrement dit, si A n'est pas dérivable comme symbole à gauche à partir de X, ce n'est pas la peine d'essayer la production $A \rightarrow A'\alpha$ parce qu'elle va mêner à un échec.

Définition d'une relation *link* :

Définition 4 *$G = \langle V, T, S, P\rangle$ soit une grammaire hors contexte. $link \subseteq V \times (V \cup T)$ est une relation tel que $\langle X, Y\rangle \in link$ ssi*

(a) $X = Y$
(b) ou il existe un $\gamma \in (V \cup T)^*$ tel que $X \rightarrow Y\gamma \in P$
(c) ou il existe un $X' \in V$ tel que $\langle X, X'\rangle, \langle X', Y\rangle \in link$

(La relation *link* est reflexive et transitive.)

Exemple 15

Productions :

$S \rightarrow NP\ VP$ $\qquad NP \rightarrow det\ n$ $\qquad VP \rightarrow v$

$link = \{\langle S, S\rangle, \langle S, NP\rangle, \langle S, det\rangle, \langle NP, NP\rangle, \langle NP, det\rangle, \langle VP, VP\rangle, \langle VP, v\rangle\}$

$stack1$	$stack2$	$stack3$
rit	S	–

Seule production/entrée lexicale avec coin gauche rit : $v \rightarrow rit$.

Mais comme $\langle S, v\rangle \notin link$, cette production ne peut pas mêner à une analyse. $\Rightarrow$ on ne la choisit pas $\Rightarrow$ échec.

Pour intégrer l'utilisation d'un look-ahead dans notre algorithme, il suffit de modifier `reduce` : On applique une production/entrée lexicale avec partie gauche/catégorie préterminale A seulement si $\langle$`top(`$stack2$`)`$, A\rangle \in link$.

Avec la même technique, on peut rendre le parsing top-down plus efficace.