CSC4251_4252 : Analyse Syntaxique

Pascal Hennequin, J. Paul Gibson, Denis Conan

Novembre 2024

Sommaire

1 Analyse syntaxique : trois stratégies

- **Résolution générale**
  - Applicable aux langages algébriques non déterministes
  - Complexité $O(n^3)$
  - Algorithmes : Earley, CYK,Valiant, etc.
- **Analyse descendante ou « LL »**
  - Applicable à « beaucoup » de langages algébriques déterministes
  - Algorithme plus intuitif, mais contraignant sur l'écriture de la grammaire
  - Complexité linéaire $O(n)$
  - Outils : « à la main », `javacc`, ANTLR, etc.
- **Analyse ascendante ou « LR » ou *Shift/Reduce***
  - Applicable à « énormément » de langages algébriques déterministes
  - Applicabilité fonction du langage et pas de la grammaire
  - Complexité linéaire $O(n)$
  - Outils : yacc, bison, CUP, GOLD, etc.

2 Analyse descendante I

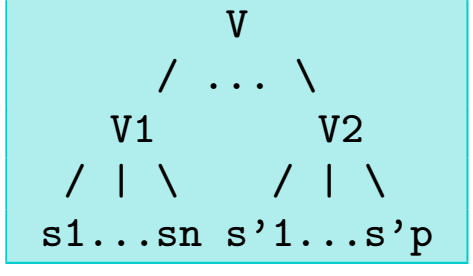
Principe

- Construction de l'arbre de syntaxe de haut en bas
- Utilisation de la grammaire en production
- Arbre de décision ET/OU
  - Nœud OU : pour remplacer un symbole non-terminal, on a le choix entre les différentes productions (ou alternatives) avec ce symbole en membre gauche
  - Nœud ET : pour une production, il faut reconnaître la séquence des symboles en membre droit

Exploration de l'arbre

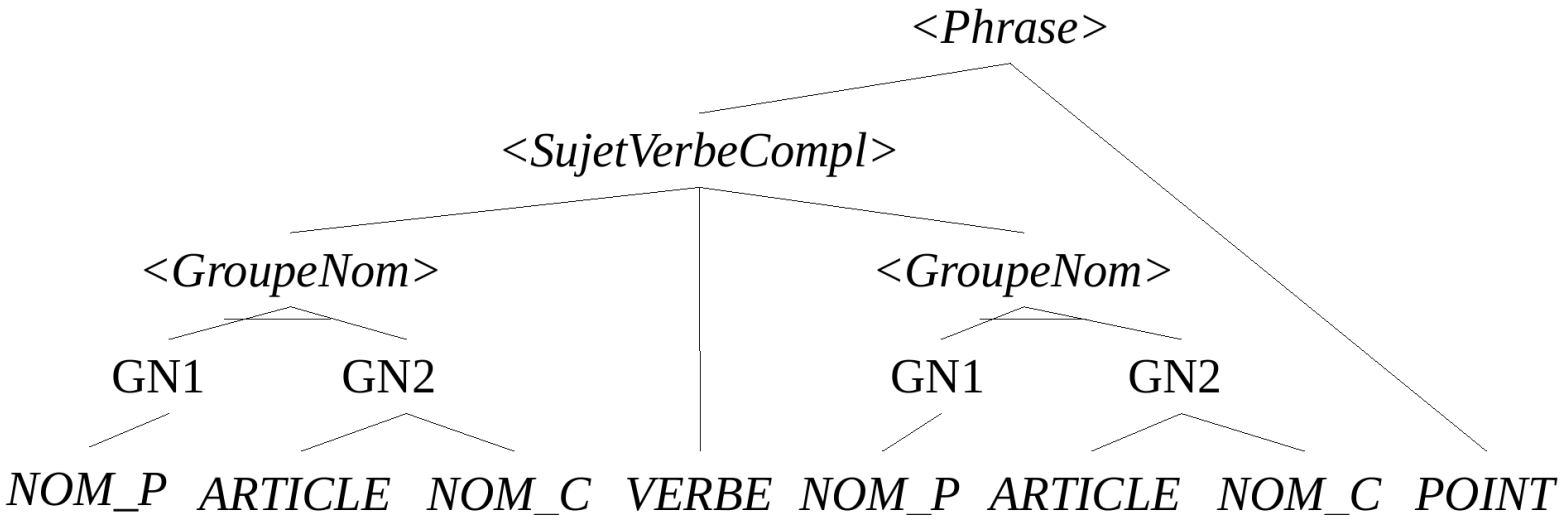
- En profondeur avec retour arrière
- En « parallèle »...
- Rendre prédictible les choix OU ?

(v1) v ::= $s_1 s_2 ... s_n$;
(v2) v ::= $s'_1 s'_2 ... s'_p$;



2 Analyse descendante II

- **Exemple de language naturel élémentaire**



- **Parcours en profondeur : « Obi-Wan mange un gnou. » ?**
  - Choix GN1, Obi-wan=NOM_P ? ok, mange=VERBE ? ok,
    - Choix GN1, un=NOM_P ? echec, retour dernier choix,
    - Choix GN2, un=ARTICLE ? ok, gnou=NOM_C ? ok, .=POINT ? ok
  - Fini, Accepté.

2 Analyse descendante III

- **Descente récursive : Mise en œuvre simple et intuitive**
  - Pour chaque symbole X, une fonction reco_X()
    - Si X est terminal, `reco_X()` lit le *token* X en entrée, ou échec
    - Si X est non-terminal, `reco_X()` choisit une production de X et appelle successivement `reco_`S_i`()` pour les symboles en membre droit
- **Contraintes sur l'écriture de la grammaire**
  - Pas de récursivité gauche dans les productions ! ! !
    - Directement « L :=L a | a ; »
    - Ou indirectement « L :=M ... ; M := L ... ; »
  - D'autres contraintes : factorisation gauche de la grammaire, etc.
- **Rendre prédictible (déterministe)**
  - Utilisation du *Lookahead*, calcul de fonctions *first()* et *next()*
  - Principe : une production n'est applicable que si récursivement elle peut générer le prochain token en entrée

3 Analyse ascendante « Shift/Reduce » I

Principe

- Construire l'arbre syntaxique de bas en haut
- Utilisation de la grammaire en reconnaissance
  - Réduction = remplacement membre droit par membre gauche
- Utilisation d'une pile
  - *Shift* = empiler le *token* suivant
  - Tester les réductions possibles en tête de pile
  - *Reduce* = dépiler n symboles d'un membre droit, empiler le symbole non-terminal du membre gauche

Indéterminisme

- « reduce/reduce » plusieurs réductions possibles au même moment
- « shift/reduce » réduction possible, mais décalage préférable
  - NB : S'il existe une production vide, il y a toujours une réduction possible même avec une pile vide !

3 Analyse ascendante « Shift/Reduce » II

- Entrée = « Obi-Wan mange un gnou. »

Opération	**Pile**
décalage	NOM_P
réduction	⟨*GroupNom*⟩
décalage	⟨*GroupNom*⟩ VERBE
décalage	⟨*GroupNom*⟩ VERBE ARTICLE
décalage	⟨*GroupNom*⟩ VERBE ARTICLE NOM_C
réduction	⟨*GroupNom*⟩ VERBE ⟨*GroupNom*⟩
réduction	⟨*SujetVerbeCompl*⟩
décalage	⟨*SujetVerbeCompl*⟩ POINT
réduction	⟨*Phrase*⟩
Succès	

NB : Succès = règle conventionnelle « $START = ⟨*Phrase*⟩ EOF »

3 Analyse ascendante « Shift/Reduce » III

- **Entrée = « alpha + bêta + 42 »**

```
Expr  :=  Expr '+' Expr
      |   INT
      |   SYMBOL ;
```

Opération	Pile
décalage	**SYMBOL**
réduction	*Expr*
décalage	*Expr* '+'
décalage	*Expr* '+' ***SYMBOL***
réduction	*Expr* '+' *Expr*

Ambiguïté
Conflit *shift/reduce*

réduction	*Expr*
décalage	*Expr* '+'
décalage	*Expr* '+' **INT**
réduction	*Expr* '+' *Expr*
réduction	*Expr*
succès	

(alpha+bêta)+42

décalage	*Expr* '+' *Expr* '+'
décalage	*Expr* '+' *Expr* '+' **INT**
réduction	*Expr* '+' *Expr* '+' *Expr*
réduction	*Expr* '+' *Expr*
réduction	*Expr*
succès	

alpha+(bêta+42)

3 Analyse ascendante « Shift/Reduce » IV

- **Entrée = « 42 222 »**

(R0)	⟨*Liste*⟩	:=	/* mot vide */
(R1)		|	⟨*Liste*⟩ INT ;

(R0)	⟨*Liste*⟩	:=	/* mot vide */
(R1)		|	INT ⟨*Liste*⟩ ;

Opération	Pile

Opération	Pile

- Quand fait-on la réduction R0 ?
- Dans quel ordre sont réduits les entiers de la liste ?

3.1 Solution

- **Entrée = « 42 222 »**

```
(R0) ⟨Liste⟩ := /* mot vide */
(R1)          |  ⟨Liste⟩ INT ;
```

Opération	Pile
réduction R0	⟨*Liste*⟩
décalage	⟨*Liste*⟩ INT
réduction R1 (42)	⟨*Liste*⟩
décalage	⟨*Liste*⟩ INT
réduction R1 (222)	⟨*Liste*⟩

```
(R0) ⟨Liste⟩ := /* mot vide */
(R1)          |  INT ⟨Liste⟩ ;
```

Opération	Pile
décalage	INT
décalage	INT
réduction R0	INT INT ⟨*Liste*⟩
réduction R1 (222)	INT ⟨*Liste*⟩
réduction R1 (42)	⟨*Liste*⟩

- **Récursivités droites ou gauches valides**
- **Récursivité gauche préférée par analyseur LR**

4 Automate LR I

- **Analyse LR : Automate à pile pour décider des *shift/reduce***
- **États de l'automate**
  - États viables du sommet de pile (séquence de n symboles)
  - Viable = le sommet de pile pourra être réduit à terme
  - Sommet de pile = préfixe de membre droit de production
- **Pile**
  - A chaque **Push**, l'état de l'automate est empilé avec le symbole
- **Transitions**
  - *Shift* : transition « simple » avec *Push* d'un *token*
  - *Reduce* : transition avec remplacement (*Pop** + *Push*) et retour à l'état stocké dans la pile (dernier *Pop*)
  - En général, les transitions *reduce* ne sont pas représentées

4 Automate LR II

- **Exemple LALR(1) avec « `cup-dump` »**

```
terminal TOK;
nonterminal list;
list ::= /* vide */
       | list TOK
;
```

```
== Terminals ==
[0]EOF
[1]error
[2]TOK
=== Non terminals =
[0]list
=== Productions =
[0] list ::=
[1] $START ::= list EOF
[2] list ::= list TOK
```

```
=== Viable Prefix Recognizer ===
---lalr_state [0]:
  [list ::= (*) list TOK , {EOF TOK }]
  [$START ::= (*) list EOF , {EOF }]
  [list ::= (*) , {EOF TOK }]
transition on list to state [1]
---lalr_state [1]:
  [list ::= list (*) TOK , {EOF TOK }]
  [$START ::= list (*) EOF , {EOF }]
transition on TOK to state [3]
transition on EOF to state [2]
---lalr_state [2]:
  [$START ::= list EOF (*) , {EOF }]
---lalr_state [3]:
  [list ::= list TOK (*) , {EOF TOK }]
```

```
---- ACTION_TABLE ----
From state #0
 [term 0:REDUCE prod 0]
 [term 2:REDUCE prod 0]
From state #1
 [term 0:SHIFT to state 2]
 [term 2:SHIFT to state 3]
From state #2
 [term 0:REDUCE prod 1]
From state #3
 [term 0:REDUCE prod 2]
 [term 2:REDUCE prod 2]
--- REDUCE_TABLE ----
From state #0
 [non term 0->state 1]
From state #1
From state #2
From state #3
```

4 Automate LR III

- **Même exemple avec « `CUP` *Eclipse Plugin* »**